5XCOS

Framework côté serveur

Thibault Six · Thomas Copin

50 chapitres

SUPPORTS DE COURS

https://cours.tsix.be/cours/framework-cote-serveur

Version 37 — Grille d'évaluation 2026-2027 sur la page d'accueil · 11/10/2026

PRÉSENTATION

Framework côté serveur

Objectifs du cours - Programme

L'étudiant sera capable :

Face au matériel et au logiciel adéquats et face à une structure informatique opérationnelle connectée à Internet, disposant des logiciels appropriés et de la documentation nécessaire, en utilisant le vocabulaire technique et l'orthographe adéquate, et en respectant les normes et standards en vigueur, et au départ d'un cahier des charges et d'un framework proposés par le chargé de cours, de déployer le framework en développant, parmi les concepts suivants :

- la définition des concepts clés de la POO : classe, objet, attributs, méthodes, héritage, ...,
- l'exploitation des « design patterns » (réutilisation,...),
- l'identification et la représentation sous forme schématique des différentes étapes de conception nécessaire à la résolution du projet posé dans le cahier des charges (classe, design patterns,...),
- l'utilisation d'un IDE (Integrated Development Environment),
- d'exploiter les différentes ressources du framework pour la mise en œuvre de l'application,
- les bonnes pratiques en matière de programmation et de sécurisation,
- l'exploitation du modèle MVC (Model View Controller),
- le débogage d'une application,
- la gestion des exceptions et des erreurs,
- l'utilisation de générateurs et assistants,
- la planification et l'exécution des différentes phases de tests,
- d'utiliser à bon escient la documentation disponible.

Grille d'évaluation

- Fiche UE 2026-2027 (PDF) PDF
- Grille d'évaluation 2026-2027 (PDF) PDF

Ce cours et ses documents ont été écrits en collaboration avec Claude Fable 5.1 et GPT-5.6.

Sommaire

- Framework côté serveur

1 Introduction

- 1.1 Lignes de commandes
- 1.2 Syntaxe
- 1.3 Variables
- 1.4 Operateurs
- 1.5 Structures de contrôle
- 1.6 Fonctions de php
- 1.7 Fonctions utilisateurs
- 1.8 Chaines de caractères
- 1.9 Tableaux
- 1.10 Gestion des erreurs
- 1.11 Fichiers
- 1.12 Dates
- 1.13 Exercices complémentaires
- 1.14 Brainfuck

2 POO

- 2.1 Objets et classes
- 2.2 L'atelier : Composer, Pest et la CI
- 2.3 L'encapsulation
- 2.4 Modéliser : le diagramme de classes
- 2.5 Relations entre objets
- 2.6 Héritage et polymorphisme
- 2.7 Contrats : interfaces, traits, enums
- 2.8 Atelier Formes
- 2.9 Atelier Collection
- 2.10 Jouer au Donjon : la classe Game
- 2.11 Lire du code Laravel

3 Framework

3.1 Framework et librairie
3.2 Laravel
3.3 Environnement de développement
3.4 Installation
3.5 MVC
3.6 Structure d'un projet Laravel
3.7 Routing
3.8 Controllers
3.9 Views
3.10 Modèles : une table
3.11 Modèles : deux tables
3.12 Modèles : plusieurs à plusieurs
3.13 Tailwind CSS
3.14 Tailwind CSS : exercices
3.15 Formulaires : créer et valider
3.16 Formulaires : modifier et supprimer
3.17 Le refuge : exercices
3.18 Téléverser une image
3.19 Projet d'examen : Perdu de vue
3.20 Installer un projet Laravel existant
3.21 Hébergement OVH
3.22 Hébergement O2Switch

CHAPITRE 1

Introduction

Présentation

Ce cours aspire à de vous faire découvrir les bases de la programmation orientée objet et de vous initier à l'utilisation d'un framework MVC.

Nous utiliserons le langage PHP pour ce cours dans sa version 8.x.

Le cours se déroulera en 3 parties :

- Révision des bases du PHP
- Programmation orientée objet
- Utilisation d'un framework MVC

Documentation officielle

- PHP
- Composer
- Laravel

Révision des bases du PHP

- Les variables
- Les structures de contrôle
- Appeller les fonctions de PHP
- Les chaînes de caractères
- Écrire nos fonctions
- Les tableaux
- Les fichiers
- Gestions des erreurs
- Les dates

1.1 Introduction au Terminal (CLI) de Laragon

> ***Objectif*** *: Apprendre à naviguer dans le système de fichiers Windows via le terminal, parce que cliquer partout comme un raton laveur hyperactif, c'est 2005.*

Pourquoi Utiliser le Terminal ? 🤔

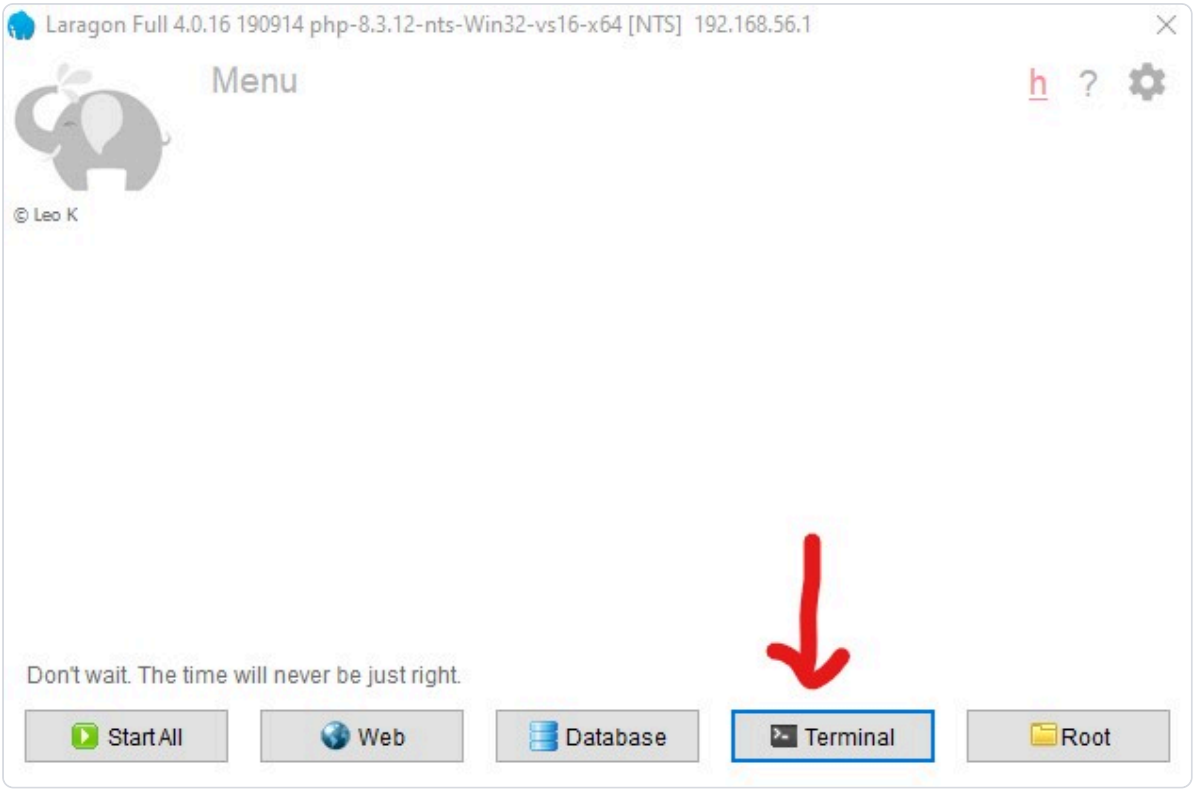
Le terminal (ou CLI - Command Line Interface) permet de :

- ✅ Naviguer **ultra rapidement** entre les dossiers
- ✅ Créer des fichiers/dossiers sans ouvrir 12 fenêtres
- ✅ Exécuter des commandes de dev (npm, git, composer...)
- ✅ Avoir l'air cool devant tes potes (argument principal tbh)

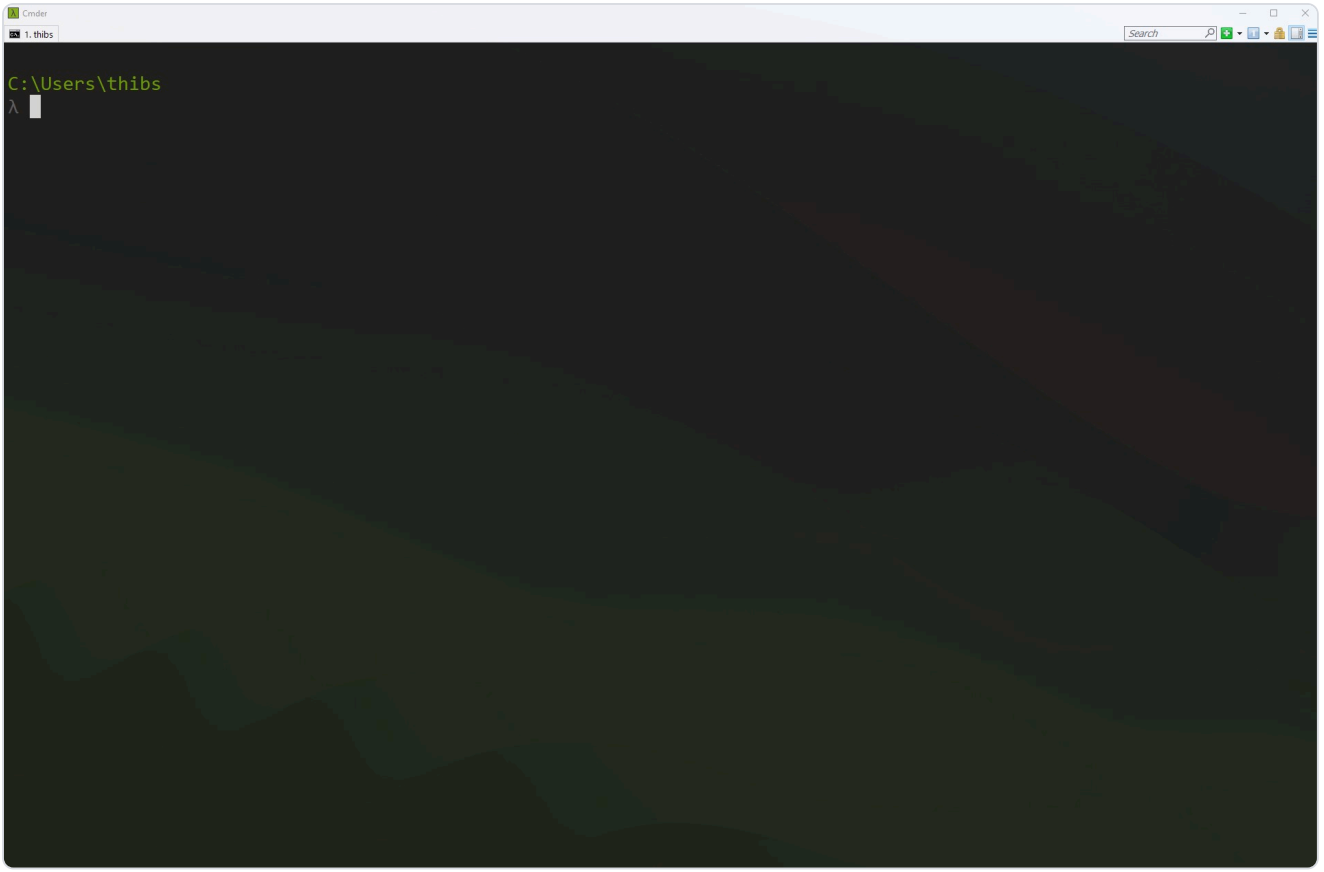
En gros : Si tu veux être efficace et pas perdre 3h par jour à chercher des dossiers, le terminal c'est **LA** compétence de base.

Ouvrir le Terminal de Laragon

Dans Laragon, clique sur le bouton **Terminal** en bas à droite de l'interface.



Une fenêtre noire (ou verte si t'as Cmder) s'ouvre. Bienvenue dans la matrice. 😎



Par défaut, tu es dans ton dossier utilisateur Windows :

```
C:\Users\thibs
λ
```

Le petit symbole `λ` (lambda) indique que le terminal est prêt à recevoir tes commandes.

⚠️ AVANT DE COMMENCER : Les Touches Spéciales

SUPER IMPORTANT : Dans ce cours, tu verras des notations comme `[ENTER]` et `[TAB]`.

- `[ENTER]` = Appuie sur la touche **Entrée** de ton clavier pour valider/exécuter la commande
- `[TAB]` = Appuie sur la touche **TAB** (↹, au-dessus de Caps Lock) pour l'autocomplétion

TU NE TAPES PAS LE MOT "ENTER" OU "TAB", tu appuies physiquement sur ces touches ! ⌨️

Exemple :

```bash
# Ce que tu vois dans le cours :
cd Documents[ENTER]

# Ce que tu fais VRAIMENT :
# 1. Tu tapes : cd Documents
# 2. Tu appuies sur la touche Entrée
```

Maintenant qu'on est clairs, let's go ! 🚀

Commandes de Base pour Naviguer 🧭

1. `cd` - Changer de Dossier (Change Directory)

Syntaxe :

```bash
cd nom_du_dossier
```

Exemples :

```bash
# Aller dans le dossier Documents
cd Documents[ENTER]

# Revenir au dossier parent
cd ..[ENTER]

# Aller directement à la racine C:\
cd \[ENTER]

# Aller dans le dossier Laragon
cd \laragon\www\[ENTER]
```

Rappel : `[ENTER]` = tu appuies sur Entrée pour exécuter la commande !

⚠️ **ATTENTION** : Sur Windows, les noms de dossiers sont **sensibles à la casse** dans certains terminaux.

- `cd documents` ≠ `cd Documents`
- Utilise **TAB** pour l'autocomplétion (voir plus bas)

---

2. `ls` - Lister les Fichiers/Dossiers (List)

Syntaxe :

```bash
ls
```

Exemple :

```bash
C:\Users\thibs
λ ls
Desktop/  Documents/  Downloads/  Pictures/  repos/
```

Affiche tous les fichiers et dossiers du répertoire actuel.

Variantes utiles :

```bash
# Afficher les détails (taille, date)
ls -l

# Afficher les fichiers cachés aussi
ls -a
```

---

3. `mkdir` - Créer un Dossier (Make Directory)

Syntaxe :

```bash
mkdir nom_du_dossier
```

Exemple :

```bash
# Créer un dossier "repos"
mkdir repos

# Créer plusieurs dossiers d'un coup
mkdir projets tests brouillons
```

---

4. `touch` - Créer un Fichier Vide

Syntaxe :

```bash
touch nom_fichier.extension
```

Exemple :

```bash
# Créer un fichier index.html
touch index.html

# Créer plusieurs fichiers
touch style.css script.js
```

Note : `touch` n'existe pas nativement sur Windows, mais fonctionne dans Cmder (le terminal de Laragon). Sinon, utilise :

```bash
echo. > fichier.txt
```

---

5. ENTER - Valider une Commande 🎯

IMPORTANT : Après avoir tapé ta commande, appuie sur **ENTER (Entrée)** pour l'exécuter !

Sans ça, ton terminal va juste te regarder comme un poisson rouge qui attend. 🐟

Exemple :

```bash
C:\Users\thibs
λ cd Documents[ENTER]
# La commande s'exécute
C:\Users\thibs\Documents
λ
```

Note : Dans les exemples de ce cours, on écrit `[ENTER]` pour te rappeler d'appuyer sur Entrée, mais dans la vraie vie tu l'écris pas, tu appuies juste sur la touche. Logique, non ? 😉

---

6. TAB - Autocomplétion Magique ✨

LE TRUC LE PLUS IMPORTANT DE TA VIE : Appuie sur la touche **TAB** (au-dessus de Caps Lock) pour autocompléter les noms de fichiers/dossiers.

C'est quoi [TAB] ?

- `[TAB]` dans les exemples = la touche TAB de ton clavier (↹)
- Tu NE TAPES PAS le mot "TAB", tu APPUIES sur la touche
- C'est comme `[ENTER]`, c'est une TOUCHE, pas du texte

Exemple concret :

```bash
C:\Users\thibs
λ cd Do        ← Tu tapes juste "Do"
λ cd Do↹       ← Tu appuies sur TAB
# Devient automatiquement :
C:\Users\thibs
λ cd Documents\
```

Si plusieurs options existent, appuie sur **TAB** 2 fois pour voir la liste :

```bash
C:\Users\thibs
λ cd \Users\↹↹
Default\  Public\   thibs\
```

Règle d'or : TOUJOURS utiliser TAB. Taper en entier c'est pour les masochistes. 😢

Pourquoi c'est VITAL ?

- ✅ Gain de temps monstrueux

- ✅ Évite les fautes de frappe
- ✅ Gère automatiquement les majuscules/minuscules
- ✅ Te montre ce qui existe vraiment

Le Système de Fichiers Windows (Rappel Rapide)

Structure Typique

```
C:\                          (Racine du disque C:\)
|-- Users\                   (Dossiers utilisateurs)
|   |-- thibs\              (Ton dossier perso)
|   |   |-- Desktop\
|   |   |-- Documents\
|   |   |-- Downloads\
|   |   +-- repos\          (<-- On va créer ça)
|   +-- Public\
|-- Program Files\
+-- laragon\                (<-- Tes projets web ici)
    +-- www\
        |-- projet-1\
        +-- projet-2\
```

Chemins Absolus vs Relatifs

Chemin Absolu (depuis la racine) :

```bash
cd C:\laragon\www\mon-projet
```

Chemin Relatif (depuis où tu es) :

```bash
# Si tu es dans C:\laragon\
cd www\mon-projet
```

Raccourcis :

- `.` = Dossier actuel
- `..` = Dossier parent
- `~` = Dossier utilisateur (C:\Users\thibs)

Exercices Pratiques 💪

RAPPEL : N'oublie JAMAIS d'appuyer sur **ENTER** après chaque commande ! Sinon rien ne se passe. 😄

Exercice 1 : Créer le Dossier `repos`

Objectif : Créer un dossier `repos` dans ton dossier utilisateur.

Étapes :

1. Ouvre le terminal Laragon
2. Vérifie que tu es dans `C:\Users\[ton-nom]\`
3. Liste les dossiers existants avec `ls`
4. Crée le dossier `repos` avec `mkdir repos`
5. Vérifie qu'il existe avec `ls`

Solution :

```bash
C:\Users\thibs
λ ls
Desktop/  Documents/  Downloads/

C:\Users\thibs
λ mkdir repos

C:\Users\thibs
λ ls
Desktop/  Documents/  Downloads/  repos/
```

Exercice 2 : Naviguer jusqu'à Laragon

Objectif : Aller dans le dossier `C:\laragon\www\` en utilisant **TAB** pour l'autocomplétion.

Étapes :

1. Depuis n'importe où, tape `cd \la` puis appuie sur **TAB** → Doit compléter en `cd \laragon\`
2. Continue avec `www\` en tapant `www` puis **TAB**
3. Appuie sur **ENTER**
4. Vérifie avec `ls` que tu vois tes projets web

Solution :

```bash
C:\Users\thibs
λ cd \la[TAB]
# Autocomplétion vers :
λ cd \laragon\

λ cd \laragon\www[TAB]
# Autocomplétion vers :
λ cd \laragon\www\
# Puis tu appuies sur ENTER

C:\laragon\www
λ ls
projet-1/  projet-2/
```

Note : `[TAB]` = appuyer sur la touche TAB, pas taper le mot "TAB". C'est une touche physique de ton clavier ! ⌨️

Exercice 3 : Créer une Structure de Projet

Objectif : Créer un nouveau projet web avec ses fichiers de base.

Étapes :

1. Va dans `C:\laragon\www\`
2. Crée un dossier `mon-premier-site`
3. Entre dans ce dossier
4. Crée les fichiers `index.html`, `style.css`, `script.js`
5. Liste les fichiers pour vérifier

Exercice 4 : Remonter dans l'Arborescence

Objectif : Naviguer entre les dossiers avec `..` (parent).

Situation : Tu es dans `C:\laragon\www\mon-projet\assets\images\`

Questions :

1. Comment revenir dans `assets\` ?
2. Comment revenir directement dans `www\` ?
3. Comment aller à la racine `C:\` ?

Exercice 5 : Challenge Final 🏆

Objectif : Créer cette structure de dossiers dans `C:\Users\[ton-nom]\repos\` :

```
repos/
|-- html-css/
|   |-- tp1/
|   +-- tp2/
|-- javascript/
|   +-- exercices/
+-- projets/
```

Contraintes :

- Utilise **uniquement** le terminal (pas l'explorateur Windows)
- Utilise **TAB** pour l'autocomplétion
- Vérifie chaque création avec `ls`

Récap' des Commandes Essentielles 📋

Commande	Action	Exemple
`cd dossier`	Aller dans un dossier	`cd Documents[ENTER]`
`cd ..`	Remonter d'un niveau	`cd ..[ENTER]`
`cd \`	Aller à la racine	`cd \[ENTER]`
`ls`	Lister fichiers/dossiers	`ls[ENTER]`
`mkdir nom`	Créer un dossier	`mkdir repos[ENTER]`
`touch fichier`	Créer un fichier	`touch index.html[ENTER]`
TAB	Autocomplétion	*Touche physique*
ENTER	Valider/Exécuter	*Touche physique*

Erreurs Fréquentes à Éviter ⚠️

1. Oublier les Majuscules

BASH

```
# ❌ FAUX (si le dossier s'appelle "Documents")
cd documents

# ✅ CORRECT
cd Documents
```

Solution : Utilise TAB pour éviter ce problème.

2. Confondre `/` et `\`

- **Linux/Mac** : `/` (slash)
- **Windows** : `\` (backslash)

```bash
# ❌ FAUX sur Windows
cd C:/laragon/www/

# ✅ CORRECT
cd C:\laragon\www\
```

Note : Sur Cmder (Laragon), les deux fonctionnent souvent, mais respecte la convention Windows.

---

3. Taper Tout à la Main

Non. Juste non. 🤦

Utilise **TAB** ou tu vas perdre 10 ans de ta vie à corriger des typos.

Prochaines Étapes 🚀

Maintenant que tu maîtrises la navigation de base, tu es prêt pour :

- Installer Node.js et npm
- Créer des projets Vue/Nuxt
- Lancer des serveurs de dev

Spoiler : Toutes ces commandes se font... dans le terminal. ðŸ˜Ž

Ressources Complémentaires 📚

- Cheat Sheet Bash - Antisèche complète
- Cmder Documentation - Le terminal de Laragon
- Explainshell - Explique les commandes complexes

1.2 Syntaxe de base

Voici quelques éléments de syntaxe de base de PHP :

- Les instructions sont inclues dans des balises `<?php` et `?>`. Tout ce qui se trouve entre ces balises sera exécuté comme du code PHP.
- Dans un fichier qui ne contient que du code PHP, la balise de fermeture `?>` peut être omise.
- Les commentaires sont précédés de `//` ou `#` pour une ligne ou de `/*` et `*/` pour plusieurs lignes.

- Les variables sont précédées de $. Leur nom peut contenir des lettres, des chiffres et le caractère underscore, mais il ne doit pas commencer par un chiffre.
- Les types de données incluent les nombres (entiers et flottants), les chaînes de caractères, les booléens, les tableaux et les objets.
- Les opérateurs incluent l'addition (+), la soustraction (-), la multiplication (*), la division (/), le modulo (%), l'opérateur d'affectation (=) et bien d'autres encore.
- Les structures de contrôle de flux incluent les instructions if, else, elseif, switch, while, do while et for.
- Les fonctions sont définies avec la fonction function suivie du nom de la fonction et de ses paramètres entre parenthèses. Elles sont appelées en utilisant leur nom suivi de parenthèses et des arguments éventuels.

Voici un exemple simple de code PHP qui utilise quelques-uns de ces éléments de syntaxe :

```php
<?php
// Cette ligne est un commentaire

// Déclaration de variables
nom="John";nom = "John";
nom="John";age = 30;

// Utilisation d'opérateurs et de structures de contrôle de flux
if (age ⩾ 18) {
  echo $nom . " est majeur.\n";
} else {
  echo $nom . " est mineur.\n";
}

// Définition et appel de fonctions
function saluer(nom) {
  echo "Bonjour, " . nom . " !\n";
}

saluer(nom);
?>
```

1.3 Les variables

Une variable est un conteneur pour stocker une valeur de n'importe quel type, comme un nombre entier, un nombre à virgule flottante, une chaîne de caractères, un tableau, etc.

Pour déclarer une variable en PHP, vous devez utiliser le signe dollar `$` suivi du nom de la variable. Le nom de la variable doit commencer par une lettre ou un underscore `_`, suivi de tout nombre de lettres, de chiffres ou d'underscores.

Voici des exemples de déclaration de variables en PHP :

```php
isAdmin=true;isAdmin = true;
isAdmin=true;age = 25;
price=49.99;price = 49.99;
price=49.99;firstName = 'John';
$items = ['item1', 'item2', 'item3'];
```

Il est important de noter que les noms de variables en PHP sont sensibles à la casse, c'est-à-dire que `$name` et `$NAME` sont considérés comme deux variables différentes. De plus, les noms de variables ne peuvent pas commencer par un chiffre. Il est recommandé de nommer les variables en utilisant le camelCase, autrement dit, en commençant par une lettre minuscule et en mettant une majuscule à chaque mot suivant.

Types de variables

En PHP, il y a plusieurs grands types de variables :

- Les variables scalaires, qui contiennent une valeur simple
- Les constantes, qui contiennent une valeur simple et qui ne peuvent pas être modifiées
- Les variables de tableaux, qui contiennent un tableau de valeurs
- Les variables de ressources, qui contiennent une ressource externe
- Les variables d'objets, qui contiennent une instance d'une classe
- et d'autres que l'on verra plus tard

Documentation officielle

Variables scalaires

En PHP, il existe quatre types de variables scalaires, c'est-à-dire qui ne sont pas des tableaux ou des objets :

- **integer** (entier) : pour stocker des nombres entiers, tels que `-1`, `0`, `1`, `2`, etc.
- **float** (nombre à virgule flottante) : pour stocker des nombres à virgule flottante, tels que `3.14`, `-2.718`, etc.
- **string** (chaîne de caractères) : pour stocker des chaînes de caractères, c'est-à-dire des séquences de lettres, de chiffres et de symboles, entourées de `"guillemets"` ou de `'simples quotes'`.
- **boolean** (booléen) : pour stocker des valeurs booléennes, c'est-à-dire `true` (vrai) ou `false` (faux).

Voici quelques exemples de déclaration de variables de chaque type :

```php
age=25;//integerage = 25; // integer
age=25;//integerprice = 49.99; // float
firstName='John';//stringfirstName = 'John'; // string
firstName='John';//stringisAdmin = true; // boolean
```

Il est important de noter que le type de variable en PHP est déterminé automatiquement en fonction de la valeur qui lui est affectée. Par exemple, si vous affectez la chaîne `'25'` à une variable, elle sera considérée comme une chaîne de caractères, même si elle contient un nombre. Si vous souhaitez forcer un type de variable, vous pouvez utiliser les fonctions de type correspondant, telles que `intval()`, `floatval()` et `strval()`.

Voici un exemple d'utilisation de ces fonctions :

```php
age='25';age = '25';
age='25';age = intval(age);//age); // age);//age est maintenant un entier

price='49.99';price = '49.99';
price='49.99';price = floatval(price);//price); // price);//price est maintenant un nombre à virgule flottante

name=25;name = 25;
name=25;name = strval(name);//name); // name);//name est maintenant une chaîne de caractères
```

Constantes

En PHP, une constante est une valeur qui ne peut pas être modifiée une fois qu'elle a été définie. Les constantes sont utiles pour stocker des valeurs qui ne doivent pas être modifiées durant l'exécution du script, comme des valeurs de configuration ou des constantes mathématiques.

Pour déclarer une constante en PHP, vous devez utiliser la fonction `define()`. Voici un exemple de déclaration de constante :

```php
define('PI', 3.14);
```

Le premier argument de `define()` est le nom de la constante, qui doit être une chaîne de caractères. Le second argument est la valeur de la constante.

Une fois que vous avez déclaré une constante, vous pouvez l'utiliser comme une variable normale grâce à son nom, mais vous ne pouvez pas lui affecter une nouvelle valeur. Si vous essayez de le faire, vous obtiendrez une erreur.

Voici un exemple d'utilisation de constante :

```php
echo PI; // affiche 3.14

// Génère une erreur car vous ne pouvez pas réaffecter une valeur à une constante
PI = 3.14159;
```

Il est important de noter que les constantes en PHP sont sensibles à la casse, c'est-à-dire que PI et pi sont considérés comme deux constantes différentes. En général, il est recommandé de suivre les conventions de codage qui consiste à utiliser des majuscules et des underscores pour séparer les mots dans les noms de constantes, comme dans MAX_VALUE ou MIN_PRICE.

Vous pouvez également utiliser la fonction `defined()` pour vérifier si une constante est définie, comme dans cet exemple :

```php
if (defined('PI')) {
  echo "La constante PI est définie";
} else {
  echo "La constante PI n'est pas définie";
}
```

Tableaux

En PHP, un tableau est une variable qui peut stocker plusieurs valeurs de n'importe quel type, organisées sous forme de liste indexée. Vous pouvez accéder à chaque valeur du tableau en utilisant l'index de l'élément, qui est un entier commençant à 0 pour le premier élément.

Pour déclarer un tableau en PHP, vous pouvez utiliser la fonction `array()` ou la syntaxe short-hand `[]` (je préfère que vous utilisiez la seconde notation). Voici quelques exemples de déclaration de tableaux en PHP :

```php
items=array('item1','item2','item3');//contenu:['item1','item2','item3']items =
array('item1', 'item2', 'item3'); // contenu: ['item1', 'item2', 'item3']
items=array('item1','item2','item3');//contenu:['item1','item2','item3']prices = [49.99,
29.99, 39.99]; // contenu: [49.99, 29.99, 39.99]
$mixed = ['item1', 2, 3.14, false]; // contenu: ['item1', 2, 3.14, false]
```

Vous pouvez également déclarer un tableau vide grâce à la syntaxe `array()` ou `[]`, puis ajouter des éléments ultérieurement avec l'opérateur de décalage `[]`. Voici un exemple :

```php
items=array();//contenu:[]items = array(); // contenu: []
items=array();//contenu:[]items[] = 'item1'; // contenu: ['item1']
items[]='item2';//contenu:['item1','item2']items[] = 'item2'; // contenu: ['item1', 'item2']
items[]='item2';//contenu:['item1','item2']items[] = 'item3'; // contenu: ['item1', 'item2',
'item3']


// ou

items=[];//contenu:[]items = []; // contenu: []
items=[];//contenu:[]items[] = 'item1'; // contenu: ['item1']
items[]='item2';//contenu:['item1','item2']items[] = 'item2'; // contenu: ['item1', 'item2']
items[]='item2';//contenu:['item1','item2']items[] = 'item3'; // contenu: ['item1', 'item2',
'item3']
```

Par ailleurs, vous pouvez accéder à un élément du tableau en utilisant son index entre crochets, comme dans cet exemple :

```php
echo items[0];//affiche'item1'echoitems[0]; // affiche 'item1'
echo items[0];//affiche'item1'echoitems[1]; // affiche 'item2'
echo $items[2]; // affiche 'item3'
```

De plus, vous pouvez également utiliser une boucle `foreach` pour parcourir tous les éléments du tableau :

```php
items=['item1','item2','item3'];foreach(items = ['item1', 'item2', 'item3'];
foreach (items=['item1','item2','item3'];foreach(items as $item) {
  echo $item;
}
// affiche 'item1item2item3'
```

Il est important de noter que les tableaux en PHP peuvent également être associatifs, c'est-à-dire qu'ils peuvent utiliser n'importe quelle chaîne de caractères comme index au lieu d'un entier. Voici un exemple de déclaration de tableau associatif :

```php
$prices = array(
  'item1' => 49.99,
  'item2' => 29.99,
  'item3' => 39.99
);

// contenu: [
//   'item1' => 49.99,
//   'item2' => 29.99,
//   'item3' => 39.99
// ]
```

Vous pouvez accéder à un élément du tableau associatif en utilisant sa clé entre crochets, comme dans cet exemple :

```php
echo prices['item1'];//affiche49.99echoprices['item1']; // affiche 49.99
echo prices['item1'];//affiche49.99echoprices['item2']; // affiche 29.99
echo $prices['item3']; // affiche 39.99
```

Vous pouvez également utiliser une boucle `foreach` pour parcourir tous les éléments du tableau associatif :

```php
prices=['item1'⇒49.99,'item2'⇒29.99,'item3'⇒39.99];foreach(prices = [
  'item1' ⇒ 49.99,
  'item2' ⇒ 29.99,
  'item3' ⇒ 39.99
];
foreach (prices=['item1'⇒49.99,'item2'⇒29.99,'item3'⇒39.99];foreach(prices as key⇒key ⇒
key⇒value) {
  echo key.":".key . " : " . key.":".value . "\n";
}
// affiche :
// item1 : 49.99
// item2 : 29.99
// item3 : 39.99
```

Les ressources

En PHP, une ressource est un type de variable qui représente un handle externe, comme un fichier ouvert, une connexion à une base de données, etc. Les ressources sont gérées de manière automatique par PHP et ne nécessitent pas d'allocation ou de libération de mémoire explicite.

Pour déclarer une variable de type ressource en PHP, vous devez utiliser une fonction qui retourne une ressource, comme `fopen()` pour ouvrir un fichier, `mysqli_connect()` pour établir une connexion à une base de données MySQL, etc. Voici un exemple de déclaration de variable de type ressource :

```php
$file = fopen('filename.txt', 'r'); // ouvre le fichier filename.txt en lecture
```

Vous pouvez utiliser la fonction `is_resource()` pour vérifier si une variable est une ressource, comme dans cet exemple :

```php
if (is_resource($file)) {
  echo "La variable \$file est une ressource";
} else {
  echo "La variable \$file n'est pas une ressource";
}
```

Il est important de noter que vous devez fermer les ressources lorsque vous n'en avez plus besoin, en utilisant la fonction correspondante, comme `fclose()` pour fermer un fichier ou `mysqli_close()` pour fermer une connexion à une base de données MySQL. Si vous oubliez de fermer les ressources, vous risquez de gaspiller des ressources système et de provoquer des fuites de mémoire.

Voici un exemple de fermeture de ressource :

```php
fclose($file);
```

Les objets

En PHP, une classe est un modèle qui définit les propriétés et les comportements d'un objet. Elle peut être utilisée pour créer de nouveaux objets de ce type.

Un objet est une instance d'une classe. Il a ses propres propriétés et comportements, définis par la classe à laquelle il appartient.

Voici un exemple simple de définition d'une classe et de création d'un objet en PHP :

```PHP
class Voiture {
  public marque;publicmarque;
  public marque;publiccouleur;
  public nb_roues;

  public function klaxonner() {
- echo "Tut tut !\n";
  }
}

ma_voiture = new Voiture();
mavoiture->marque="Dacia";ma_voiture→marque = "Dacia";
mavoiture->marque="Dacia";ma_voiture→couleur = "bleu";
mavoiture->nbroues=4;echo"Mavoitureestune".ma_voiture→nb_roues = 4;

echo "Ma voiture est une " . mavoiture->nbroues=4;echo"Mavoitureestune".ma_voiture→marque .
" de couleur " . mavoiture->couleur."\n";ma_voiture→couleur . "\n";
mavoiture->couleur."\n";ma_voiture→klaxonner();
```

Dans cet exemple, la classe Voiture définit trois propriétés (marque, couleur et nb_roues) et une méthode (klaxonner). Nous créons un objet de cette classe avec la syntaxe `$ma_voiture = new Voiture();`, puis nous définissons les valeurs de ses propriétés et nous appelons sa méthode.

Il est important de noter que les propriétés et les méthodes d'un objet sont accessibles en utilisant l'opérateur flèche `→`, comme dans cet exemple :

```PHP
mavoiture->marque="Dacia";ma_voiture→marque = "Dacia";
mavoiture->marque="Dacia";ma_voiture→couleur = "bleu";
mavoiture->nbroues=4;ma_voiture→nb_roues = 4;
mavoiture->nbroues=4;ma_voiture→klaxonner();
```

Il existe des classes prédéfinies dans PHP, comme DateTime, PDO, etc. Vous pouvez également créer vos propres classes et les utiliser dans vos scripts PHP.

Par exemplen, le PDO (PHP Data Objects) est une classe de base de PHP qui permet de se connecter à une base de données et d'exécuter des requêtes SQL. Elle offre une interface commune pour accéder à différents types de bases de données (comme MySQL, PostgreSQL, Oracle, etc.), ce qui facilite l'écriture de code portable.

Voici comment utiliser la classe PDO pour se connecter à une base de données MySQL et exécuter une requête de sélection :

```php
try {
  // Connexion à la base de données
  pdo = new PDO("mysql:host=localhost;dbname=nom_de_la_base", "nom_d_utilisateur",
"mot_de_passe");

  // Exécution d'une requête de sélection
  stmt =
pdo->query("SELECT*FROMutilisateursWHEREnom='John'");//Re´cupe´rationetaffichagedesre´sultat
swhile(pdo→query("SELECT * FROM utilisateurs WHERE nom = 'John'");

  // Récupération et affichage des résultats
  while
(pdo->query("SELECT*FROMutilisateursWHEREnom='John'");//Re´cupe´rationetaffichagedesre´sulta
tswhile(row = stmt→fetch()) {
- echo $row['nom'] . " " . $row['prenom'] . "\n";
  }
} catch (PDOException e) {
  // Gestion des erreurs
  echo "Erreur : " . $e→getMessage() . "\n";
}
```

Dans cet exemple, nous créons un objet PDO en passant les informations de connexion à la base de données dans le constructeur de la classe. Nous utilisons ensuite la méthode query pour exécuter une requête de sélection, puis nous récupérons les résultats en utilisant la méthode fetch. Enfin, nous gérons les erreurs en utilisant un bloc try/catch.

Il existe de nombreuses autres méthodes et propriétés de la classe PDO qui peuvent être utilisées pour exécuter d'autres types de requêtes, préparer des requêtes à l'avance, obtenir des informations sur la base de données, etc. Vous pouvez en savoir plus sur la classe PDO dans la documentation de PHP.

Variables prédéfinies

En PHP, il existe des variables prédéfinies disponibles dans tous les scripts et qui contiennent des informations sur l'environnement d'exécution, le serveur, la requête HTTP, les variables de session, etc. Voici une liste de quelques-unes des variables prédéfinies les plus couramment utilisées en PHP :

- $_SERVER : contient des informations sur l'environnement d'exécution du script, comme le nom du serveur, le nom du script en cours d'exécution, le chemin du script, etc.
- $_GET : contient les données envoyées dans la requête HTTP via la méthode GET.
- $_POST : contient les données envoyées dans la requête HTTP via la méthode POST.
- $_FILES : contient les informations sur les fichiers téléchargés via une requête HTTP.
- $_COOKIE : contient les informations sur les cookies envoyés avec la requête HTTP.
- $_SESSION : contient les informations sur les variables de session en cours.
- $_ENV : contient les informations sur les variables d'environnement du serveur.

Voici un exemple d'utilisation de la variable prédéfinie `$_SERVER` pour afficher le nom du serveur :

```php
echo 'Le nom du serveur est : ' . $_SERVER['SERVER_NAME'];
```

Il existe également d'autres variables prédéfinies, comme `$_REQUEST`, `$_GLOBALS`, etc. Vous pouvez trouver la liste complète des variables prédéfinies en PHP dans la documentation officielle de PHP.

1.4 Les opérateurs

Voici la liste des types d'opérateurs disponibles en PHP, regroupés par catégorie :

Opérateurs unaires

- opérateur de négation logique `!`
- opérateurs d'incrémentation `++` et de décrémentation `--`

Les opérateurs unaires sont des opérateurs qui n'ont qu'un seul opérande. En PHP, il existe plusieurs types d'opérateurs unaires, notamment l'opérateur de négation logique `!` et les opérateurs d'incrémentation/décrémentation `++` et `--`.

L'opérateur de négation logique `!` permet de changer la valeur booléenne d'une expression. Si l'expression est `true`, l'opérateur renverra `false` et vice versa. Par exemple :

```php
a=true;a = true;
a=true;b = !a;//a; // a;//b vaut false
```

Les opérateurs d'incrémentation `++` et de décrémentation `--` permettent respectivement d'ajouter ou de soustraire 1 à la valeur d'une variable. Par exemple :

```php
a=5;a = 5;
a=5;b = ++a;//a; // a;//a vaut 6 et bvaut6b vaut 6

bvaut6c = 10;
d=--d = --d=--c; // cvaut9etc vaut 9 et cvaut9etd vaut 9
```

Il est important de noter que l'ordre dans lequel l'opérateur d'incrémentation ou de décrémentation est placé par rapport à la variable peut avoir des conséquences sur le résultat de l'opération. Si l'opérateur est placé avant la variable (préfixé), la variable est modifiée avant que sa valeur ne soit utilisée dans l'expression. Si l'opérateur est placé après la variable (suffixé), la variable est modifiée après que sa valeur a été utilisée dans l'expression. Par exemple :

```php
a=5;a = 5;
a=5;b = a++;//a++; // a++;//a vaut 6 et bvaut5b vaut 5

bvaut5c = 10;
d=d = d=c--; // cvaut9etc vaut 9 et cvaut9etd vaut 10
```

Opérateurs binaires

- Opérateurs arithmétiques : `+`, `-`, `*`, `/`, `%`, `**`
- Opérateurs de comparaison : `⩵`, `≠`, `>`, `<`, `⩾`, `⩽`, `≡`, `≢`
- Opérateurs de chaînage de comparaison, aussi appelé spaceship operator : `⟺`
- Opérateurs logiques : `&&`, `||`, `!`
- Opérateurs d'affectation : `=`, `+=`, `-=`, `*=`, `=`, `%=`, `•=`, `&=`, `⊨`, `^=`, `⇐=`, `»=`

Les opérateurs binaires sont des opérateurs qui ont deux opérandes. En PHP, il existe plusieurs types d'opérateurs binaires, notamment :

Les opérateurs arithmétiques : `+`, `-`, `*`, `/`, `%` et `**`. Ces opérateurs permettent d'effectuer des opérations mathématiques, comme l'addition, la soustraction, la multiplication, la division, le modulo et l'exponentiation. Par exemple :

```php
a=5;a = 5;
a=5;b = 10;
c=c = c=a + b;//b; // b;//c vaut 15
d=d = d=b - a;//a; // a;//d vaut 5
e=e = e=a * b;//b; // b;//e vaut 50
f=f = f=b / a;//a; // a;//f vaut 2
g=g = g=b % a;//a; // a;//g vaut 0
h=h = h=a \*\* b;//b; // b;//h vaut 9765625
```

Les opérateurs de comparaison : `⩵`, `≠`, `>`, `<`, `⩾`, `⩽`, `≡`, `≢`. Ces opérateurs permettent de comparer deux valeurs et de renvoyer `true` ou `false` en fonction du résultat de la comparaison. Par exemple :

```php
a=5;a = 5;
a=5;b = 10;
c="5";vardump(c = "5";

var_dump(c="5";vardump(a ⩵ b);//falsevardump(b); // false
var_dump(b);//falsevardump(a ≠ b);//truevardump(b); // true
var_dump(b);//truevardump(a > b);//falsevardump(b); // false
var_dump(b);//falsevardump(a < b);//truevardump(b); // true
var_dump(b);//truevardump(a ⩾ b);//falsevardump(b); // false
var_dump(b);//falsevardump(a ⩽ b);//truevardump(b); // true
var_dump(b);//truevardump(a ≡ c);//falsevardump(c); // false
var_dump(c);//falsevardump(a ≢ $c); // true
```

L'opérateur de chaînage de comparaison ⇔. Cet opérateur permet de comparer deux valeurs et de renvoyer -1 si la première valeur est inférieure à la seconde, 0 si les deux valeurs sont égales et 1 si la première valeur est supérieure à la seconde. Par exemple :

```php
a=5;a = 5;
a=5;b = 10;
c="5";vardump(c = "5";

var_dump(c="5";vardump(a ⟺ b);//-1vardump(b); // -1
var_dump(b);//-1vardump(b ⟺ a);//1vardump(a); // 1
var_dump(a);//1vardump(a ⟺ $c); // 0
```

Les opérateurs logiques : `&&`, `||` et `!`. Ces opérateurs permettent de combiner des expressions booléennes et de renvoyer `true` ou `false` en fonction du résultat de la combinaison. Par exemple :

```php
a=true;a = true;
a=true;b = false;

var_dump(a && b); // false
var_dump(a||a || a||b); // true
var_dump(!$b); // true
```

Opérateurs ternaires

Opérateur ternaire : `? :`

L'opérateur ternaire est un opérateur de condition qui permet de simplifier l'écriture de code en remplaçant une instruction if et else. Il se présente sous la forme suivante `condition ? expression1 : expression2`. Si la condition est vraie, l'expression1 sera évaluée et retournée, sinon l'expression2 sera évaluée et retournée. Voici un exemple d'utilisation de l'opérateur ternaire :

```php
age=18;age = 18;
age=18;autorisation = (age⩾18)?"Vouse^tesautorise´":"Vousn'e^tespasautorise´";echoage ⩾
18) ? "Vous êtes autorisé" : "Vous n'êtes pas autorisé";
echo age⩾18)?"Vouse^tesautorise´":"Vousn'e^tespasautorise´";echoautorisation;
```

Dans cet exemple, si l'âge est supérieur ou égal à 18, la chaîne de caractères "Vous êtes autorisé" sera retournée, sinon la chaîne "Vous n'êtes pas autorisé" sera retournée.

Il est déconseillé d'utiliser l'opérateur ternaire pour des conditions trop complexes, car cela rend le code difficile à lire. Dans ce cas, il est préférable d'utiliser une instruction if et else.

Opérateur Elvis

Opérateur Elvis : `?:`

L'opérateur Elvis est un opérateur de condition qui permet de simplifier l'écriture de code en remplaçant une instruction if et else. Il se présente sous la forme suivante `expression1 ?: expression2`. Si `expression1` est vraie, `expression1` sera retournée, sinon `expression2` retournée. Il est utile pour définir une valeur par défaut pour une variable. Les deux notations suivantes sont équivalentes :

```php
// Opérateur ternaire
text=(text = (text=(text !== null) ? text:"default";//Ope´rateurElvistext : "default";

// Opérateur Elvis
text:"default";//Ope´rateurElvistext = $text ?: "default";
```

Opérateurs sur les bits

Opérateurs de décalage de bits : `<<`, `>>` Opérateurs de masquage de bits : `&`, `|`, `^` Opérateur de complément de bits : `~`

Les opérateurs de décalage de bits permettent de déplacer les bits d'un nombre binaire vers la gauche (opérateur `<<`) ou vers la droite (opérateur `>>`). Par exemple, si on utilise l'opérateur `<<` avec un nombre binaire `0101` et un décalage de deux bits, le résultat sera `010100`. Si on utilise l'opérateur `>>` avec le même nombre binaire et un décalage de deux bits, le résultat sera `0001`.

Les opérateurs de masquage de bits permettent de masquer certaines parties d'un nombre binaire en utilisant des masques binaires. L'opérateur `&` permet de faire un ET logique entre deux nombres binaires, l'opérateur `|` permet de faire un OU logique et l'opérateur `^` permet de faire un OU exclusif. Par exemple, si on utilise l'opérateur `&` avec le nombre binaire `0101` et le masque binaire `0011`, le résultat sera `0001`.

L'opérateur de complément de bits `~` permet de inverser tous les bits d'un nombre binaire. Par exemple, si on utilise l'opérateur `~` sur le nombre binaire `0101`, le résultat sera `1010`.

Il est important de noter que ces opérateurs ne sont utilisables que sur des entiers non signés (unsigned integers).

Opérateurs sur les chaînes de caractères

Opérateur de concaténation : `.`

L'opérateur de concaténation `.` permet de concaténer, c'est-à-dire de fusionner, deux chaînes de caractères en PHP. Par exemple :

```php
chaine1="Bonjour";chaine1 = "Bonjour";
chaine1="Bonjour";chaine2 = " tout le monde";
resultat=resultat = resultat=chaine1 . chaine2;//chaine2;
// chaine2;//resultat vaut "Bonjour tout le monde"
```

Il est également possible de concaténer des variables et des expressions dans une chaîne de caractères en utilisant la syntaxe suivante :

```php
prenom="John";prenom = "John";
prenom="John";age = 30;
echo "Bonjour, je m'appelle prenometj'aiprenom et j'ai prenometj'aiage ans.";
// Affiche "Bonjour, je m'appelle John et j'ai 30 ans."
```

Il est également possible de concaténer des chaînes de caractères avec l'opérateur `•=`, qui permet d'ajouter une chaîne de caractères à la fin d'une autre chaîne :

```php
chaine="Bonjour";chaine = "Bonjour";
chaine="Bonjour";chaine •= " tout le monde";
// $chaine vaut "Bonjour tout le monde"
```

Opérateurs sur les tableaux

Opérateur d'union de tableaux : `+` Opérateurs de comparaison : `=`, `≠`, `≡`, `≢` Opérateur d'accès à un élément de tableau : `[]`

L'opérateur d'union de tableaux `+` permet de fusionner deux tableaux en PHP. Le résultat est un nouveau tableau qui contient tous les éléments des deux tableaux originaux. Si les deux tableaux ont des clés identiques, les éléments du second tableau remplaceront ceux du premier tableau. Par exemple :

```php
tableau1=["a","b","c"];tableau1 = ["a", "b", "c"];
tableau1=["a","b","c"];tableau2 = ["d", "e", "f"];
resultat=resultat = resultat=tableau1 + tableau2;//tableau2;
// tableau2;//resultat vaut ["a", "b", "c", "d", "e", "f"]
```

Il existe également plusieurs opérateurs de comparaison pour les tableaux en PHP. L'opérateur `=` permet de vérifier si deux tableaux ont les mêmes valeurs, l'opérateur `≠` permet de vérifier s'ils ont des valeurs différentes, l'opérateur `≡` permet de vérifier si deux tableaux ont les mêmes valeurs et sont de type identique (tableau) et l'opérateur `≢` permet de vérifier s'ils ont des valeurs différentes ou ne sont pas de type identique (tableau). Par exemple :

```php
tableau1=[];tableau1 = [];
tableau1=[];tableau2 = [];
tableau3=null;tableau3 = null;
tableau3=null;tableau4 = false;

var_dump(tableau1==tableau1 == tableau1==tableau2); // Affiche "bool(true)"
var_dump(tableau1≠tableau1 ≠ tableau1≠tableau2); // Affiche "bool(false)"
var_dump(tableau1===tableau1 === tableau1===tableau2); // Affiche "bool(true)"
var_dump(tableau1≠=tableau1 ≠= tableau1≠=tableau2); // Affiche "bool(false)"

var_dump(tableau1==tableau1 == tableau1==tableau3); // Affiche "bool(true)"
var_dump(tableau1≠tableau1 ≠ tableau1≠tableau3); // Affiche "bool(false)"
var_dump(tableau1===tableau1 === tableau1===tableau3); // Affiche "bool(false)"
var_dump(tableau1≠=tableau1 ≠= tableau1≠=tableau3); // Affiche "bool(true)"

var_dump(tableau1==tableau1 == tableau1==tableau4); // Affiche "bool(true)"
var_dump(tableau1≠tableau1 ≠ tableau1≠tableau4); // Affiche "bool(false)"
var_dump(tableau1===tableau1 === tableau1===tableau4); // Affiche "bool(false)"
var_dump(tableau1≠=tableau1 ≠= tableau1≠=tableau4); // Affiche "bool(true)"
```

Enfin, l'opérateur d'accès à un élément de tableau `[]` permet de lire ou de modifier un élément d'un tableau en utilisant sa clé. Par exemple :

```php
tableau=["a","b","c"];echotableau = ["a", "b", "c"];
echo tableau=["a","b","c"];echotableau[0]; // Affiche "a"
$tableau[1] = "d"; // Modifie la valeur de l'élément de clé 1 du tableau
```

Exercices de lecture

Ternaires

1.1. Donnez le résultat de l'exécution du programme suivant :

```php
number=25;number = 25;
number=25;result = (numberechonumber % 2 == 0) ? "even" : "odd";
echo numberechoresult;
```

1.2. Donnez le résultat de l'exécution du programme suivant :

```php
age=18;age = 18;
age=18;can_vote = (age⩾18)?true:false;age ⩾ 18) ? true : false;
age⩾18)?true:false;can_vote = ($age ⩾ 18);
```

1.3. Donnez le résultat de l'exécution du programme suivant :

```php
studentscore=75;student_score = 75;
studentscore=75;passing_score = 60;
result=(result = (result=(student_score ⩾
passingscore)?"Pass":"Fail";echo"Studentscore:".passing_score) ? "Pass" : "Fail";

echo "Student score: " . passingscore)?"Pass":"Fail";echo"Studentscore:".student_score . "
<br>";
echo "Passing score: " . passingscore."<br>";echo"Result:".passing_score . "<br>";
echo "Result: " . passingscore."<br>";echo"Result:".result;

if(studentscore>100||student_score > 100 || studentscore>100||student_score < 0)
{
    echo "Invalid score";
    $result = "Invalid";
}
```

Elvis Operator

2.1. Donnez le résultat de l'exécution du programme suivant :

```php
studentname="John";student_name = "John";
studentname="John";student_age = 25;

echo "Name: " . studentname?:"Unknown"."\n";echo"Age:".student_name ?: "Unknown" . "\n";
echo "Age: " . studentname?:"Unknown"."\n";echo"Age:".student_age ?: "Unknown" . "\n";
```

2.2. Donnez le résultat de l'exécution du programme suivant :

```php
student=["name"⇒"John","age"⇒25,"course"⇒"ComputerScience"];student = ["name" ⇒ "John",
"age" ⇒ 25, "course" ⇒ "Computer Science"];

student=["name"⇒"John","age"⇒25,"course"⇒"ComputerScience"];name =
student["name"]?:"Unknown";student["name"] ?: "Unknown";
student["name"]?:"Unknown";age = student["age"]?:"Unknown";student["age"] ?: "Unknown";
student["age"]?:"Unknown";course =
student["course"]?:"Unknown";echo"Name:".student["course"] ?: "Unknown";

echo "Name: " . student["course"]?:"Unknown";echo"Name:".name . "\n";
echo "Age: " . age."\n";echo"Course:".age . "\n";
echo "Course: " . age."\n";echo"Course:".course . "\n";

if (!isset(student["city"])) {
    $student["city"] = "Unknown";
}

city = student["city"]?:"N/A";echo"City:".student["city"] ?: "N/A";
echo "City: " . student["city"]?:"N/A";echo"City:".city . "\n";
```

Null Coalescing Operator

3.1. Donnez le résultat de l'exécution du programme suivant :

```php
studentname="John";student_name = "John";
studentname="John";student_age = 25;

echo "Name: " . studentname??"Unknown"."\n";echo"Age:".student_name ?? "Unknown" . "\n";
echo "Age: " . studentname??"Unknown"."\n";echo"Age:".student_age ?? "Unknown" . "\n";
```

3.2. Donnez le résultat de l'exécution du programme suivant :

```php
student=["name"⇒"John","age"⇒25];student = ["name" ⇒ "John", "age" ⇒ 25];

student=["name"⇒"John","age"⇒25];name = student["name"]??"Unknown";student["name"] ??
"Unknown";
student["name"]??"Unknown";age = student["age"]??"Unknown";student["age"] ?? "Unknown";

student["age"]??"Unknown";course = student["course"]??null;student["course"] ?? null;
student["course"]??null;grade = student["grade"]??null;student["grade"] ?? null;

student["grade"]??null;course = course??"NotEnrolled";course ?? "Not Enrolled";
course??"NotEnrolled";grade =?? "N/A";

echo "Name: " . name."\n";echo"Age:".name . "\n";
echo "Age: " . name."\n";echo"Age:".age . "\n";
echo "Course: " . course."\n";echo"Grade:".course . "\n";
echo "Grade: " . course."\n";echo"Grade:".grade . "\n";
```

Exercices

Exercice 1 : Calcul de la somme

Écrivez un programme en PHP qui demande à l'utilisateur de saisir deux nombres entiers et qui affiche la somme de ces deux nombres.

Voici les étapes à suivre pour réaliser cet exercice :

1. Demandez à l'utilisateur de saisir le premier nombre et stockez-le dans une variable.
2. Demandez à l'utilisateur de saisir le second nombre et stockez-le dans une autre variable.
3. Calculez la somme des deux nombres avec l'opérateur d'addition.
4. Affichez le résultat de la somme à l'utilisateur.

Voici un exemple de début du programme :

```php
<?php
// Demande à l'utilisateur de saisir le premier nombre
echo "Entrez le premier nombre : ";
premierNombre=readline();vardump(premierNombre = readline();

var_dump(premierNombre=readline();vardump(premierNombre);
```

Exercice 2 : Calcul du périmètre et de l'aire d'un rectangle

Écrivez un programme en PHP qui demande à l'utilisateur de saisir la longueur et la largeur d'un rectangle, puis qui calcule et affiche le périmètre et l'aire de ce rectangle.

Voici les étapes à suivre pour réaliser cet exercice :

1. Demandez à l'utilisateur de saisir la longueur du rectangle et stockez-la dans une variable.
2. Demandez à l'utilisateur de saisir la largeur du rectangle et stockez-la dans une autre variable.
3. Calculez le périmètre du rectangle.
4. Calculez l'aire du rectangle.
5. Affichez le périmètre et l'aire du rectangle à l'utilisateur.

Exercice 3 : Swap de variables

Écrivez un programme en PHP qui demande à l'utilisateur de saisir deux valeurs et qui effectue un échange (swap) des valeurs de ces deux variables.

Voici les étapes à suivre pour réaliser cet exercice :

1. Demandez à l'utilisateur de saisir la première valeur et stockez-la dans une variable.
2. Demandez à l'utilisateur de saisir la deuxième valeur et stockez-la dans une autre variable.
3. Affichez les valeurs initiales des deux variables.
4. Effectuez l'échange des valeurs.
5. Affichez les nouvelles valeurs des deux variables après l'échange.

Exercice 4 : Utilisation des expressions ternaires

Écrivez un programme en PHP qui demande à l'utilisateur de saisir un nombre entier, puis qui affiche si ce nombre est pair ou impair en utilisant une expression ternaire.

Voici les étapes à suivre pour réaliser cet exercice :

1. Demandez à l'utilisateur de saisir un nombre entier et stockez-le dans une variable.
2. Utilisez une expression ternaire pour déterminer si le nombre est pair ou impair.
3. Affichez le résultat à l'utilisateur.

1.5 Les structures de contrôle

Les structures de contrôle permettent de contrôler l'exécution d'un script. Elles permettent de tester une condition et d'exécuter un bloc de code si la condition est vraie, ou de répéter un bloc de code un nombre précis de fois.

Voici une liste des principales structures de contrôle en PHP :

1. `if` : permet de tester une condition et d'exécuter un bloc de code si la condition est vraie.
2. `if...else` : permet de tester une condition et d'exécuter un premier bloc de code si la condition est vraie, et un second bloc de code si la condition est fausse.
3. `if...elseif...else` : permet de tester plusieurs conditions et d'exécuter un bloc de code correspondant à la première condition qui est vraie. Si aucune condition n'est vraie, un dernier bloc de code peut être exécuté.
4. `switch` : permet de tester plusieurs valeurs possibles pour une expression et d'exécuter un bloc de code correspondant à la première valeur qui correspond.
5. `match` : permet de tester une valeur et d'exécuter un bloc de code correspondant à la première valeur qui correspond. Cette structure est disponible à partir de PHP 8.0.
6. `while` : permet de répéter un bloc de code tant qu'une condition est vraie.
7. `do...while` : permet de répéter un bloc de code au moins une fois, puis de continuer à le répéter tant qu'une condition est vraie.
8. `for` : permet de répéter un bloc de code un certain nombre de fois en spécifiant une condition de fin et un compteur.
9. `foreach` : permet de parcourir les éléments d'un tableau ou d'un objet et de les traiter un à un.

Les conditions

Les conditions permettent de tester une valeur ou une expression et d'exécuter un bloc de code si la condition est vraie.

La structure if

La structure if permet de tester une condition et d'exécuter un bloc de code si cette condition est vraie. La syntaxe de base de la structure if est la suivante :

```php
if (condition) {
// code à exécuter si la condition est vraie
}
```

Par exemple, si on veut vérifier si une variable `$age` est supérieure à 18 et afficher un message en conséquence, on peut utiliser la structure if de la façon suivante :

```php
if ($age > 18) {
echo "Vous êtes majeur.";
}
```

Il est également possible d'ajouter une clause `else` pour exécuter un bloc de code si la condition est fausse :

```php
if ($age > 18) {
echo "Vous êtes majeur.";
} else {
echo "Vous êtes mineur.";
}
```

Il est possible d'ajouter autant de clauses `elseif` que nécessaire pour tester plusieurs conditions :

```php
if (age>18)echo"Vouse^tesmajeur.";elseif(age > 18) {
    echo "Vous êtes majeur.";
} elseif (age>18)echo"Vouse^tesmajeur.";elseif(age == 18) {
echo "Vous avez 18 ans.";
} else {
echo "Vous êtes mineur.";
}
```

la structure switch

La structure `switch` permet de tester plusieurs valeurs possibles pour une expression et d'exécuter un bloc de code correspondant à la première valeur qui correspond. Elle peut être utilisée pour remplacer plusieurs instructions `if...elseif...else`.

Voici la syntaxe de base de la structure `switch` :

```php
switch (expression) {
    case valeur1:
        // code à exécuter si expression vaut valeur1
        break;
    case valeur2:
        // code à exécuter si expression vaut valeur2
        break;
    // ...
    default:
        // code à exécuter si aucune des valeurs ne correspond
        break;
}
```

L'instruction `break` permet de quitter la structure switch une fois que le bloc de code correspondant à la valeur de l'expression a été exécuté. Si break n'est pas utilisé, le code suivant sera également exécuté.

Voici un exemple d'utilisation de la structure `switch` pour afficher un message en fonction de la valeur d'une variable `$note` :

```php
switch ($note) {
    case 0:
    case 1:
    case 2:
        echo "Insuffisant.";
        break;
    case 3:
    case 4:
        echo "Moyen.";
        break;
    case 5:
    case 6:
        echo "Assez bien.";
        break;
    case 7:
    case 8:
    case 9:
        echo "Bien.";
        break;
    case 10:
        echo "Très bien.";
        break;
    default:
        echo "Note non valide.";
        break;
}
```

Lorsqu'une structure `switch` est exécutée, l'expression spécifiée est évaluée et sa valeur est comparée aux valeurs des différents case. Si une valeur correspond, le bloc de code associé est exécuté. Si aucune valeur ne correspond, le bloc de code associé à la clause `default` est exécuté (si elle est présente).

La structure match

La structure `match` permet de tester une valeur et d'exécuter un bloc de code correspondant à la première valeur qui correspond. Elle est similaire à la structure `switch`, mais offre une syntaxe différente et de nouvelles fonctionnalités.

Voici la syntaxe de base de la structure match :

```php
match (expression) {
    valeur1 => bloc de code,
    valeur2 => bloc de code,
    // ...
    default => bloc de code,
}
```

Voici un exemple d'utilisation de la structure match pour afficher un message en fonction de la valeur d'une variable `$note` :

```php
match ($note) {
    0 | 1 | 2  => echo "Insuffisant.",
    3 | 4      => echo "Moyen.",
    5 | 6      => echo "Assez bien.",
    7 | 8 | 9  => echo "Bien.",
    10         => echo "Très bien.",
    default    => echo "Note non valide.",
}
```

La structure `match` est souvent plus appropriée que la structure `switch` lorsque l'expression à tester est complexe et que la comparaison de valeurs doit se faire à l'aide d'opérateurs de comparaison ou de fonctions de test.

Par exemple, si on veut vérifier si une variable `$x` est supérieure à `10` et inférieure à `20`, on peut utiliser la structure match de la façon suivante :

```php
match ($x) {
    $x > 10 and $x < 20   => echo "x est compris entre 10 et 20.",
    default               => echo "x n'est pas compris entre 10 et 20.",
}
```

La structure `match` est disponible à partir de PHP 8.0.

Les boucles

Les boucles permettent d'exécuter plusieurs fois un bloc de code. Elles sont très utiles pour traiter des tableaux.

Voici une liste des principales boucles disponibles en PHP :

- `while` : permet de répéter un bloc de code tant qu'une condition est vraie.
- `do...while` : permet de répéter un bloc de code au moins une fois, puis de continuer à le répéter tant qu'une condition est vraie.
- `for` : permet de répéter un bloc de code un certain nombre de fois en spécifiant une condition de fin et un compteur.
- `foreach` : permet de parcourir les éléments d'un tableau ou d'un objet et de les traiter un à un.
- `goto` : permet de sauter à un endroit précis du code. Elle est déconseillée, car elle rend le code difficile à lire et à maintenir. Elle est également considérée comme une mauvaise pratique en programmation.

Source : https://xkcd.com/292/

La boucle while

La boucle `while` permet de répéter un bloc de code tant qu'une condition est vraie. La syntaxe de base de la boucle `while` est la suivante :

```php
while (condition) {
    // code à exécuter tant que la condition est vraie
}
```

Voici un exemple d'utilisation de la boucle `while` pour afficher les entiers de 1 à 10 :

```php
i=1;while(i = 1;
while (i=1;while(i ⩽ 10) {
    echo i;i;
    i;i++;
}
```

Il est important de noter que la condition doit être mise à jour à l'intérieur de la boucle pour éviter une boucle infinie. Si la condition n'est jamais mise à jour, la boucle continuera à s'exécuter indéfiniment.

La boucle do...while

La boucle `do...while` permet de répéter un bloc de code au moins une fois, puis de continuer à le répéter tant qu'une condition est vraie. La syntaxe de base de la boucle `do...while` est la suivante :

```php
do {
    // code à exécuter au moins une fois
} while (condition);
```

Voici un exemple d'utilisation de la boucle do...while pour afficher les entiers de 1 à 10 :

```php
i = 1;
do {
    echo $i;
    $i++;
} while (i ⩽ 10);
```

Il est important de noter que la condition est vérifiée à la fin de chaque itération, ce qui signifie que le bloc de code sera exécuté au moins une fois même si la condition est fausse.

La boucle for

La boucle `for` permet de répéter un bloc de code un certain nombre de fois en spécifiant une condition de fin et un compteur. La syntaxe de base de la boucle `for` est la suivante :

```php
for (initialisation; condition; incrémentation) {
    // code à exécuter tant que la condition est vraie
}
```

Voici un exemple d'utilisation de la boucle `for` pour afficher les entiers de `1` à `10` :

```php
for (i=1;i = 1; i=1;i ⩽ 10; $i++) {
    echo $i;
}
```

La boucle `for` est souvent utilisée lorsque le nombre de répétitions est connu à l'avance, car elle permet de spécifier une condition de fin et un compteur de manière claire et concise.

La boucle `for` est similaire à la boucle `while`, mais elle permet de regrouper toutes les informations nécessaires à la boucle (initialisation, condition et incrémentation) en un seul endroit. Elle est donc souvent plus lisible et plus facile à comprendre que la boucle while.

La boucle `for` précédente peut être réécrite de la façon suivante en utilisant une boucle `while` :

```php
i=1;while(i = 1;
while (i=1;while(i ⩽ 10) {
    echo i;i;
    i;i++;
}
```

Il est parfois nécessaire d'inverser l'ordre de parcours d'une boucle, c'est-à-dire de commencer par la fin et de se déplacer vers le début. Cela peut être utile lorsque l'on veut parcourir un tableau ou une chaîne de caractères à l'envers, par exemple.

Pour inverser une boucle, il suffit de changer l'opérateur de comparaison de la condition de fin de la boucle. Par exemple, pour inverser une boucle `for` qui compte de `1` à `10`, il suffit de changer l'opérateur de comparaison de `⩽` à `⩾` :

```php
for (i=10;i = 10; i=10;i ⩾ 1; $i--) {
    echo $i;
}
```

Il est également possible d'inverser une boucle `while` en modifiant la condition de la même manière. Par exemple :

```php
i=10;while(i = 10;
while (i=10;while(i ⩾ 1) {
    echo i;i;
    i;i--;
}
```

Il est important de noter que l'incrémentation ou la décrémentation de la variable de compteur doit être modifiée en conséquence pour qu'elle corresponde à l'ordre de parcours inverse de la boucle.

Il est possible d'imbriquer plusieurs boucles `for`, c'est-à-dire de mettre une boucle `for` à l'intérieur d'une autre boucle `for`. Cette technique est souvent utilisée pour parcourir des tableaux à plusieurs dimensions, comme un tableau à deux dimensions.

Voici un exemple de boucles `for` imbriquées qui parcourent un tableau à deux dimensions :

```php
tableau=[[1,2,3],[4,5,6],[7,8,9]];for(tableau = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
for (tableau=[[1,2,3],[4,5,6],[7,8,9]];for(i = 0; i<count(i < count(i<count(tableau); $i++)
{
    for ($j = 0; $j < count($tableau[$i]); $j++) {
        echo $tableau[$i][$j];
    }
}
```

Dans cet exemple, la boucle `for` externe parcourt chaque ligne du tableau à deux dimensions, tandis que la boucle for interne parcourt chaque colonne de chaque ligne. Le résultat de cette boucle sera l'affichage des entiers de `1` à `9` dans l'ordre.

Les boucles imbriquées sont utiles lorsqu'il est nécessaire de parcourir des tableaux à plusieurs dimensions ou lorsqu'il est nécessaire de répéter un bloc de code plusieurs fois pour chaque valeur d'une variable. Elles peuvent également être utilisées pour créer des motifs ou des structures de données complexes.

Il est important de noter que les boucles imbriquées peuvent être difficiles à comprendre et à déboguer si elles sont mal utilisées.

La boucle foreach

La boucle `foreach` permet de parcourir les éléments d'un tableau ou d'un objet et de les traiter un à un. La syntaxe de base de la boucle `foreach` est la suivante :

```php
foreach (tableau ou objet as $variable) {
    // code à exécuter pour chaque élément du tableau ou de l'objet
}
```

Voici un exemple d'utilisation de la boucle `foreach` pour afficher les éléments d'un tableau :

```php
tableau=[1,2,3,4,5];foreach(tableau = [1, 2, 3, 4, 5];
foreach (tableau=[1,2,3,4,5];foreach(tableau as $element) {
    echo $element;
}
```

Dans cet exemple, la boucle `foreach` parcourt chaque élément du tableau et affiche sa valeur.

Il est également possible de parcourir les éléments d'un tableau associatif en utilisant la boucle `foreach`. Un tableau associatif est un tableau qui a des clés et des valeurs, au lieu d'avoir des indices numériques comme dans un tableau ordinaire.

Voici un exemple de boucle `foreach` qui parcourt les clés et les valeurs d'un tableau associatif :

```php
tableau=["cle1"⇒"valeur1","cle2"⇒"valeur2","cle3"⇒"valeur3"];foreach(tableau = ["cle1" ⇒
"valeur1", "cle2" ⇒ "valeur2", "cle3" ⇒ "valeur3"];
foreach (tableau=["cle1"⇒"valeur1","cle2"⇒"valeur2","cle3"⇒"valeur3"];foreach(tableau as
cle⇒cle ⇒ cle⇒valeur) {
    echo "cle:cle : cle:valeur";
}
```

Dans cet exemple, la boucle `foreach` parcourt chaque élément du tableau associatif et affiche la clé et la valeur de chaque élément.

Break et continue

Break

Le mot-clé `break` permet d'interrompre l'exécution d'une boucle. Il peut être utilisé dans une boucle `for`, une boucle `while`, une boucle `do...while` ou une boucle `foreach`.

Voici un exemple de boucle `for` qui affiche les entiers de `1` à `10` et qui s'arrête lorsque la valeur de la variable de compteur est égale à `5` :

```php
for (i=1;i = 1; i=1;i ⩽ 10; $i++) {
    if ($i == 5) {
        break;
    }
    echo $i;
}

// Affiche 1234
```

Continue

Le mot-clé `continue` permet d'interrompre l'exécution de l'itération courante d'une boucle et de passer à l'itération suivante. Il peut être utilisé dans une boucle `for`, une boucle `while`, une boucle `do...while` ou une boucle `foreach`.

Voici un exemple de boucle `for` qui affiche les entiers de `1` à `10` et qui saute l'itération courante lorsque la valeur de la variable de compteur est égale à `5` :

```php
for (i=1;i = 1; i=1;i ⩽ 10; $i++) {
    if ($i == 5) {
        continue;
    }
    echo $i;
}

// Affiche 1234678910
```

Exercices de lecture

If / elseif / else

1.1. Donnez le résultat de l'exécution du programme suivant :

```php
age=25;if(age = 25;

if (age=25;if(age < 18) {
    echo "Vous êtes mineur.";
} elseif (age ⩾ 18 && age < 30) {
    echo "Vous êtes un adulte jeune.";
} else {
    echo "Vous êtes un adulte.";
}
```

1.2. Donnez le résultat de l'exécution du programme suivant :

```php
x=5;if(x = 5;

if (x=5;if(x > 10) {
    x = 10;
}

if (x < 0) {
    x = 0;
}

if (x % 2 == 0) {
    echo "La valeur de x est paire et égale à " . x;
} else {
    echo "La valeur de x est impaire et égale à " . x;
}
```

1.3. Donnez le résultat de l'exécution du programme suivant :

```php
score=75;if(score = 75;

if (score=75;if(score > 100) {
    score = 100;
}

if (score < 0) {
    score = 0;
}

if (score ⩾ 90) {
    echo "Vous avez obtenu un A!";
} elseif (score⩾80)echo"VousavezobtenuunB!";elseif(score ⩾ 80) {
    echo "Vous avez obtenu un B!";
} elseif (score⩾80)echo"VousavezobtenuunB!";elseif(score ⩾ 70) {
    echo "Vous avez obtenu un C!";
} elseif ($score ⩾ 60) {
    echo "Vous avez obtenu un D!";
} else {
    echo "Vous avez obtenu un E (échec)";
}
```

1.4. Donnez le résultat de l'exécution du programme suivant :

```php
orderTotal=250;orderTotal = 250;
orderTotal=250;couponCode = "DISCOUNT10";

if (couponCode=="FREESHIPPING")echo"Lalivraisonestgratuite!";elseif(couponCode ==
"FREESHIPPING") {
    echo "La livraison est gratuite!";
} elseif (couponCode=="FREESHIPPING")echo"Lalivraisonestgratuite!";elseif(couponCode ==
"DISCOUNT10") {
    orderTotal=orderTotal = orderTotal=orderTotal - (orderTotal*0.1);echo"10orderTotal *
0.1);
    echo "10% de remise appliquée! Montant total de la commande : orderTotal*0.1);echo"10" .
orderTotal;
} else {
    echo "Aucun coupon valide n'a été entré. Montant total de la commande : " . $orderTotal;
}
```

1.5. Donnez le résultat de l'exécution du programme suivant :

```php
a=5;a = 5;
a=5;b = 10;
c=15;if((c = 15;

if ((c=15;if((a + b>b > b>c) && (a+a + a+c > b) && (b + c>c > c>a)) {
    if ((a**2+a ** 2 + a**2+b ** 2 == c**2)||(c ** 2) || (c**2)||(a ** 2 + c**2==c ** 2 ==
c**2==b ** 2) || (b**2+b ** 2 + b**2+c ** 2 == $a ** 2)) {
        echo "Les nombres sont des côtés d'un triangle rectangle.";
    } else {
        echo "Les nombres sont des côtés d'un triangle quelconque.";
    }
} else {
    echo "Les nombres ne peuvent pas être des côtés d'un triangle.";
}
```

1.6. Donnez le résultat de l'exécution du programme suivant :

```php
temp=-10;temp = -10;
temp=-10;unit = "C";
convertedTemp=0;if(convertedTemp = 0;

if (convertedTemp=0;if(unit == "F") {
    convertedTemp=(convertedTemp = (convertedTemp=(temp - 32) * (5/9);
    echo temp."degre´sFahrenheiteste´gala`".round(temp . " degrés Fahrenheit est égal à " .
round(temp."degre´sFahrenheiteste´gala`".round(convertedTemp, 2) . " degrés Celsius.";
} elseif ($unit == "K") {
    $convertedTemp = $temp - 273.15;
    echo $temp . " degrés Kelvin est égal à " . round($convertedTemp, 2) . " degrés
Celsius.";
} else {
    echo $temp . " degrés Celsius est déjà en degrés Celsius.";
}
```

1.7. Donnez le résultat de l'exécution du programme suivant :

```php
text="Thisisanexampleoftext";if(strpos(text = "This is an example of text";

if (strpos(text="Thisisanexampleoftext";if(strpos(text, "example") !== false) {
    text=strreplace("example","EXAMPLE",text = str_replace("example", "EXAMPLE", text=strr
eplace("example","EXAMPLE",text);
}

if (strlen(text) > 20) {
    $text = substr($text, 0, 20) . "...";
}

echo text . PHP_EOL;
```

Switch

2.1. Donnez le résultat de l'exécution du programme suivant :

```php
item="Orange";item = "Orange";
item="Orange";price = 0;

switch (item) {
    case "Apple":
        $price = 0.5;
        break;
    case "Banana":
        $price = 0.25;
        break;
    case "Orange":
        $price = 0.75;
        break;
    case "Mango":
        $price = 1.5;
        break;
    default:
        echo "Invalid item.";
}

quantity = 10;
total=total = total=price * quantity;echo"Item:".quantity;

echo "Item: " . quantity;echo"Item:".item . "<br>";
echo "Price: "." . ".price . "<br>";
echo "Quantity: " . quantity."<br>";echo"Total:quantity . "<br>";
echo "Total: quantity."<br>";echo"Total:" . $total . "<br>";
```

2.2. Donnez le résultat de l'exécution du programme suivant :

```php
input="b";input = "b";
input="b";output = "";

switch (input) {
    case "a":
    case "e":
    case "i":
    case "o":
    case "u":
        $output = "Input is a vowel.";
        break;
    case "1":
    case "2":
    case "3":
    case "4":
    case "5":
    case "6":
    case "7":
    case "8":
    case "9":
    case "0":
        $output = "Input is a number.";
        break;
    default:
        $output = "Input is a consonant.";
}
echo output;
```

2.3. Donnez le résultat de l'exécution du programme suivant :

```php
char='?';char = '?';
char='?';code = ord($char);

switch (true) {
    case ($code > 64 && $code < 91):
        echo "Input is an uppercase letter";
        break;
    case ($code > 96 && $code < 123):
        echo "Input is a lowercase letter";
        break;
    case ($code > 47 && $code < 58):
        echo "Input is a number";
        break;
    case ($code > 32 && $code < 48) || ($code > 57 && $code < 65) || ($code > 90 && $code <
97) || ($code > 122 && $code < 127):
        echo "Input is a special character";
        break;
    default:
        echo "Invalid input";
}
```

Match

3.1. Donnez le résultat de l'exécution du programme suivant :

```php
vehicle="motorcycle";vehicle = "motorcycle";

vehicle="motorcycle";result = match
(vehicle)"car"⇒"Thisisacar","truck"⇒"Thisisatruck","motorcycle"⇒"Thisisamotorcycle","bicy
cle"⇒"Thisisabicycle",default⇒"Thisisnotacar,truck,motorcycleorbicycle";echovehicle) {
    "car" ⇒ "This is a car",
    "truck" ⇒ "This is a truck",
    "motorcycle" ⇒ "This is a motorcycle",
    "bicycle" ⇒ "This is a bicycle",
    default ⇒ "This is not a car, truck, motorcycle or bicycle"
};

echo
vehicle)"car"⇒"Thisisacar","truck"⇒"Thisisatruck","motorcycle"⇒"Thisisamotorcycle","bicyc
le"⇒"Thisisabicycle",default⇒"Thisisnotacar,truck,motorcycleorbicycle";echoresult;
```

3.2. Donnez le résultat de l'exécution du programme suivant :

```php
student=['name'⇒'JohnDoe','age'⇒22,'gender'⇒'male','grade'⇒'A','absences'⇒2,];student =
[
    'name' ⇒ 'John Doe',
    'age' ⇒ 22,
    'gender' ⇒ 'male',
    'grade' ⇒ 'A',
    'absences' ⇒ 2,
];

student=['name'⇒'JohnDoe','age'⇒22,'gender'⇒'male','grade'⇒'A','absences'⇒2,];result =
match (true) {
    student['age']<18⇒'Studentisunder18yearsold',student['age'] < 18 ⇒ 'Student is under
18 years old',
    student['age']<18⇒'Studentisunder18yearsold',student['gender'] == 'male' &&
student['grade']=='A'⇒'MalestudentwithgradeA',student['grade'] == 'A' ⇒ 'Male student with
grade A',
    student['grade']=='A'⇒'MalestudentwithgradeA',student['gender'] == 'female' &&
student['absences']==0⇒'Femalestudentwithnoabsences',student['absences'] == 0 ⇒ 'Female
student with no absences',
    student['absences']==0⇒'Femalestudentwithnoabsences',student['age'] ⩾ 18 &&
student['grade'] == 'F' && student['absences'] > 5 ⇒ 'Adult student with grade F and more
than 5 absences',
    default ⇒ 'Student does not fit in any specific category'
};

echo $result;
```

While

4.1.1. Donnez le résultat de l'exécution du programme suivant :

```php
i=1;while(i = 1;
while (i=1;while(i ⩽ 5) {
    echo i.PHPEOL;i . PHP_EOL;
    i.PHPEOL;i++;
}
```

4.2. Donnez le résultat de l'exécution du programme suivant :

```php
numbers=[3,9,15,5,8,14];numbers = [3, 9, 15, 5, 8, 14];
numbers=[3,9,15,5,8,14];i = 0;
while (i<count(i < count(i<count(numbers)) {
    if (numbers[numbers[numbers[i] % 2 == 0) {
        echo numbers[numbers[numbers[i] . " is even" . PHP_EOL;
    } else {
        echo numbers[numbers[numbers[i] . " is odd" . PHP_EOL;
    }
    $i++;
}
```

4.3. Donnez le résultat de l'exécution du programme suivant :

```php
list=[1,2,3,4,5];list = [1, 2, 3, 4, 5];
list=[1,2,3,4,5];i = 0;
sum=0;//(max/2)*(max+1)sum = 0; // (max/2)*(max+1)
sum=0;//(max/2)*(max+1)max = 0;
min=PHPINTMAX;while(min = PHP_INT_MAX;

while(min=PHPINTMAX;while(i < count(list)) {
    if($list[$i] > $max) {
        $max = $list[$i];
    }

    if($list[$i] < $min) {
        $min = $list[$i];
    }

    $sum += $list[$i];
    $i++;
}

average = sum/count(sum / count(sum/count(list);
echo "The sum of numbers is: " . sum.PHPEOL;echo"Theminimumnumberis:".sum . PHP_EOL;
echo "The minimum number is: " . sum.PHPEOL;echo"Theminimumnumberis:".min . PHP_EOL;
echo "The maximum number is: " . max.PHPEOL;echo"Theaverageofnumbersis:".max . PHP_EOL;
echo "The average of numbers is: " . max.PHPEOL;echo"Theaverageofnumbersis:".average .
PHP_EOL;
```

4.4. Donnez le résultat de l'exécution du programme suivant :

```php
input=[['name'⇒'John','age'⇒25,'gender'⇒'male'],['name'⇒'Jane',
'age'⇒20,'gender'⇒'female'],['name'⇒'Mike','age'⇒27,'gender'⇒'male'],['name'⇒'Emily',
'age'⇒31,'gender'⇒'female'],//...];input = [
    ['name' ⇒ 'John', 'age' ⇒ 25, 'gender' ⇒ 'male'],
    ['name' ⇒ 'Jane', 'age' ⇒ 20, 'gender' ⇒ 'female'],
    ['name' ⇒ 'Mike', 'age' ⇒ 27, 'gender' ⇒ 'male'],
    ['name' ⇒ 'Emily', 'age' ⇒ 31, 'gender' ⇒ 'female'],
    //...
];
input=[['name'⇒'John','age'⇒25,'gender'⇒'male'],['name'⇒'Jane',
'age'⇒20,'gender'⇒'female'],['name'⇒'Mike','age'⇒27,'gender'⇒'male'],['name'⇒'Emily',
'age'⇒31,'gender'⇒'female'],//...];i = 0;
totalAge=0;totalAge = 0;
totalAge=0;totalMales = 0;
totalFemales=0;while(totalFemales = 0;

while (totalFemales=0;while(i < count(input)) {
    $totalAge += $input[$i]['age'];
    if($input[$i]['gender'] ⩵ 'male') {
        $totalMales++;
    } else {
        $totalFemales++;
    }
    $i++;
}
averageAge = totalAge/count(totalAge / count(totalAge/count(input);
echo "Total number of males: " . totalMales.PHPEOL;echo"Totalnumberoffemales:".totalMales .
PHP_EOL;
echo "Total number of females: " . totalMales.PHPE
OL;echo"Totalnumberoffemales:".totalFemales . PHP_EOL;
echo "Average age of all individuals: " . $averageAge . PHP_EOL;
```

Do While

5.1. Donnez le résultat de l'exécution du programme suivant :

```php
password = "";
do {
    $password = readline("Entrez votre mot de passe (au moins 8 caractères, une majuscule,
une minuscule et un chiffre): ");
} while (!preg_match("/^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)[A-Za-z\d]{8,}/", $password));

echo "Mot de passe valide!";
```

For

6.1. Donnez le résultat de l'exécution du programme suivant :

```php
for (i=1;i = 1; i=1;i < 10; $i++) {
    echo $i . PHP_EOL;
}
```

6.2. Donnez le résultat de l'exécution du programme suivant :

```php
text="Thisisaexampleoftext";for(text = "This is a example of text";

for (text="Thisisaexampleoftext";for(i = 0; i<strlen(i < strlen(i<strlen(text); $i++) {
    echo $text[$i] . '-';
}
```

6.3. Donnez le résultat de l'exécution du programme suivant :

```php
text="Thisisaexampleoftext";text = "This is a example of text";
text="Thisisaexampleoftext";vowels = ['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'];
newText='';for(newText = '';

for (newText='';for(i = 0; i<strlen(i < strlen(i<strlen(text); i++) {
    if (!in_array($text[$i], $vowels)) {
        $newText .= $text[$i];
    }
}
echo newText;
```

6.4. Donnez le résultat de l'exécution du programme suivant :

```php
numbers=[1,2,3,4,5,6,7,8,9,10];for(numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

for (numbers=[1,2,3,4,5,6,7,8,9,10];for(i = count(numbers)-1;numbers) - 1; numbers)-1;i >=
0; $i--) {
    if ($numbers[$i] % 2 == 0) {
        echo $numbers[$i] . ' is even' . PHP_EOL;
    } else {
        echo $numbers[$i] . ' is odd' . PHP_EOL;
    }
}
```

6.5. Donnez le résultat de l'exécution du programme suivant :

```php
text="ThiSISAnexAMPleOfTeXT";text = "ThiS IS An exAMPle Of TeXT";
text="ThiSISAnexAMPleOfTeXT";search = "text";
newText="";if(pregmatch('/[A-Z]/',newText = "";

if(preg_match('/[A-Z]/', newText="";if(pregmatch('/[A-Z]/',text)){
    text=strtolower(text = strtolower(text=strtolower(text);
}

if (strpos(text,text, text,search) !== false) {
    text=strreplace(text = str_replace(text=strreplace(search, strtoupper(search),search),
search),text);
}

for (i=0;i = 0; i=0;i < strlen(text);text); text);i++) {
    if (i % 2 == 0) {
        $newText .= strtoupper($text[$i]);
    } else {
        $newText .= $text[$i];
    }
}

echo newText . PHP_EOL;
```

6.6. Donnez le résultat de l'exécution du programme suivant :

```php
numbers=[5,3,2,1];for(numbers = [5, 3, 2, 1];

for (numbers=[5,3,2,1];for(i = 0; i<count(i < count(i<count(numbers); $i++) {
    for ($j = $i+1; $j < count($numbers); $j++) {
        echo $numbers[$i] . ' * ' . $numbers[$j] . ' = ' . $numbers[$i] * $numbers[$j] .
PHP_EOL;
    }
}
```

Foreach

7.1. Donnez le résultat de l'exécution du programme suivant :

```php
fruits=['apple','banana','orange','strawberry'];foreach(fruits = ['apple', 'banana',
'orange', 'strawberry'];
foreach (fruits=['apple','banana','orange','strawberry'];foreach(fruits as $fruit) {
    echo $fruit . PHP_EOL;
}
```

7.2. Donnez le résultat de l'exécution du programme suivant :

```php
employees=[['name'⇒'JohnDoe','salary'⇒70000,'position'⇒'Manager'],['name'⇒'JaneSmith',
'salary'⇒60000,'position'⇒'Developer'],['name'⇒'BobRoss',
'salary'⇒55000,'position'⇒'Designer'],['name'⇒'CharlieBrown',
'salary'⇒65000,'position'⇒'Developer'],];employees = [
    ['name' ⇒ 'John Doe', 'salary' ⇒ 70000, 'position' ⇒ 'Manager'],
    ['name' ⇒ 'Jane Smith', 'salary' ⇒ 60000, 'position' ⇒ 'Developer'],
    ['name' ⇒ 'Bob Ross', 'salary' ⇒ 55000, 'position' ⇒ 'Designer'],
    ['name' ⇒ 'Charlie Brown', 'salary' ⇒ 65000, 'position' ⇒ 'Developer'],
];
employees=[['name'⇒'JohnDoe','salary'⇒70000,'position'⇒'Manager'],['name'⇒'JaneSmith',
'salary'⇒60000,'position'⇒'Developer'],['name'⇒'BobRoss',
'salary'⇒55000,'position'⇒'Designer'],['name'⇒'CharlieBrown',
'salary'⇒65000,'position'⇒'Developer'],];totalSalary = 0;
foreach (employeesasemployees as employeesasemployee) {
    if(employee['position'] ⩵ "Developer"){
        echo $employee['name'] . " - " . $employee['salary'] . PHP_EOL;
        $totalSalary += $employee['salary'];
    }
}
echo "Total salaire des développeurs: " . totalSalary;
```

7.3. Donnez le résultat de l'exécution du programme suivant :

```php
data=[[’firstname’⇒’John’,’lastname’⇒’Doe’,’age’⇒25],[’firstname’⇒’Jane’,
’lastname’⇒’Smith’,’age’⇒30],[’firstname’⇒’Bob’,’lastname’⇒’Ross’,’age’⇒35],
[’firstname’⇒’Charlie’,’lastname’⇒’Brown’,’age’⇒40],[’firstname’⇒’Maggie’,
’lastname’⇒’Simpson’,’age’⇒3],];data = [
    ['first_name' ⇒ 'John', 'last_name' ⇒ 'Doe', 'age' ⇒ 25],
    ['first_name' ⇒ 'Jane', 'last_name' ⇒ 'Smith', 'age' ⇒ 30],
    ['first_name' ⇒ 'Bob', 'last_name' ⇒ 'Ross', 'age' ⇒ 35],
    ['first_name' ⇒ 'Charlie', 'last_name' ⇒ 'Brown', 'age' ⇒ 40],
    ['first_name' ⇒ 'Maggie', 'last_name' ⇒ 'Simpson', 'age' ⇒ 3],
];
data=[[’firstname’⇒’John’,’lastname’⇒’Doe’,’age’⇒25],[’firstname’⇒’Jane’,’lastn
ame’⇒’Smith’,’age’⇒30],[’firstname’⇒’Bob’,’lastname’⇒’Ross’,’age’⇒35],[’firstn
ame’⇒’Charlie’,’lastname’⇒’Brown’,’age’⇒40],[’firstname’⇒’Maggie’,’lastname’⇒’Simpson’,
’age’⇒3],];oldest_person = "";
youngestperson="";youngest_person = "";
youngestperson="";oldest_age = 0;
youngestage=PHPINTMAX;foreach(youngest_age = PHP_INT_MAX;
foreach (youngestage=PHPINTMAX;foreach(data as item) {
    $fullname = $item['first_name'] . ' ' . $item['last_name'];
    if($item['age'] > $oldest_age) {
        $oldest_age = $item['age'];
        $oldest_person = $fullname;
    }
    if($item['age'] < $youngest_age) {
        $youngest_age = $item['age'];
        $youngest_person = $fullname;
    }
}
echo "The oldest person is oldest_person with age
oldestage\n";echo"Theyoungestpersonisoldest_age \n";
echo "The youngest person is oldestage\n";echo"Theyoungestpersonisyoungest_person with age
$youngest_age";
```

7.4. Donnez le résultat de l’exécution du programme suivant :

```php
data=[['firstname'⇒'John','lastname'⇒'Doe','age'⇒25,'gender'⇒'male'],
['firstname'⇒'Jane','lastname'⇒'Smith','age'⇒30,'gender'⇒'female'],['firstname'⇒'Bob',
'lastname'⇒'Ross','age'⇒35,'gender'⇒'male'],['firstname'⇒'Charlie','lastname'⇒'Brown',
'age'⇒40,'gender'⇒'male'],['firstname'⇒'Maggie','lastname'⇒'Simpson',
'age'⇒20,'gender'⇒'female'],];data = [
    ['first_name' ⇒ 'John', 'last_name' ⇒ 'Doe', 'age' ⇒ 25, 'gender' ⇒ 'male'],
    ['first_name' ⇒ 'Jane', 'last_name' ⇒ 'Smith', 'age' ⇒ 30, 'gender' ⇒ 'female'],
    ['first_name' ⇒ 'Bob', 'last_name' ⇒ 'Ross', 'age' ⇒ 35, 'gender' ⇒ 'male'],
    ['first_name' ⇒ 'Charlie', 'last_name' ⇒ 'Brown', 'age' ⇒ 40, 'gender' ⇒ 'male'],
    ['first_name' ⇒ 'Maggie', 'last_name' ⇒ 'Simpson', 'age' ⇒ 20, 'gender' ⇒ 'female'],
];
data=[['firstname'⇒'John','lastname'⇒'Doe','age'⇒25,'gender'⇒'male'],['firstn
ame'⇒'Jane','lastname'⇒'Smith','age'⇒30,'gender'⇒'female'],['firstname'⇒'Bob','lastn
ame'⇒'Ross','age'⇒35,'gender'⇒'male'],['firstname'⇒'Charlie','lastname'⇒'Brown',
'age'⇒40,'gender'⇒'male'],['firstname'⇒'Maggie','lastname'⇒'Simpson',
'age'⇒20,'gender'⇒'female'],];male_counter = 0;
femalecounter=0;female_counter = 0;
femalecounter=0;male_age_sum = 0;
femaleagesum=0;foreach(female_age_sum = 0;
foreach (femaleagesum=0;foreach(data as item) {
    if($item['gender'] == 'male') {
        $male_counter++;
        $male_age_sum += $item['age'];
    }
    elseif ($item['gender'] == 'female') {
        $female_counter++;
        $female_age_sum += $item['age'];
    }
}
male_age_avg = maleagesum/male_age_sum / maleagesum/male_counter;
femaleageavg=female_age_avg = femaleageavg=female_age_sum /
femalecounter;echo"Theaverageageofmalesisfemale_counter;
echo "The average age of males is femalecounter;echo"Theaverageageofmalesismale_age_avg \n";
echo "The average age of females is $female_age_avg";
```

7.5. Donnez le résultat de l'exécution du programme suivant :

```php
employees=[['firstname'⇒'John','lastname'⇒'Doe','projects'⇒['project1','project2',
'project3']],['firstname'⇒'Jane','lastname'⇒'Smith','projects'⇒['project2','project4',
'project5']],['firstname'⇒'Bob','lastname'⇒'Ross','projects'⇒['project1','project3']],
['firstname'⇒'Charlie','lastname'⇒'Brown','projects'⇒['project4','project6']],
['firstname'⇒'Maggie','lastname'⇒'Simpson','projects'⇒['project2']],];employees = [
    ['first_name' ⇒ 'John', 'last_name' ⇒ 'Doe', 'projects' ⇒ ['project1', 'project2',
'project3']],
    ['first_name' ⇒ 'Jane', 'last_name' ⇒ 'Smith', 'projects' ⇒ ['project2', 'project4',
'project5']],
    ['first_name' ⇒ 'Bob', 'last_name' ⇒ 'Ross', 'projects' ⇒ ['project1', 'project3']],
    ['first_name' ⇒ 'Charlie', 'last_name' ⇒ 'Brown', 'projects' ⇒ ['project4',
'project6']],
    ['first_name' ⇒ 'Maggie', 'last_name' ⇒ 'Simpson', 'projects' ⇒ ['project2']],
];
employees=[['firstname'⇒'John','lastname'⇒'Doe','projects'⇒['project1','project2',
'project3']],['firstname'⇒'Jane','lastname'⇒'Smith','projects'⇒['project2','project4',
'project5']],['firstname'⇒'Bob','lastname'⇒'Ross','projects'⇒['project1','project3']],
['firstname'⇒'Charlie','lastname'⇒'Brown','projects'⇒['project4','project6']],['firstn
ame'⇒'Maggie','lastname'⇒'Simpson','projects'⇒['project2']],];project1 = [];
project2=[];project2 = [];
project2=[];project3 = [];
project4=[];project4 = [];
project4=[];project5 = [];
project6=[];foreach(project6 = [];
foreach (project6=[];foreach(employees as employee) {
    foreach ($employee['projects'] as $project) {
        if($project == 'project1') {
            $project1[] = $employee['first_name'] . ' ' . $employee['last_name'];
        } elseif($project == 'project2') {
            $project2[] = $employee['first_name'] . ' ' . $employee['last_name'];
        } elseif($project == 'project3') {
            $project3[] = $employee['first_name'] . ' ' . $employee['last_name'];
        } elseif($project == 'project4') {
            $project4[] = $employee['first_name'] . ' ' . $employee['last_name'];
        } elseif($project == 'project5') {
            $project5[] = $employee['first_name'] . ' ' . $employee['last_name'];
        } elseif($project == 'project6') {
            $project6[] = $employee['first_name'] . ' ' . $employee['last_name'];
        }
    }
}
echo "Employees working on project1: " . implode(", ",project1) . "\n";
echo "Employees working on project2: " . implode(",
",project2)."\n";echo"Employeesworkingonproject3:".implode(",",project2) . "\n";
echo "Employees working on project3: " . implode(",
",project2)."\n";echo"Employeesworkingonproject3:".implode(",",project3) . "\n";
echo "Employees working on project4: " . implode(",
",project4)."\n";echo"Employeesworkingonproject5:".implode(",",project4) . "\n";
echo "Employees working on project5: " . implode(",
",project4)."\n";echo"Employeesworkingonproject5:".implode(",",project5) . "\n";
echo "Employees working on project6:
```

Exercices

If/elseif/else

Le système de notation

Dans cet exercice, vous allez créer un programme PHP qui fonctionne en ligne de commande. Le programme lira une note entrée par l'utilisateur et renverra une appréciation en fonction de cette note.

Voici les étapes de l'exercice :

1. Commencez par lire une note entrée par l'utilisateur. Vous pouvez utiliser la fonction `fgets(STDIN)` pour cela.
2. Ensuite, vérifiez si la note est valide. Une note valide est un nombre entre 0 et 20. Si la note n'est pas valide, affichez un message d'erreur et arrêtez le programme.
3. Si la note est valide, utilisez des instructions `if` pour déterminer l'appréciation à afficher. Voici les appréciations correspondant à chaque intervalle de notes :
   - 0 à 10 : "Malheureusement, vous avez échoué."
   - 10 à 12 : "Vous avez juste réussi!"
   - 12 à 14 : "C'est un assez bien!"
   - 14 à 16 : "C'est un bien!"
   - 16 à 20 : "C'est un très bien!"

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
echo "Veuillez entrer une note : ";
$note = fgets(STDIN);

// Vérifiez si la note est valide ici

// Utilisez des instructions if pour déterminer l'appréciation à afficher
```

Le calculateur de remise avancé

Dans cet exercice, vous allez créer un programme PHP qui fonctionne en ligne de commande. Le programme lira le prix d'un article et le statut de l'utilisateur (étudiant ou non) et calculera le prix final de l'article après remise.

Voici les étapes de l'exercice :

1. Commencez par lire le prix de l'article et le statut de l'utilisateur. Vous pouvez utiliser la fonction `fgets(STDIN)` pour cela.
2. Ensuite, utilisez des instructions `if` et `elseif` pour déterminer le taux de remise. Voici les remises correspondant à chaque intervalle de prix :

- Plus de 100€ : 20% de remise
- Entre 50€ et 100€ : 10% de remise
- Moins de 50€ : pas de remise

3. Si l'utilisateur est un étudiant, ajoutez une remise supplémentaire de 5%.
4. Calculez le prix final de l'article après remise et affichez-le.

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
echo "Veuillez entrer le prix de l'article : ";
prix=fgets(STDIN);echo"L’utilisateurest-ilune´tudiant?(oui/non):";prix = fgets(STDIN);

echo "L'utilisateur est-il un étudiant ? (oui/non) : ";
prix=fgets(STDIN);echo"L’utilisateurest-ilune´tudiant?(oui/non):";estEtudiant =
trim(fgets(STDIN)) == "oui";

// Déterminez le taux de remise ici

// Calculez le prix final ici
```

Le système de réservation de billets de cinéma

Dans cet exercice, vous créerez un programme PHP qui fonctionne en ligne de commande. Le programme lira le jour de la semaine, l'heure de la journée et l'âge du client, puis calculera le prix du billet de cinéma.

Voici les étapes de l'exercice :

1. Commencez par lire le jour de la semaine, l'heure de la journée et l'âge du client. Vous pouvez utiliser la fonction `fgets(STDIN)` pour cela.
2. Ensuite, utilisez des instructions `if` pour déterminer le prix de base du billet. Le prix de base est de 15€ les samedis et dimanches, et de 10€ les autres jours.
3. Si l'heure est 20h ou plus, ajoutez un supplément de 2€ au prix du billet.
4. Si le client est un enfant (12 ans ou moins) ou un senior (65 ans ou plus), appliquez une remise de 50% au prix du billet.
5. Affichez le prix final du billet.

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
echo "Veuillez entrer le jour de la semaine : ";
jour=trim(fgets(STDIN));echo"Veuillezentrerl'heuredelajourne´e:";jour = trim(fgets(STDIN));

echo "Veuillez entrer l'heure de la journée : ";
jour=trim(fgets(STDIN));echo"Veuillezentrerl'heuredelajourne´e:";heure = fgets(STDIN);

echo "Veuillez entrer l'âge du client : ";
$age = fgets(STDIN);

// Déterminez le prix de base du billet ici

// Appliquez le supplément pour les heures tardives ici

// Appliquez la remise pour les enfants et les seniors ici

// Affichez le prix final du billet ici
```

Switch

Convertisseur de jours de la semaine

Dans cet exercice, vous créerez un programme qui convertit un nombre de 1 à 7 en jour de la semaine correspondant. Vous utiliserez l'instruction `switch` pour effectuer la conversion.

Voici les étapes de l'exercice :

1. Demandez à l'utilisateur d'entrer un nombre de 1 à 7 pour représenter un jour de la semaine.
2. Utilisez l'instruction `switch` pour convertir le nombre en jour de la semaine correspondant. Voici les correspondances :
   - 1 : "Lundi"
   - 2 : "Mardi"
   - 3 : "Mercredi"
   - 4 : "Jeudi"
   - 5 : "Vendredi"
   - 6 : "Samedi"
   - 7 : "Dimanche"
3. Si l'utilisateur entre un nombre en dehors de la plage de 1 à 7, affichez un message d'erreur.

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
echo "Veuillez entrer un nombre de 1 à 7 : ";
$numeroJour = fgets(STDIN);

// Utilisez l'instruction switch pour convertir le nombre en jour de la semaine
```

Une fois que vous aurez terminé l'exercice, écrivez le en utilisant l'expression `match`.

Match

Gestion des catégories d'âge

Dans cet exercice, vous allez créer un programme qui prend en entrée l'âge d'une personne et détermine sa catégorie d'âge en utilisant la structure `match`. Vous explorerez comment `match` peut être une alternative plus concise et lisible par rapport à l'utilisation d'un `if` ou d'un `switch`.

Voici les étapes de l'exercice :

1. Demandez à l'utilisateur d'entrer son âge.
2. Utilisez la structure `match` pour déterminer la catégorie d'âge en fonction de l'âge de la personne. Voici les catégories d'âge et leurs plages correspondantes :
   - Moins de 18 ans : "Mineur"
   - De 18 à 64 ans : "Adulte"
   - 65 ans et plus : "Senior"
3. Affichez la catégorie d'âge correspondante.

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
echo "Veuillez entrer votre âge : ";
$age = fgets(STDIN);

// Utilisez la structure match pour déterminer la catégorie d'âge
```

While

Calcul de la somme des entiers jusqu'à un nombre donné

Dans cet exercice, vous allez créer un programme qui calcule la somme des entiers de 1 jusqu'à un nombre donné par l'utilisateur en utilisant une boucle `while`.

Voici les étapes de l'exercice :

1. Demandez à l'utilisateur d'entrer un nombre entier positif.
2. Utilisez une boucle `while` pour calculer la somme des entiers de 1 jusqu'au nombre donné.
3. Affichez la somme calculée.

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
echo "Veuillez entrer un nombre entier positif : ";
$nombre = intval(fgets(STDIN));

// Vérifiez si le nombre est positif ici

// Initialisez la somme et le compteur

// Utilisez une boucle while pour calculer la somme

// Affichez la somme calculée
```

Une fois l'exercice terminé, regardez comment le résoudre sans boucle.

Lancez la version avec une boucle en donnant des nombres de plus en plus grands jusqu'à ce que l'exécution prenne plus de 10 secondes.

Affichage des puissances de 2 jusqu'à un certain exposant

Dans cet exercice, vous allez créer un programme qui affiche les puissances de 2 jusqu'à un certain exposant donné par l'utilisateur en utilisant une boucle `while`.

Voici les étapes de l'exercice :

1. Demandez à l'utilisateur d'entrer un exposant entier positif.
2. Utilisez une boucle `while` pour afficher les puissances de 2, en commençant par 2^0 jusqu'à 2^exposant.
3. Affichez chaque puissance de 2 à chaque itération de la boucle.

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
echo "Veuillez entrer un exposant entier positif : ";
$exposant = intval(fgets(STDIN));

// Vérifiez si l'exposant est positif ici

// Initialisez le compteur et la puissance

// Utilisez une boucle while pour afficher les puissances de 2

// Affichez chaque puissance de 2
```

Do while

Validation d'un mot de passe

Dans cet exercice, vous allez créer un programme qui demande à l'utilisateur de saisir un mot de passe et valide si ce mot de passe répond à certaines conditions prédéfinies. Vous utiliserez une boucle `do while` pour permettre à l'utilisateur de réessayer tant que le mot de passe n'est pas valide.

Voici les étapes de l'exercice :

1. Définissez les conditions pour un mot de passe valide. Par exemple, le mot de passe doit avoir une longueur minimale de 8 caractères et contenir au moins une lettre majuscule, une lettre minuscule et un chiffre.
2. Demandez à l'utilisateur de saisir un mot de passe.
3. Utilisez une boucle `do while` pour valider le mot de passe. La boucle doit se répéter tant que le mot de passe ne répond pas aux conditions définies.
4. Affichez un message indiquant si le mot de passe est valide ou non.

For

Calcul de la factorielle

Dans cet exercice, vous allez créer un programme qui calcule la factorielle d'un nombre donné par l'utilisateur en utilisant une boucle `for`.

La factorielle d'un nombre entier positif n, notée n!, est le produit de tous les entiers positifs de 1 à n. La formule pour calculer la factorielle est la suivante :

`n! = n (n-1) (n-2) ... 2 * 1`

Voici les étapes de l'exercice :

1. Demandez à l'utilisateur d'entrer un nombre entier positif.
2. Utilisez une boucle `for` pour calculer la factorielle du nombre donné.
3. Affichez le résultat de la factorielle.

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
echo "Veuillez entrer un nombre entier positif : ";
$nombre = intval(fgets(STDIN));

// Vérifiez si le nombre est positif ici

// Initialisez la variable pour stocker le résultat de la factorielle

// Utilisez une boucle for pour calculer la factorielle

// Affichez le résultat de la factorielle
```

Inversion d'une chaîne de caractères

Dans cet exercice, vous allez créer un programme qui inverse une chaîne de caractères donnée par l'utilisateur en utilisant une boucle `for`.

Voici les étapes de l'exercice :

1. Demandez à l'utilisateur d'entrer une chaîne de caractères.
2. Utilisez une boucle `for` pour inverser la chaîne de caractères.
3. Affichez la chaîne de caractères inversée.

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
echo "Veuillez entrer une chaîne de caractères : ";
$chaine = trim(fgets(STDIN));

// Initialisez la variable pour stocker la chaîne de caractères inversée

// Utilisez une boucle for pour inverser la chaîne de caractères

// Affichez la chaîne de caractères inversée
```

Recherche dans un carnet d'adresses

Dans cet exercice, vous allez créer un programme qui permet de rechercher une personne dans un petit carnet d'adresses contenant le nom et l'adresse e-mail des personnes. Vous utiliserez une boucle `for` pour parcourir le carnet d'adresses et effectuer la recherche.

Voici les étapes de l'exercice :

1. Déclarez un tableau associatif contenant les noms et les adresses e-mail des personnes.

```php
$carnetAdresses = [
    "Alice" => "alice@example.com",
    "Bob" => "bob@example.com",
    "Charlie" => "charlie@example.com",
    "David" => "david@example.com"
];
```

1. Demandez à l'utilisateur d'entrer le nom d'une personne à rechercher.
2. Utilisez une boucle `for` pour parcourir le carnet d'adresses et rechercher la personne.
3. Affichez l'adresse e-mail de la personne si elle est trouvée, ou un message indiquant que la personne n'a pas été trouvée.

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
carnetAdresses=
["Alice"⇒"alice@example.com","Bob"⇒"bob@example.com","Charlie"⇒"charlie@example.com","Dav
id"⇒"david@example.com"];echo"Veuillezentrerlenomd'unepersonnea`rechercher:";carnetAdresses
= [
    "Alice" ⇒ "alice@example.com",
    "Bob" ⇒ "bob@example.com",
    "Charlie" ⇒ "charlie@example.com",
    "David" ⇒ "david@example.com"
];

echo "Veuillez entrer le nom d'une personne à rechercher : ";
carnetAdresses=
["Alice"⇒"alice@example.com","Bob"⇒"bob@example.com","Charlie"⇒"charlie@example.com","Dav
id"⇒"david@example.com"];echo"Veuillezentrerlenomd'unepersonnea`rechercher:";nomRecherche =
trim(fgets(STDIN));

// Utilisez une boucle for pour rechercher la personne dans le carnet d'adresses

// Affichez l'adresse e-mail de la personne si elle est trouvée
```

Partie 2: Distance de Levenshtein

Modifiez l'exercice précédent en utilisant la distance de Levenshtein. Cela va permettre de rendre la recherche résistante aux fautes de frappe.

La distance de Levenshtein est une mesure de la différence entre deux chaînes de caractères. Elle représente le nombre minimum d'opérations (insertion, suppression ou substitution d'un caractère) nécessaires pour transformer une chaîne en une autre.

En PHP, vous pouvez utiliser la fonction `levenshtein()` pour calculer la distance de Levenshtein entre deux chaînes. Voici un exemple d'utilisation :

```php
chaine1="chat";chaine1 = "chat";
chaine1="chat";chaine2 = "chien";

distance=levenshtein(distance = levenshtein(distance=levenshtein(chaine1,
chaine2);echo"LadistancedeLevenshteinentre'chaine2);

echo "La distance de Levenshtein entre 'chaine2);echo"LadistancedeLevenshteinentre'chaine1'
et 'chaine2'estdechaine2' est de chaine2'estdedistance";
```

Dans cet exemple, la fonction `levenshtein()` calcule la distance de Levenshtein entre les chaînes "chat" et "chien". Le résultat affiché sera "La distance de Levenshtein entre 'chat' et 'chien' est de 2", car il faut deux opérations (une substitution et une insertion) pour transformer "chat" en "chien".

Foreach

Recherche du plus grand élément dans un tableau

Dans cet exercice, vous allez créer un programme qui recherche le plus grand élément dans un tableau donné en utilisant la boucle `foreach`.

Voici les étapes de l'exercice :

1. Déclarez un tableau contenant plusieurs nombres.
2. Initialisez une variable pour stocker le plus grand élément et assignez-lui une valeur initiale.
3. Utilisez la boucle `foreach` pour parcourir le tableau et mettre à jour la variable du plus grand élément si un élément plus grand est trouvé.
4. Affichez le plus grand élément trouvé.

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
$nombres = [15, 7, 22, 13, 9];

// Initialisez la variable pour stocker le plus grand élément

// Utilisez la boucle foreach pour rechercher le plus grand élément

// Affichez le plus grand élément
```

Calcul de la moyenne des notes

Dans cet exercice, vous allez créer un programme qui calcule la moyenne des notes d'un groupe d'étudiants en utilisant la boucle `foreach`.

Voici les étapes de l'exercice :

Voici un exemple de structure de code que vous pouvez utiliser pour commencer :

```php
$notes = [
    "Alice" => 15.5,
    "Bob" => 12.8,
    "Charlie" => 18.2,
    "David" => 14.9
];

// Initialisez la variable pour stocker la somme des notes

// Initialisez la variable pour compter le nombre d'étudiants

// Utilisez la boucle foreach pour calculer la somme des notes et compter le nombre d'étudiants

// Calculez la moyenne en divisant la somme des notes par le nombre d'étudiants

// Affichez la moyenne calculée
```

1.6 Les fonctions de PHP

Les fonctions de base disponibles avec PHP sont des outils pratiques qui permettent de réaliser des tâches courantes de manière simple et efficace.

À quoi peuvent-elles servir ?

- Traiter des chaînes de caractères : par exemple, la fonction "strtolower" permet de convertir une chaîne de caractères en minuscules, tandis que la fonction "ucfirst" met en majuscule le premier caractère d'une chaîne.

```php
echo strtolower("HELLO WORLD"); // hello world
echo ucfirst("hello world"); // Hello world
```

- Traiter des nombres : par exemple, la fonction "round" permet d'arrondir un nombre, tandis que la fonction "rand" retourne un nombre aléatoire.

```php
echo round(3.14159); // 3
echo rand(1, 10); // 5
```

- Manipuler des tableaux : par exemple, la fonction "array_sum" permet de calculer la somme des éléments d'un tableau, tandis que la fonction "array_diff" retourne les éléments présents dans un tableau qui ne sont pas présents dans un autre tableau.

```php
echo array_sum([1, 2, 3, 4, 5]); // 15
print_r(array_diff([1, 2, 3], [2, 3, 4])); // Array ( [0] => 1 )
```

- Traiter des dates et des heures : par exemple, la fonction "date" permet de récupérer la date et l'heure courantes au format choisi, tandis que la fonction "strtotime" permet de convertir une chaîne de caractères représentant une date en un timestamp (nombre de secondes écoulées depuis le 1er janvier 1970).

```php
echo date("d/m/Y"); // 01/01/2020 mais à la date du jour
echo strtotime("2020-01-01"); // 1577836800
```

Voici quelques exemples de fonctions plus avancées disponibles :

- Fonctions de gestion de fichiers : par exemple, la fonction "file_get_contents" permet de lire le contenu d'un fichier, tandis que la fonction "file_put_contents" permet d'écrire du contenu dans un fichier.

```php
echo file_get_contents("hello.txt"); // Hello World
file_put_contents("hello.txt", "Hello World");
```

- Fonctions de gestion de bases de données : par exemple, la fonction "mysqli_query" permet d'exécuter une requête SQL sur une base de données MySQL, tandis que la fonction "PDO::query" permet de faire de même sur une base de données utilisant le driver PDO.

```php
mysqli=mysqliconnect("localhost","username","password","database");mysqli = mysqli_connect(
    "localhost",
    "username",
    "password",
    "database"
);
mysqli=mysqliconnect("localhost","username","password","database");mysqli→query("SELECT *
FROM users");

pdo=newPDO("mysql:host=localhost;dbname=database","username","password");pdo = new PDO(
    "mysql:host=localhost;dbname=database",
    "username",
    "password"
);
pdo=newPDO("mysql:host=localhost;dbname=database","username","password");pdo→query("SELECT
* FROM users");
```

- Fonctions de gestion de sessions : par exemple, la fonction "session_start" permet de démarrer une session, tandis que la fonction "session_destroy" permet de la détruire.

```php
session_start();
session_destroy();
```

- Fonctions de gestion de emails : par exemple, la fonction "mail" permet d'envoyer un email, tandis que la fonction "imap_open" permet de se connecter à un serveur de messagerie et de récupérer les emails stockés sur celui-ci.

```php
isSent = mail(
    to: "test@gmail.com",
    subject: "Hello World",
    message: "Hello World"
);

$inbox = imap_open(
    "{imap.gmail.com:993/imap/ssl}INBOX",
    "username",
    "password"
);
```

Où les trouver ?

Vous pouvez trouver la liste complète des fonctions de base dans la documentation officielle de PHP, disponible sur le site web suivant : https://www.php.net/manual/fr/indexes.functions.php. Cette liste comportent presque 9000 fonctions, ce qui peut être un peu intimidant au premier abord. Cela fait aussi de PHP une language très riche et très puissant.

Pour trouver une fonction particulière, cliquez sur le lien correspondant dans la liste des fonctions ou saisir le nom de la fonction dans la barre de recherche.

La documentation de chaque fonction indique son nom, sa syntaxe, sa description et son exemple d'utilisation. Elle précise également les valeurs renvoyées par la fonction et les éventuelles erreurs qui peuvent survenir lors de son utilisation.

Il est également possible de trouver des tutoriels et des exemples d'utilisation des fonctions de PHP sur différents sites web et forums dédiés au développement web. Ces ressources peuvent particulièrement être utiles pour comprendre comment utiliser les fonctions de manière concrète dans le cadre de vos projets de développement.

Comment lire la documentation ?

Les informations suivantes sont disponibles dans la documentation de chaque fonction :

- **Paramètres** : la documentation de chaque fonction indique la liste des paramètres qu'elle attend en entrée. Par exemple, la fonction `str_replace` prend trois paramètres : la chaîne de caractères à rechercher, la chaîne de caractères de remplacement et la chaîne de caractères dans laquelle effectuer la recherche.
- **Paramètres facultatifs** : certains paramètres peuvent être optionnels et ne pas être obligatoirement fournis lors de l'appel de la fonction. Par exemple, la fonction `date` prend un paramètre facultatif qui indique le format de la date à renvoyer. Si ce paramètre n'est pas fourni, la fonction renvoie la date au format par défaut.
- **Type de retour** : la documentation de chaque fonction indique le type de données qu'elle renvoie. Par exemple, la fonction "strtoupper" renvoie une chaîne de caractères en majuscules, tandis que la fonction "array_sum" renvoie un nombre entier représentant la somme des éléments d'un tableau.

- **Flags** : certains paramètres peuvent être des "flags", c'est-à-dire des valeurs qui activent ou désactivent certaines options de la fonction. Par exemple, la fonction `htmlspecialchars` prend un flag qui indique si les caractères spéciaux doivent être convertis ou non en entités HTML.

Vous trouverez aussi dans la documentation de chaque fonction des exemples d'utilisation.

Voici un exemple de documentation de la fonction `str_replace` (https://www.php.net/manual/fr/function.str-replace.php) :

```php
 str_replace(
    array|string search,    array|stringsearch,
    array|string search,    array|stringreplace,
    string|array subject,
    int &count = null
): string|array
```

- `$search` : chaîne de caractères à rechercher ou tableau de chaînes de caractères à rechercher
- `$replace` : chaîne de caractères de remplacement ou tableau de chaînes de caractères de remplacement
- `$subject` : chaîne de caractères ou tableau de chaînes de caractères dans laquelle effectuer la recherche
- `$count` (optionnel) : référence à un entier qui sera rempli avec le nombre de remplacements effectués

`Retourne` : chaîne de caractères avec les remplacements effectués

Cette fonction remplace toutes les occurrences de la chaîne de caractères `$search` par la chaîne de caractères `$replace` dans la chaîne de caractères `$subject`. Si `$search` et `$replace` sont des tableaux, tous les éléments de `$search` sont remplacés par les éléments de `$replace` ayant le même index. Si `$replace` a moins d'éléments que `$search`, les éléments de `$replace` ayant un index supérieur à celui de la dernière occurrence de `$search` sont ignorés.

1.7 Les fonctions utilisateurs

Les fonctions définies par l'utilisateur sont des blocs de code qui peuvent être exécutés plusieurs fois à différents endroits de votre script. Elles vous permettent de structurer votre code et de le rendre plus lisible et maintenable en le découpant en parties plus petites et plus faciles à gérer.

Voici comment définir une fonction :

```php
function nom_de_ma_fonction(arg1,arg1, arg1,arg2, ...) {
// corps de la fonction
}
```

Le nom de la fonction doit suivre les règles de nommage des variables.

Les arguments de la fonction sont des variables passées à la fonction lors de son appel et qui peuvent être utilisées à l'intérieur du corps de la fonction. Vous pouvez définir autant d'arguments que vous le souhaitez, en les séparant par une virgule.

Le corps de la fonction est le code qui sera exécuté lorsque la fonction sera appelée. Il peut contenir des instructions de contrôle de flux, des appels à d'autres fonctions, etc.

Voici comment appeler une fonction :

```php
nom_de_ma_fonction(arg1value,arg1_value, arg1value,arg2_value, ...);
```

Vous devez fournir des valeurs pour chaque argument de la fonction, en les séparant par une virgule. Les valeurs sont passées à la fonction et affectées aux variables définies dans la liste des arguments.

Les fonctions peuvent également retourner une valeur avec l'instruction `return`. Par exemple :

```php
function addition(a,a, a,b) {
return a+a + a+b;
}

resultat=addition(1,2);//resultat = addition(1, 2); // resultat=addition(1,2);//resultat vaut 3
```

Vous pouvez utiliser des fonctions pour réaliser des tâches répétitives, organiser votre code et le rendre plus lisible et maintenable. Ainsi, elles sont un élément essentiel de la programmation en PHP et sont largement utilisées dans de nombreux scripts et applications.

Arguments par défaut

Vous pouvez donner à un argument de fonction une valeur par défaut en le déclarant de cette façon :

```php
function example(arg1,arg1, arg1,arg2 = 'default value')
{
    // code de la fonction
}
```

Cela signifie que si vous appelez la fonction `example` sans fournir de valeur pour `$arg2`, cette dernière prendra la valeur par défaut `'default value'`. Vous pouvez toutefois passer une valeur pour `$arg2` lors de l'appel de la fonction si vous le souhaitez :

```php
example('some value', 'another value');
```

Les arguments avec une valeur par défaut doivent être déclarés après les arguments sans valeur par défaut. Par exemple, la déclaration suivante est incorrecte :

```php
function example(arg1='defaultvalue',arg1 = 'default value', arg1='defaultvalue',arg2)
{
    // code de la fonction
}
```

Cela entraînera une erreur, car `$arg2` ne possède pas de valeur par défaut et doit donc être déclaré après tous les arguments qui en ont une.

Voici un exemple complet d'utilisation d'arguments par défaut :

```php
function greet(name,name, name,greeting = 'Hello')
{
    echo greeting.',’.greeting . ', ' . greeting.',’.name . '!';
}

greet('John');    // affiche 'Hello, John!'
greet('John', 'Hi');   // affiche 'Hi, John!'
```

Nombre variable d'arguments

En PHP, vous pouvez utiliser la fonction `func_get_args` pour accéder à un tableau contenant tous les arguments passés à une fonction, même si cette fonction ne déclare pas explicitement ces arguments. Voici un exemple de comment cela peut être fait :

```php
function example($arg1)
{
$args = func_get_args();
// $args est un tableau contenant tous les arguments passés à la fonction, y compris $arg1

    // Vous pouvez utiliser $args comme vous le souhaitez, par exemple pour itérer dessus ou pour en extraire des valeurs spécifiques
    foreach ($args as $arg) {
        // faire quelque chose avec $arg
    }

}

// Vous pouvez appeler la fonction avec un nombre variable d'arguments
example(1, 2, 3, 4);
```

La surcharge de fonction, en revanche, est un concept de programmation qui permet à une fonction de posséder plusieurs définitions, chacune ayant un nombre différent de paramètres ou des types de paramètres différents. Lorsque vous appelez une fonction surchargée, le compilateur ou l'interpréteur du langage choisit la définition qui correspond le mieux aux arguments passés à la fonction. En PHP, il est impossible de surcharger les fonctions de cette façon.

Voici un exemple de surcharge de fonction en C++ :

```
#include <iostream>

void print(int x)
{
std::cout << "Entier : " << x << std::endl;
}

void print(double x)
{
std::cout << "Reel : " << x << std::endl;
}

int main()
{
print(1); // appelle la fonction print(int)
print(1.5); // appelle la fonction print(double)

    return 0;

}
```

Déballage d'arguments

Le déballage des arguments est une façon de passer un tableau ou un objet "iterable" (c'est-à-dire une classe implémentant l'interface `Traversable`) en tant qu'arguments d'une fonction. Cela permet de décomposer le contenu du tableau ou de l'objet en une série d'arguments individuels passés à la fonction.

Voici un exemple d'utilisation du déballage des arguments avec un tableau :

PHP

```
function sum(x,x, x,y, z)
{
    return $x + $y + $z;
}

numbers = [1, 2, 3];
echo sum(...$numbers);  // affiche 6
```

Ici, le tableau `$numbers` est décomposé en trois arguments individuels (1, 2 et 3) passés à la fonction `sum`.

Il est important de noter que le déballage des arguments nécessite l'opérateur de répartition `...` en PHP 7.4 ou une version ultérieure. Si vous utilisez une version antérieure de PHP, vous devrez utiliser la fonction `call_user_func_array` au lieu du déballage des arguments :

PHP

```
numbers=[1,2,3];echocalluserfuncarray(’sum’,numbers = [1, 2, 3];
echo call_user_func_array('sum', numbers=[1,2,3];echocalluserfuncarray(’sum’,numbers);  //
affiche 6
```

Factorisation

La factorisation en programmation consiste à séparer un programme en plusieurs parties ou « facteurs » afin de le rendre plus facile à comprendre, à maintenir et à réutiliser. Cela peut être fait en décomposant un programme en sous-programmes indépendants, en utilisant des fonctions ou des modules, ou en organisant le code en différents fichiers ou dossiers.

Utilité

Il existe plusieurs raisons pour lesquelles la factorisation peut être utile :

- Elle peut rendre le code plus lisible et plus facile à comprendre en répartissant les tâches en différentes parties clairement définies.
- Elle peut permettre de réutiliser du code en créant des fonctions ou des modules qui peuvent être utilisés à plusieurs endroits dans le programme.
- Elle peut rendre le code plus simple à maintenir en permettant de modifier ou d'actualiser une partie du code sans avoir à toucher à l'ensemble du programme.
- Elle peut améliorer les performances du programme en autorisant la réutilisation du code qui a déjà été exécuté au lieu de le recalculer chaque fois.

Il est important de noter que la factorisation n'est pas seulement bénéfique pour les programmes de grande taille, mais peut également être utile pour les programmes plus petits. En effet, en organisant le code de manière cohérente et en utilisant des fonctions ou des modules, même pour de petits programmes, vous pouvez rendre votre code plus facile à comprendre et à maintenir.

Exemple

Voici un exemple de factorisation d'un programme impératif avec des fonctions :

Voici le code impératif original :

```php
<?php

customers=[[’name’⇒’John’,’age’⇒25,’gender’⇒’male’],[’name’⇒’Jane’,
’age’⇒35,’gender’⇒’female’],[’name’⇒’Bill’,’age’⇒45,’gender’⇒’male’],[’name’⇒’Susan’,
’age’⇒55,’gender’⇒’female’],[’name’⇒’Mike’,’age’⇒65,’gender’⇒’male’],];customers = [
    ['name' ⇒ 'John', 'age' ⇒ 25, 'gender' ⇒ 'male'],
    ['name' ⇒ 'Jane', 'age' ⇒ 35, 'gender' ⇒ 'female'],
    ['name' ⇒ 'Bill', 'age' ⇒ 45, 'gender' ⇒ 'male'],
    ['name' ⇒ 'Susan', 'age' ⇒ 55, 'gender' ⇒ 'female'],
    ['name' ⇒ 'Mike', 'age' ⇒ 65, 'gender' ⇒ 'male'],
];

customers=[[’name’⇒’John’,’age’⇒25,’gender’⇒’male’],[’name’⇒’Jane’,
’age’⇒35,’gender’⇒’female’],[’name’⇒’Bill’,’age’⇒45,’gender’⇒’male’],[’name’⇒’Susan’,
’age’⇒55,’gender’⇒’female’],[’name’⇒’Mike’,’age’⇒65,’gender’⇒’male’],];males = [];
females=[];foreach(females = [];

foreach (females=[];foreach(customers as customer) {
    if ($customer['gender'] == 'male') {
        $males[] = $customer;
    } else {
        $females[] = $customer;
    }
}

echo 'Males:';
foreach (males as male) {
    echo $male['name'] . ' ' . $male['age'] . ' ' . $male['gender'] . PHP_EOL;
}

echo PHP_EOL;

echo 'Females:';
foreach (females as $female) {
    echo $female['name'] . ' ' . $female['age'] . ' ' . $female['gender'] . PHP_EOL;
}

echo PHP_EOL;
```

Voici comment ce code pourrait être factorisé grâce à des fonctions :

```php
<?php

function getCustomers()
{
    return [
        ['name' => 'John', 'age' => 25, 'gender' => 'male'],
        ['name' => 'Jane', 'age' => 35, 'gender' => 'female'],
        ['name' => 'Bill', 'age' => 45, 'gender' => 'male'],
        ['name' => 'Susan', 'age' => 55, 'gender' => 'female'],
        ['name' => 'Mike', 'age' => 65, 'gender' => 'male'],
    ];
}

function filterCustomersByGender(array customers,stringcustomers, string
customers,stringgender)
{
    filteredCustomers=[];foreach(filteredCustomers = [];
    foreach (filteredCustomers=[];foreach(customers as customer) {
        if ($customer['gender'] == $gender) {
            $filteredCustomers[] = $customer;
        }
    }

    return filteredCustomers;
}

function printCustomers(array customers)
{
    foreach ($customers as $customer) {
        echo $customer['name'] . ' ' . $customer['age'] . ' ' . $customer['gender'] .
PHP_EOL;
    }
}

customers = getCustomers();

males=filterCustomersByGender(males =
filterCustomersByGender(males=filterCustomersByGender(customers, 'male');
females=filterCustomersByGender(females =
filterCustomersByGender(females=filterCustomersByGender(customers, 'female');

echo 'Males:' . PHP_EOL;
printCustomers(males);echoPHPEOL;echo’Females:’.PHPEOL;printCustomers(males);

echo PHP_EOL;

echo 'Females:' . PHP_EOL;
printCustomers(males);echoPHPEOL;echo’Females:’.PHPEOL;printCustomers(females);

echo PHP_EOL;
```

Voici comment le programme pourrait être écrit avec des classes et en définissant une méthode pour chaque tâche (on apprendra à utiliser les classes dans le chapitre suivant) :

PHP

```php
<?php

class Customer
{
    public name;publicname;
    public name;publicage;
    public gender;

    public function __construct(name, age,age, age,gender)
    {
        this->name=this→name = this->name=name;
        this->age=this→age = this->age=age;
        this->gender=this→gender = this->gender=gender;
    }
}

class CustomerFilter
{
    public static function byGender(array customers,stringcustomers, string
customers,stringgender)
    {
        filteredCustomers=[];foreach(filteredCustomers = [];
        foreach (filteredCustomers=[];foreach(customers as customer) {
            if ($customer→gender == $gender) {
                $filteredCustomers[] = $customer;
            }
        }

        return filteredCustomers;
    }
}

class CustomerPrinter
{
    public static function print(array customers)
    {
        foreach ($customers as $customer) {
            echo $customer→name . ' ' . $customer→age . ' ' . $customer→gender . PHP_EOL;
        }
    }
}

customers = [
    new Customer('John', 25, 'male'),
    new Customer('Jane', 35, 'female'),
    new Customer('Bill', 45, 'male'),
    new Customer('Susan', 55, 'female'),
    new Customer('Mike', 65, 'male'),
];

males=CustomerFilter::byGender(males =
CustomerFilter::byGender(males=CustomerFilter::byGender(customers, 'male');
females=CustomerFilter::byGender(females =
CustomerFilter::byGender(females=CustomerFilter::byGender(customers, 'female');
```

```
echo 'Males:' . PHP_EOL;
CustomerPrinter::print(males);echoPHPEOL;echo'Females:
'.PHPEOL;CustomerPrinter::print(males);

echo PHP_EOL;

echo 'Females:' . PHP_EOL;
CustomerPrinter::print(males);echoPHPEOL;echo'Females:'.PHPE
OL;CustomerPrinter::print(females);
```

Dans cet exemple, nous avons créé une classe `Customer` qui représente un client, une classe `CustomerFilter` qui contient une méthode statique `byGender` pour filtrer les clients par genre, et une classe `CustomerPrinter` qui contient une méthode statique `print` pour imprimer les informations sur les clients.

Cette factorisation permet de séparer explicitement les différentes tâches du programme en différentes parties et de rendre le code plus simple à comprendre et à maintenir. Elle permet ainsi de réutiliser ces méthodes à plusieurs endroits dans le programme sans avoir à réécrire le même code à chaque fois.

Fonctions anonymes

Les fonctions anonymes, également appelées "fonctions lambda" ou "fonctions closures" en PHP, sont des fonctions qui ne possèdent pas de nom, qui sont définies et exécutées sur place, sans être enregistrées dans le code. Elles sont souvent utilisées lorsque vous devez définir une fonction pour une utilisation unique ou temporaire, ou pour passer une fonction en tant qu'argument à une autre fonction.

Voici comment déclarer une fonction anonyme :

PHP

```
greet=function(greet = function(greet=function(name) {
    return "Hello, name!";
};

echo greet('John'); // affiche "Hello, John!"
```

Il est également possible de déclarer une fonction anonyme avec une syntaxe abrégée avec la flèche `⇒` (*arrow function syntax* en anglais) depuis PHP 7.4 :

PHP

```
greet=fn(greet = fn(greet=fn(name) ⇒ "Hello, name!";echoname!";

echo name!";echogreet('Jane'); // affiche "Hello, Jane!"
```

Les fonctions anonymes peuvent être utilisées de la même manière que les fonctions normales. Par exemple en les passant en tant qu'arguments à d'autres fonctions ou en les utilisant comme valeurs de retour :

```php
function arrayFilter(array array,callablearray, callable array,callablecallback)
{
    filteredArray=[];foreach(filteredArray = [];
    foreach (filteredArray=[];foreach(array as value) {
        if ($callback($value)) {
            $filteredArray[] = $value;
        }
    }

    return filteredArray;
}

numbers=[1,2,3,4,5,6,7,8,9,10];numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

numbers=[1,2,3,4,5,6,7,8,9,10];evenNumbers = arrayFilter(numbers,function(numbers,
function(numbers,function(number) {
    return number % 2 == 0;
});

print_r(evenNumbers); // affiche [2, 4, 6, 8, 10]
```

Appel de fonctions dynamique

Les appels de fonctions dynamiques permettent d'appeler une fonction dont le nom est déterminé au `runtime`, c'est-à-dire lorsque le programme est exécuté. Cela peut être utile lorsque vous devez déterminer le nom de la fonction à appeler en fonction de certaines conditions ou de la valeur d'une variable.

Voici comment effectuer un appel de fonction dynamique :

```php
function greet(name)
{
    return "Hello, $name!";
}

functionName = 'greet';

echo $functionName('John'); // affiche "Hello, John!"
```

Il est également possible d'utiliser la fonction `call_user_func` ou `call_user_func_array` pour appeler une fonction dynamiquement en spécifiant le nom de la fonction et, éventuellement, ses arguments en tant qu'arguments de ces fonctions :

```php
echo call_user_func('greet', 'Jane'); // affiche "Hello, Jane!"

echo call_user_func_array('greet', ['Mike']); // affiche "Hello, Mike!"
```

Quand vous utilisez un appel de fonction dynamique, vous devez vérifier que la fonction existe avant de l'appeler, afin d'éviter les erreurs. Vous pouvez utiliser la fonction `function_exists` pour vérifier si une fonction existe avant de l'appeler.

```php
if (function_exists('greet')) {
    echo greet('John');
}
```

Il est également possible d'utiliser les appels de fonction dynamiques avec les méthodes de classe en utilisant la fonction `call_user_func_array` et en spécifiant l'objet et le nom de la méthode en tant qu'arguments. Par exemple :

```php
class Hello
{
    public function greet(name)
    {
        return "Hello, $name!";
    }
}

hello = new Hello();
methodName='greet';echocalluserfuncarray([methodName = 'greet';

echo call_user_func_array([methodName='greet';echocalluserfuncarray([hello, $methodName],
['John']); // affiche "Hello, John!"
```

Les appels de fonction dynamiques peuvent rendre le code plus difficile à lire et à maintenir, il faut donc les utiliser avec prudence et uniquement quand cela est nécessaire. Néanmoins, c'est un outil extrêmement puissant qui peut être utilisé pour créer des fonctionnalités avancées.

Récursivité

La récursivité en programmation est un concept qui consiste à définir une fonction ou une procédure qui s'appelle elle-même de manière répétée pour résoudre un problème. La récursivité peut être utilisée lorsque le problème à résoudre peut-être divisé en sous-problèmes comparables au problème original, de manière à ce que chaque sous-problème puisse être résolu de manière indépendante.

Factorielle

Voici un exemple de fonction récursive qui calcule la factorielle d'un nombre :

```php
function factorial($number)
{
    if ($number <= 1) {
        return 1;
    } else {
        return $number * factorial($number - 1);
    }
}

echo factorial(5); // affiche 120
```

Dans cet exemple, la fonction `factorial` s'appelle elle-même avec un nombre inférieur de 1 à chaque itération jusqu'à ce que le nombre atteigne 1, puis elle retourne 1. Chaque itération calcule le produit du nombre courant avec le résultat de l'itération précédente, ce qui permet de calculer la factorielle du nombre donné.

Il convient de prendre en compte que la récursivité doit être utilisée avec prudence, car elle peut entraîner des appels de fonction répétés, ce qui entraîne une charge de mémoire et une complexité accrue du code. Il est impératif de disposer d'une condition de fin pour arrêter l'appel récursif et de s'assurer que le problème peut effectivement être divisé en sous-problèmes de manière à ce qu'une solution récursive soit possible.

Voici comment l'exécution de l'appel à la fonction `factorial(5)` se déroule étape par étape :

1. Appel de factorial(5)
2. if ($number <= 1) retourne false, la fonction s'exécute donc avec return 5 * factorial(4)
3. Appel de factorial(4)
4. if ($number <= 1) retourne false, la fonction s'exécute donc avec return 4 * factorial(3)
5. Appel de factorial(3)
6. if ($number <= 1) retourne false, la fonction s'exécute donc avec return 3 * factorial(2)
7. Appel de factorial(2)
8. if ($number <= 1) retourne false, la fonction s'exécute donc avec return 2 * factorial(1)
9. Appel de factorial(1)
.0. if ($number <= 1) retourne true, la fonction retourne donc 1
.1. La valeur de factorial(1) est retournée à l'appel de factorial(2), qui retourne 2 * 1 = 2
.2. La valeur de factorial(2) est retournée à l'appel de factorial(3), qui retourne 3 * 2 = 6
.3. La valeur de factorial(3) est retournée à l'appel de factorial(4), qui retourne 4 * 6 = 24
.4. La valeur de factorial(4) est retournée à l'appel de factorial(5), qui retourne 5 * 24 = 120
.5. La valeur de factorial(5), soit 120, est retournée à l'appel initial de la fonction.

Voici comment l'arbre d'appels récursif ressemble pour cet exemple :

```yaml
factorial(5)
  |
  |
factorial(4)
  |
  |
factorial(3)
  |
  |
factorial(2)
  |
  |
factorial(1)
```

Fibonacci

La suite de Fibonacci est une suite de nombres entiers dans laquelle chaque nombre est la somme des deux nombres précédents. La suite de Fibonacci commence généralement par 0 et 1, et les nombres suivants sont calculés en additionnant les deux nombres précédents. Voici les premiers éléments de la suite de Fibonacci :

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

Voici comment la fonction récursive Fibonacci pourrait être écrite :

```php
function fibonacci($number)
{
    if ($number ⩽ 1) {
        return $number;
    } else {
        return fibonacci($number - 1) + fibonacci($number - 2);
    }
}

echo fibonacci(8); // affiche 21
```

Cette fonction calcule la valeur de l'élément de la suite de Fibonacci à l'index donné en appelant récursivement la fonction avec l'index précédent et l'index précédent à celui-ci jusqu'à ce que l'index atteigne 0 ou 1, moment où la fonction retourne cet index. Chaque appel récursif retourne la somme des deux valeurs précédentes, ce qui permet de calculer la valeur de l'élément de la suite de Fibonacci correspondant à l'index donné.

Voici comment l'arbre d'appels récursif ressemble lorsque la fonction fibonacci est appelée avec l'index 5, en considérant que les appels fibonacci(1) et fibonacci(2) sont effectués plusieurs fois :

```yaml
fibonacci(5)
  |
  |
fibonacci(4)
  |
  |
fibonacci(3)
  |         \
  |           \
fibonacci(2)   fibonacci(1)
  |               \
  |                \
fibonacci(1)     fibonacci(0)
```

Il est important de noter que les appels `fibonacci(1)` et `fibonacci(2)` sont effectués à plusieurs reprises, ce qui rend cette implémentation de la fonction peu efficace pour calculer des valeurs de la suite de Fibonacci pour des indices élevés. Pour améliorer l'efficacité, il est recommandé d'utiliser une approche différente, telle que la programmation dynamique, qui stocke les résultats des appels récursifs dans un tableau afin de ne pas avoir à les recalculer à chaque fois.

```php
function fibonacci($number)
{
    $fibonacci = [0, 1];

    for ($i = 2; $i ⩽ $number; $i++) {
        $fibonacci[$i] = $fibonacci[$i - 1] + $fibonacci[$i - 2];
    }

    return $fibonacci[$number];
}

echo fibonacci(8); // affiche 21
```

1.8 Les chaines de caractères

Les chaînes de caractères sont des séquences de caractères utilisées pour représenter du texte. Elles peuvent être définies de différentes manières :

- En utilisant des guillemets simples `'` :

```php
$chaine = 'Ceci est une chaîne de caractères';
```

- En utilisant des guillemets doubles `"` :

```php
$chaine = "Ceci est une chaîne de caractères";
```

Il est également possible de définir une chaîne de caractères sur plusieurs lignes en utilisant la syntaxe `heredoc` :

```php
$chaine = <<<TEXT
Ceci est une chaîne de caractères
sur plusieurs lignes.
TEXT;
```

Concaténation

En PHP, il est possible de concaténer des chaînes de caractères en utilisant le point `.` :

```php
chaine1=’Bonjour’;chaine1 = 'Bonjour';
chaine1=’Bonjour’;chaine2 = ' comment vas-tu ?';
chainecomplete=chaine_complete = chainecomplete=chaine1 . $chaine2; // 'Bonjour comment vas-tu ?'
```

Manipulation de chaînes

Il est également possible de récupérer une sous-chaîne d'une chaîne en utilisant la fonction `substr()`. Par exemple, pour récupérer les 4 premiers caractères d'une chaîne, on peut utiliser la syntaxe suivante :

```php
chaine=’Bonjour’;chaine = 'Bonjour';
chaine=’Bonjour’;sous_chaine = substr($chaine, 0, 4); // 'Bonj'
```

Il existe de nombreuses autres fonctions permettant de manipuler les chaînes de caractères, telles que `strlen()` pour obtenir la longueur d'une chaîne, `strtolower()` pour convertir une chaîne en minuscules, ou encore `str_replace()` pour remplacer un sous-texte par un autre dans une chaîne.

- strlen() : renvoie la longueur d'une chaîne de caractères.
- strtolower() : convertit une chaîne de caractères en minuscules.
- strtoupper() : convertit une chaîne de caractères en majuscules.
- ucfirst() : met le premier caractère d'une chaîne en majuscule.
- ucwords() : met le premier caractère de chaque mot d'une chaîne en majuscule.
- str_replace() : remplace un sous-texte par un autre dans une chaîne de caractères.
- substr() : renvoie une sous-chaîne d'une chaîne de caractères.
- trim() : supprime les espaces en début et fin de chaîne.
- explode() : divise une chaîne de caractères en plusieurs sous-chaînes en utilisant un délimiteur.
- implode() : assemble des éléments d'un tableau en une chaîne de caractères en utilisant un délimiteur.
- htmlentities() : convertit les caractères spéciaux en équivalents HTML.

Formatage de chaînes

Il existe plusieurs façons de formater une chaîne de caractères :

1. Utiliser la fonction `sprintf()` : cette fonction permet de formater une chaîne de caractères en utilisant des placeholders représentés par des symboles `%` suivis d'une lettre indiquant le type de données à afficher (par exemple `%s` pour une chaîne de caractères, `%d` pour un entier, etc.). Par exemple :

```php
nom=’John’;nom = 'John';
nom=’John’;age = 30;
chaine=sprintf(’Bonjour,jema´ppellechaine = sprintf('Bonjour, je m\'appelle %s et j\'ai %d
ans', chaine=sprintf(’Bonjour,jema´ppellenom, age);//age);
// age);//chaine vaut 'Bonjour, je m'appelle John et j'ai 30 ans'
```

1. Utiliser la concaténation avec le point `.` : cette méthode permet de fusionner plusieurs chaînes de caractères en une seule en utilisant le point `.` comme opérateur de concaténation. Par exemple :

```php
nom=’John’;nom = 'John';
nom=’John’;age = 30;
chaine=’Bonjour,jema´ppelle’.chaine = 'Bonjour, je m\'appelle ' .
chaine=’Bonjour,jema´ppelle’.nom . ' et j\'ai ' . age.’ans’;//age . ' ans';
// age.’ans’;//chaine vaut 'Bonjour, je m'appelle John et j'ai 30 ans'
```

1. Utiliser la syntaxe heredoc : cette syntaxe permet de définir une chaîne de caractères sur plusieurs lignes en utilisant la syntaxe `<<<` suivie d'un identifiant, puis la chaîne de caractères elle-même, et enfin l'identifiant à nouveau pour terminer la chaîne. Par exemple :

```php
nom=’John’;nom = 'John';
nom=’John’;age = 30;
chaine=<<<EOTBonjour,jem’appellechaine = <<<EOT
Bonjour,
je m'appelle chaine=<<<EOTBonjour,jem’appellenom et j'ai ageans.EOT;//age ans.
EOT;
// ageans.EOT;//chaine vaut 'Bonjour,\nje m'appelle John et j'ai 30 ans.'
```

Il existe également d'autres façons de formater une chaîne de caractères. Notamment, grâce à des fonctions de mise en forme telles que `nl2br()` pour convertir les retours à la ligne en balises HTML `<br>`, ou encore `str_pad()` pour ajouter des caractères de remplissage à une chaîne.

Exercices de lecture

1.1.

```php
text="Thisisanexampleoftext";text = "This is an example of text";
text="Thisisanexampleoftext";search = "ex";
replace="EX";replace = "EX";

replace="EX";newText = str_replace(search,search, search,replace, text);text);
text);newText = strtoupper(newText);newText);
newText);newText = strrev(newText);echonewText);

echo newText);echonewText . PHP_EOL;
```

1.2.

```php
string1="Thisstring";string1 = "   This string   ";
string1="Thisstring";string2 = "   has extra spaces ";
string3="atthebeginningandend";string3 = " at the beginning and end  ";

string3="atthebeginningandend";newString = trim(string1)."".strtoupper(trim(string1) . " " .
strtoupper(trim(string1)."".strtoupper(trim(string2)) . " " .
strtolower(trim(string3));echostring3));

echo string3));echonewString . PHP_EOL;
```

1.3

```php
string1="Thisstringhas";string1 = "  This  string  has  ";
string1="Thisstringhas";string2 = " lots of ";
string3="whitespaceand";string3 = " whitespace and ";
string3="whitespaceand";string4 = " unwanted characters ";
string5 = " !@#%^&* ";

string1 = trim(string1);string1);
string1);string2 = ucwords(strtolower(trim(string1)));string1)));
string1)));string3 = str_replace(" ", "-", trim(string3));string3));
string3));string5 = preg_replace("/[^a-zA-Z0-9]/", "", string5);string5);
string5);string4 = strrev(trim(string4));string4));

string4));newString = string1."".string1 . " " . string1."".string2 . " " .
string3."".string3 . " " . string3."".string4 . " " . string5;echostring5;

echo string5;echonewString . PHP_EOL;
```

Exercices

Exercice 1 : Générateur de phrases aléatoires

Dans cet exercice, vous allez créer une fonction qui génère une phrase aléatoire. Vous aurez besoin de trois listes de chaînes de caractères : une liste de sujets, une liste de verbes et une liste de compléments. Votre fonction devrait choisir un sujet, un verbe et un complément au hasard dans chaque liste et les

concaténer pour former une phrase.

Par exemple, si vos listes sont les suivantes :

- Sujets : ["Je", "Tu", "Il"]
- Verbes : ["mange", "vois", "aime"]
- Compléments : ["une pomme", "la télé", "la programmation"]

Votre fonction pourrait générer des phrases comme "Je vois la télé" ou "Il aime une pomme".

Exercice 2 : Conversion de texte en camel case

Écrivez une fonction qui prend en entrée une phrase en texte normal et la convertit en camel case. Le camel case est une convention d'écriture où chaque mot commence par une lettre majuscule, sauf le premier mot qui commence par une lettre minuscule. Par exemple, la phrase "bonjour tout le monde" devrait être convertie en "bonjourToutLeMonde".

Exercice 3 : Calcul du score d'un mot

Écrivez une fonction qui prend en entrée un mot et calcule son score en attribuant une valeur à chaque lettre selon sa position dans l'alphabet (A=1, B=2, etc.). Le score d'un mot est la somme des valeurs de ses lettres. Par exemple, le score du mot "hello" serait 8+5+12+12+15 = 52.

Exercice 4 : Encoder et Décoder une chaîne de caractères

Écrivez une fonction `encode` qui prend en entrée une chaîne de caractères et renvoie une nouvelle chaîne où chaque caractère est remplacé par le caractère qui se trouve trois positions après lui dans l'alphabet. Par exemple, 'a' deviendrait 'd', 'b' deviendrait 'e', etc. Pour les trois dernières lettres, 'x' deviendrait 'a', 'y' deviendrait 'b' et 'z' deviendrait 'c'.

Ensuite, écrivez une fonction `decode` qui fait l'opération inverse, c'est-à-dire qu'elle prend en entrée une chaîne de caractères encodée et renvoie la chaîne de caractères originale.

Exercice 5 : Compter les voyelles et les consonnes

Écrivez une fonction qui prend en entrée une chaîne de caractères et renvoie le nombre de voyelles et de consonnes dans cette chaîne. La fonction devrait être capable de gérer à la fois les lettres majuscules et minuscules. Par exemple, pour la chaîne "Bonjour tout le monde", la fonction devrait renvoyer "Il y a 8 voyelles et 9 consonnes".

1.9 Les tableaux

En PHP, les tableaux (ou arrays en anglais) sont des structures de données permettant de stocker plusieurs valeurs dans une seule variable. Ils sont très utiles lorsqu'il est nécessaire de manipuler des collections de données, comme par exemple une liste de noms ou un ensemble de valeurs numériques.

Les tableaux en PHP peuvent être définis de différentes manières :

- En utilisant la syntaxe suivante :

```php
$tableau = array(1, 2, 3, 4, 5);
```

- En utilisant la syntaxe suivante :

```php
$tableau = [1, 2, 3, 4, 5];
```

Il est également possible de définir un tableau associatif, c'est-à-dire un tableau dans lequel chaque valeur est associée à une clé :

```php
$tableau = [
  'cle1' => 'valeur1',
  'cle2' => 'valeur2',
  'cle3' => 'valeur3'
];
```

Il est possible de récupérer une valeur d'un tableau en utilisant la syntaxe suivante :

```php
valeur=valeur = valeur=tableau[$cle];
```

Les tableaux en PHP sont similaires aux tableaux de nombreux autres langages de programmation, mais ils ont quelques différences notables :

- En PHP, les tableaux peuvent contenir des éléments de différents types (entiers, chaînes de caractères, objets, etc.), alors que dans certains langages, les tableaux sont limités à un type de données spécifique.
- En PHP, les tableaux peuvent être associatifs (c'est-à-dire qu'ils peuvent avoir des clés qui ne sont pas des entiers), alors que dans certains langages, les tableaux sont toujours indexés par des entiers.
- En PHP, il est possible de définir des tableaux dynamiques (c'est-à-dire des tableaux dont la taille peut changer dynamiquement), alors que dans certains langages, les tableaux sont statiques (c'est-à-dire que leur taille est fixe).
- En PHP, il est possible de définir des tableaux hétérogènes (c'est-à-dire des tableaux contenant des éléments de différents types), alors que dans certains langages, les tableaux sont homogènes (c'est-à-dire qu'ils ne peuvent contenir que des éléments d'un seul type).

Cela rend les tableaux en PHP très flexibles et puissants, mais cela peut également les rendre plus dangereux à utiliser. Par exemple, il est possible de définir un tableau associatif dont les clés sont des chaînes de caractères, et de récupérer une valeur en utilisant une clé qui n'existe pas. Dans ce cas, PHP renverra une erreur, et le script sera interrompu.

Pour éviter ce genre de problème, il est recommandé de toujours vérifier que la clé utilisée pour récupérer une valeur d'un tableau existe bien. Par exemple, pour récupérer la valeur associée à la clé `cle1` dans le tableau défini ci-dessus, on peut utiliser la syntaxe suivante :

```php
if (array_key_exists('cle1', $tableau)) {
  $valeur = $tableau['cle1'];
}
```

Parcourir un tableau

Il est possible de parcourir un tableau en utilisant une boucle `foreach`. Par exemple, pour afficher toutes les valeurs d'un tableau, on peut utiliser la syntaxe suivante :

```php
foreach (tableauastableau as tableauasvaleur) {
  echo $valeur;
}
```

Il est également possible de récupérer la clé associée à chaque valeur en utilisant la syntaxe suivante :

```php
foreach (tableauastableau as tableauascle => $valeur) {
  echo $cle . ' => ' . $valeur;
}
```

Vous pouvez aussi parcourir un tableau en utilisant une boucle `for`. Par exemple, pour afficher toutes les valeurs d'un tableau, on peut utiliser la syntaxe suivante :

```php
for (i=0;i = 0; i=0;i < count(tableau);tableau); tableau);i++) {
  echo tableau[tableau[tableau[i];
}
```

Vous pouvez parcourir des tableaux a plusieurs dimensions en utilisant une boucle `foreach` imbriquée. Par exemple, pour afficher toutes les valeurs d'un tableau à deux dimensions, on peut utiliser la syntaxe suivante :

```php
foreach (tableauastableau as tableauasligne) {
  foreach (ligneasligne as ligneasvaleur) {
    echo $valeur;
  }
}
```

Ou alors, en utilisant une boucle `for` imbriquée :

```php
for (i=0;i = 0; i=0;i < count(tableau);tableau); tableau);i++) {
  for (j=0;j = 0; j=0;j < count(tableau[tableau[tableau[i]); $j++) {
    echo $tableau[$i][$j];
  }
}
```

Et cela fonctionne toujours pour des tableaux à trois dimensions, quatre dimensions, etc.

Fonctions utiles

Voici une liste non exhaustive de fonctions utiles pour manipuler des tableaux en PHP :

- count() : renvoie le nombre d'éléments d'un tableau.
- array_keys() : renvoie la liste des clés d'un tableau.
- array_values() : renvoie la liste des valeurs d'un tableau.
- array_merge() : fusionne plusieurs tableaux en un seul.
- array_slice() : extrait une portion de tableau.
- array_reverse() : renverse l'ordre des éléments d'un tableau.
- array_search() : recherche un élément dans un tableau et renvoie sa clé s'il est trouvé, ou false s'il n'est pas trouvé.
- array_key_exists() : vérifie si une clé existe dans un tableau.
- in_array() : vérifie si une valeur est présente dans un tableau.
- sort() : trie les éléments d'un tableau dans l'ordre croissant.
- rsort() : trie les éléments d'un tableau dans l'ordre décroissant.
- asort() : trie les éléments d'un tableau associatif en fonction de leurs valeurs.
- ksort() : trie les éléments d'un tableau associatif en fonction de leurs clés.

Fonctions de plus haut ordre

Les fonctions de haut ordre sont des fonctions qui prennent une ou plusieurs fonctions en argument, ou qui retournent une fonction. Elles peuvent être très utiles pour manipuler des tableaux, car elles permettent d'en traiter les éléments de manière très flexible et concise. Elles peuvent remplacer les boucles `for` et `foreach` dans de nombreux cas.

Voici quelques exemples de fonctions de haut ordre utiles pour manipuler des tableaux :

- array_filter() : cette fonction permet de filtrer les éléments d'un tableau en fonction d'un critère donné. Par exemple, pour récupérer uniquement les éléments pairs d'un tableau, on peut utiliser la syntaxe suivante :

```php
tableau=[1,2,3,4,5];tableau = [1, 2, 3, 4, 5];
tableau=[1,2,3,4,5];tableau_pairs = array_filter(tableau,function(tableau, function
(tableau,function(element) {
  return element % 2 == 0;
});
// tableau_pairs vaut [2, 4]
```

- array_map() : cette fonction permet d'appliquer une fonction à chaque élément d'un tableau. Par exemple, pour doubler chaque valeur d'un tableau, on peut utiliser la syntaxe suivante :

```php
tableau=[1,2,3,4,5];tableau = [1, 2, 3, 4, 5];
tableau=[1,2,3,4,5];tableau_doubles = array_map(function (element) {
  return $element * 2;
}, tableau);
// $tableau_doubles vaut array(2, 4, 6, 8, 10)
```

- array_reduce() : cette fonction permet de réduire un tableau à une seule valeur en utilisant une fonction de réduction. Par exemple, pour calculer la somme des éléments d'un tableau, on peut utiliser la syntaxe suivante :

```php
tableau=[1,2,3,4,5];tableau = [1, 2, 3, 4, 5];
tableau=[1,2,3,4,5];somme = array_reduce(tableau,function(tableau, function
(tableau,function(accumulateur, element) {
  return $accumulateur + $element;
}, 0);
// somme vaut 15
```

Exercices de lecture

```php
numbers=[3,17,4,2,16,18];numbers = [3, 17, 4, 2, 16, 18];

numbers=[3,17,4,2,16,18];min_number = min(numbers);numbers);
numbers);max_number = max(numbers);numbers);
numbers);sum_numbers = array_sum(numbers);echo"Minimumnumberis:".numbers);

echo "Minimum number is: " . numbers);echo"Minimumnumberis:".min_number . PHP_EOL;
echo "Maximum number is: " . maxnumber.PHPEOL;echo"Thesumofnumbersis:".max_number . PHP_EOL;
echo "The sum of numbers is: " . maxnumber.PHPEOL;echo"Thesumofnumbersis:".sum_numbers .
PHP_EOL;
```

1.

```php
fruits=["banana","apple","orange","lemon","apple","banana"];fruits = ["banana", "apple",
"orange", "lemon", "apple", "banana"];

fruits=["banana","apple","orange","lemon","apple","banana"];unique_fruits =
array_unique(fruits);sort(fruits);
sort(fruits);sort(unique_fruits);

fruitsstring=implode("and",arrayslice(fruits_string = implode(" and ", array_slice(fruitss
tring=implode("and",arrayslice(unique_fruits, 1));

echo "Fruits: " . $fruits_string . PHP_EOL;
```

1.

```php
products=[["name"⇒"Apple","category"⇒"Fruit","price"⇒0.5],
["name"⇒"Banana","category"⇒"Fruit","price"⇒0.2],
["name"⇒"Carrot","category"⇒"Vegetable","price"⇒0.3],
["name"⇒"Lemon","category"⇒"Fruit","price"⇒0.4],
["name"⇒"Tomato","category"⇒"Vegetable","price"⇒0.6]];products = [
    ["name" ⇒ "Apple", "category" ⇒ "Fruit", "price" ⇒ 0.5],
    ["name" ⇒ "Banana", "category" ⇒ "Fruit", "price" ⇒ 0.2],
    ["name" ⇒ "Carrot", "category" ⇒ "Vegetable", "price" ⇒ 0.3],
    ["name" ⇒ "Lemon", "category" ⇒ "Fruit", "price" ⇒ 0.4],
    ["name" ⇒ "Tomato", "category" ⇒ "Vegetable", "price" ⇒ 0.6]
];

products=[["name"⇒"Apple","category"⇒"Fruit","price"⇒0.5],
["name"⇒"Banana","category"⇒"Fruit","price"⇒0.2],
["name"⇒"Carrot","category"⇒"Vegetable","price"⇒0.3],
["name"⇒"Lemon","category"⇒"Fruit","price"⇒0.4],
["name"⇒"Tomato","category"⇒"Vegetable","price"⇒0.6]];fruits =
array_filter(products,fn(products, fn (products,fn(item) ⇒
item["category"]⩵⩵"Fruit");item["category"] ⩵⩵ "Fruit");
item["category"]⩵⩵"Fruit");fruits_name = array_column(fruits,
'name');echo"Thenameoffruitsis:".implode(",",fruits, 'name');

echo "The name of fruits is: " . implode(", ", fruits,
'name');echo"Thenameoffruitsis:".implode(",",fruits_name) . PHP_EOL;
```

Exercices

Filtrage et transformation

Énoncé : Écrivez une fonction `filtrageTransformation` qui prend en entrée un tableau de nombres et renvoie un nouveau tableau contenant les carrés des nombres pairs du tableau d'origine. Utilisez les fonctions `array_filter()` et `array_map()` pour effectuer ces opérations.

```php
function filtrageTransformation(numbers)//Exempled'utilisationnumbers) {

}

// Exemple d'utilisation
numbers)//Exempled'utilisationnumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
result=filtrageTransformation(result =
filtrageTransformation(result=filtrageTransformation(numbers);
print_r($result);
```

Calcul statistique

Énoncé : Écrivez une fonction `calculStatistique` qui prend en entrée un tableau de nombres et renvoie la moyenne des nombres. Utilisez les fonctions `array_reduce()` et `count()` pour effectuer ces opérations.

```php
function calculStatistique(numbers)//Exempled'utilisationnumbers) {

}

// Exemple d'utilisation
numbers)//Exempled'utilisationnumbers = [10, 15, 20, 25, 30];
result=calculStatistique(result = calculStatistique(result=calculStatistique(numbers);
echo "La moyenne est : " . $result;
```

Tri personnalisé

Énoncé : Écrivez une fonction `triPersonnalise` qui prend en entrée un tableau de chaînes de caractères et renvoie le tableau trié par ordre alphabétique en utilisant la longueur des chaînes de caractères comme critère de tri. Les chaînes de caractères plus courtes doivent être triées avant les plus longues. Utilisez la fonction `usort()` et une closure personnalisée pour effectuer ces opérations.

```php
function triPersonnalise(strings)//Exempled'utilisationstrings) {

}

// Exemple d'utilisation
strings)//Exempled'utilisationstrings = ["abcd", "ef", "ghijk", "lmnopq", "r"];
result=triPersonnalise(result = triPersonnalise(result=triPersonnalise(strings);
print_r($result);
```

Manipulation de données

Énoncé : Écrivez une fonction `manipulationDonnees` qui prend en entrée un tableau associatif contenant des informations sur des étudiants (nom, âge, note) et renvoie un nouveau tableau contenant uniquement le nom et l'âge des étudiants dont la note est supérieure à 80. Utilisez les fonctions `array_filter()` et `array_map()` avec une closure personnalisée pour effectuer ces opérations.

```php
function manipulationDonnees(students)//Exempled'utilisationstudents) {

}

// Exemple d'utilisation
students)//Exempled'utilisationstudents = [
    ['nom' => 'Alice', 'âge' => 20, 'note' => 85],
    ['nom' => 'Bob', 'âge' => 21, 'note' => 75],
    ['nom' => 'Charlie', 'âge' => 19, 'note' => 90],
    ['nom' => 'David', 'âge' => 22, 'note' => 95]
];
result=manipulationDonnees(result =
manipulationDonnees(result=manipulationDonnees(students);
print_r($result);
```

Recherche et transformation de données

Énoncé : Écrivez une fonction `rechercheTransformationDonnees` qui prend en entrée un tableau associatif contenant des informations sur des produits (nom, prix) et renvoie une chaîne de caractères contenant le nom et le prix de chaque produit dont le prix est inférieur à 10. Utilisez les fonctions `array_filter()` et `array_map()` avec une closure personnalisée pour effectuer ces opérations.

```php
function rechercheTransformationDonnees(products)products) {

}


products)products = [
    ['nom' => 'Produit A', 'prix' => 8.50],
    ['nom' => 'Produit B', 'prix' => 12.75],
    ['nom' => 'Produit C', 'prix' => 5.25],
    ['nom' => 'Produit D', 'prix' => 9.99]
];
result=rechercheTransformationDonnees(result =
rechercheTransformationDonnees(result=rechercheTransformationDonnees(products);
echo $result;
```

1.10 Les erreurs et exceptions

Les erreurs

En PHP, une erreur est un événement qui se produit lorsqu'il y a une anomalie dans le code qui ne peut pas être gérée de manière appropriée. Les erreurs peuvent être causées par de nombreux facteurs, tels que des erreurs de syntaxe, des références à des variables non définies, des appels à des fonctions inexistantes, etc.

Il existe plusieurs types d'erreurs en PHP, qui sont regroupées en plusieurs catégories :

- Les erreurs fatales : ce sont les erreurs les plus graves et qui interrompent immédiatement l'exécution du script. Elles incluent les erreurs de syntaxe, les erreurs de type (par exemple, lorsqu'une variable est utilisée comme un objet alors qu'elle n'en est pas un), etc.
- Les erreurs de type "warning" : ce sont des erreurs moins graves qui ne sont pas fatales, mais qui peuvent causer des problèmes si elles ne sont pas gérées. Elles incluent les références à des variables non définies, les appels à des fonctions inexistantes, etc.
- Les erreurs de type "notice" : ce sont des erreurs encore moins graves qui sont généralement causées par des accès à des variables non initialisées ou par des appels à des méthodes d'objet qui n'existent pas.

Il est possible de gérer les erreurs de différentes manières :

- En utilisant la fonction error_reporting() pour définir les types d'erreurs qui doivent être signalés par PHP.

- En utilisant la fonction set_error_handler() pour définir une fonction qui sera appelée chaque fois qu'une erreur est détectée. Cela permet de personnaliser la manière dont les erreurs sont gérées dans le script.

Voici quelques exemples de codes qui pourraient lever une erreur :

- Erreur de syntaxe :

```php
// Cette ligne de code générera une erreur de syntaxe car il manque une parenthèse fermante
$x = (5 + 3;
```

- Référence à une variable non définie :

```php
// Cette ligne de code générera une erreur de type "warning" car la variable yn'apase'te'de'finieechoy n'a pas été définie
echo yn'apase'te'de'finieechoy;
```

- Appel à une fonction inexistante :

```php
// Cette ligne de code générera une erreur de type "warning" car la fonction myFunction n'existe pas
myFunction();
```

- Accès à un élément d'un tableau qui n'existe pas :

```php
array=
[1,2,3];//Cettelignedecodege'ne'rerauneerreurdetype"notice"carl'e'le'menta`l'index4n'existepasdansletableauarray = [1, 2, 3];

// Cette ligne de code générera une erreur de type "notice" car l'élément à l'index 4 n'existe pas dans le tableau array=
[1,2,3];//Cettelignedecodege'ne'rerauneerreurdetype"notice"carl'e'le'menta`l'index4n'existepasdansletableauarray
echo $array[4];
```

Les exceptions

Les exceptions sont utilisées pour gérer de manière structurée les erreurs et les anomalies qui peuvent survenir lors de l'exécution d'un programme.

Les exceptions peuvent survenir pour différentes raisons, par exemple :

- Lorsqu'une fonction est appelée avec des arguments invalides.
- Lorsqu'on dépasse la valeur maximale d'un entier.
- Quand l'on divise un nombre par zéro.

En général, les exceptions sont levées par le code qui a détecté une erreur, et elles sont capturées par le code qui a appelé la fonction qui a levé l'exception. Cela permet de gérer les erreurs de manière appropriée et de s'assurer que le programme continue à fonctionner correctement.

Lever une exception

Pour lever une exception, vous devez utiliser l'instruction `throw`. Voici un exemple de code qui lève une exception :

```php
function divide(x,x, x,y) {
    if (y==0)thrownewException('Divisionparze´ro.');returny == 0) {
        throw new Exception('Division par zéro.');
    }

    return y==0)thrownewException('Divisionparze´ro.');returnx / $y;
}

echo divide(5, 0);
```

L'instruction `throw` prend en paramètre un objet de type Exception. Lorsque l'exception est levée, le programme s'arrête immédiatement et le code qui suit l'instruction `throw` n'est pas exécuté.

Capturer une exception

Pour capturer une exception, vous devez utiliser l'instruction `try...catch`. Voici un exemple de code qui capture une exception :

```php

function divide(x,x, x,y) {
    if (y==0)thrownewException('Divisionparze´ro.');returny == 0) {
        throw new Exception('Division par zéro.');
    }

    return y==0)thrownewException('Divisionparze´ro.');returnx / y;
}

try {
    echo divide(5, 0);
} catch (Exception e) {
    echo 'Une exception a été levée : ',  $e->getMessage(), "\n";
}
```

L'instruction `try...catch` prend en paramètre un bloc de code qui peut lever une exception, et un bloc de code qui sera exécuté si une exception est levée. L'instruction `try...catch` prend également en paramètre un objet de type `Exception` qui représente l'exception levée.

Si une exception est levée lors de l'exécution de notre programme, mais qu'aucune instruction `try...catch` ne la capture, alors le programme s'arrête immédiatement et le message d'erreur suivant est affiché :

```
Fatal error: Uncaught Exception: Division par zéro. in /home/username/test.php:3
Stack trace:
#0 /home/username/test.php(9): divide(5, 0)
#1 {main}
  thrown in /home/username/test.php on line 3
```

Depuis PHP 7.1, il est possible de capturer de traiter plusieurs type d'exception de manière différente. Voici un exemple de code qui capture plusieurs types d'exceptions :

```php
try {
    // Code qui peut lever une exception
} catch (ExceptionType1
e)//Codepourtraiterl'exceptiondetypeExceptionType1catch(ExceptionType2e) {
    // Code pour traiter l'exception de type ExceptionType1
} catch (ExceptionType2
e)//Codepourtraiterl'exceptiondetypeExceptionType1catch(ExceptionType2e) {
    // Code pour traiter l'exception de type ExceptionType2
} catch (Exception $e) {
    // Code pour traiter toutes les autres exceptions qui pourraient être levées
}
```

Les classes d'exceptions

Il existe plusieurs classes d'exceptions utilisées pour représenter différentes catégories d'erreurs. Voici la liste des classes d'exceptions qui sont définies dans PHP :

Error

- ArithmeticError
  - DivisionByZeroError
- AssertionError
- ParseError
- TypeError
- ArgumentCountError

Exception

- ClosedGeneratorException
- DOMException
- ErrorException
- IntlException
- LogicException
  - BadFunctionCallException
    - BadMethodCallException
  - DomainException
  - InvalidArgumentException

  - LengthException
  - OutOfRangeException
- PharException
- ReflectionException
- RuntimeException
  - OutOfBoundsException
  - OverflowException
  - PDOException
  - RangeException
  - UnderflowException
  - UnexpectedValueException
- SodiumException

Lorsque vous programmez, vous pouvez utiliser ces classes pour lever des exceptions dans votre code. C'est utile pour indiquer à l'utilisateur (un autre programmeur, ou l'utilisateur de votre site internet) qu'une erreur est survenue, et ainsi lui permettre de gérer l'erreur de manière appropriée.

Voici un exemple de code qui utilise la classe InvalidArgumentException pour lever une exception :

```php
function deleteFile(name) {
    if (!file_exists($name)) {
        throw new InvalidArgumentException('Le fichier ' . $name . ' n\'existe pas.');
    }

    unlink($name);
}

try {
    deleteFile('test.txt');
} catch (InvalidArgumentException e) {
    echo 'Une exception a été levée : ',  $e->getMessage(), "\n";
}
```

1.11 Fichiers

Lecture des fichiers

PHP offre plusieurs méthodes pour lire le contenu des fichiers.

Ouvrir un fichier

Pour lire un fichier, nous devons d'abord l'ouvrir en utilisant la fonction `fopen()`. Cette fonction prend deux paramètres : le chemin du fichier et le mode d'ouverture. Par exemple, pour ouvrir un fichier en mode lecture ('r'), nous utilisons :

```php
$file = fopen('file.txt', 'r');
```

Lecture ligne par ligne

Une fois le fichier ouvert, nous pouvons lire son contenu ligne par ligne en utilisant la fonction `fgets()`. Cette fonction retourne la ligne courante pointée par le pointeur de fichier, puis déplace le pointeur à la ligne suivante.

```php
while(!feof($file)) {
    $line = fgets($file);
    echo $line . "<br>";
}
```

Lecture intégrale

Si nous voulons lire tout le contenu d'un fichier en une seule fois, nous pouvons utiliser la fonction `file_get_contents()`. Cette fonction retourne tout le contenu du fichier sous forme de chaîne.

```php
content=filegetcontents('file.txt');echocontent = file_get_contents('file.txt');
echo content=filegetcontents('file.txt');echocontent;
```

Fermeture du fichier

Après avoir terminé la lecture du fichier, il est important de le fermer en utilisant la fonction `fclose()`. Cette fonction libère la ressource associée au fichier ouvert.

```php
fclose($file);
```

Écriture et modification des fichiers

PHP fournit également plusieurs méthodes pour écrire ou modifier le contenu d'un fichier.

Écriture dans un fichier

Pour écrire dans un fichier, nous devons d'abord l'ouvrir en mode écriture ('w') ou ajout ('a') en utilisant la fonction `fopen()`. Ensuite, nous utilisons la fonction `fwrite()` pour écrire dans le fichier.

```php
file=fopen('file.txt','w');fwrite(file = fopen('file.txt', 'w');
fwrite(file=fopen('file.txt','w');fwrite(file, "Contenu à écrire");
fclose($file);
```

Ajout de contenu à un fichier existant

Pour ajouter du contenu à un fichier existant sans écraser le contenu précédent, nous ouvrons le fichier en mode ajout ('a').

```php
file=fopen('file.txt','a');fwrite(file = fopen('file.txt', 'a');
fwrite(file=fopen('file.txt','a');fwrite(file, "\nContenu supplémentaire");
fclose($file);
```

Modification du contenu d'un fichier

Pour modifier le contenu d'un fichier, nous devons déplacer le pointeur de fichier à la position souhaitée en utilisant la fonction `fseek()`, puis écrire le nouveau contenu avec `fwrite()`.

```php
file=fopen('file.txt','r+');fseek(file = fopen('file.txt', 'r+');
fseek(file=fopen('file.txt','r+');fseek(file, 5);
fwrite(file,"nouveaucontenu");fclose(file, "nouveau contenu");
fclose(file,"nouveaucontenu");fclose(file);
```

Suppression du contenu d'un fichier

Pour supprimer tout le contenu d'un fichier, nous ouvrons le fichier en mode écriture ('w'), ce qui écrase automatiquement tout le contenu précédent.

```php
file=fopen('file.txt','w');fclose(file = fopen('file.txt', 'w');
fclose(file=fopen('file.txt','w');fclose(file);
```

Vérification de l'existence des fichiers

Avant d'ouvrir ou de manipuler un fichier, il est souvent nécessaire de vérifier si le fichier existe. PHP offre plusieurs fonctions pour cela.

Vérification de l'existence d'un fichier

La fonction `file_exists()` permet de vérifier si un fichier ou un répertoire existe. Elle retourne `true` si le fichier ou le répertoire existe, et `false` sinon.

```php
if(file_exists('file.txt')) {
    echo "Le fichier existe";
} else {
    echo "Le fichier n'existe pas";
}
```

La fonction `is_file()` vérifie si le chemin donné correspond à un fichier régulier (et non à un répertoire). Elle retourne `true` si le chemin correspond à un fichier, et `false` sinon.

```php
if(is_file('file.txt')) {
    echo "C'est un fichier";
} else {
    echo "Ce n'est pas un fichier";
}
```

Gestion des erreurs

Lors de la manipulation de fichiers en PHP, diverses erreurs peuvent survenir. Il est important de savoir comment les gérer.

Gestion des erreurs de fichier

PHP offre plusieurs fonctions pour gérer les erreurs lors de l'ouverture, de la lecture ou de l'écriture de fichiers. Par exemple, la fonction `feof()` permet de vérifier si le pointeur de fichier est à la fin du fichier, et la fonction `ferror()` permet de vérifier si une erreur est survenue lors de la lecture du fichier.

```php
file=@fopen('file.txt','r');if(!file = @fopen('file.txt', 'r');

if(!file=@fopen('file.txt','r');if(!file) {
    echo "Erreur lors de l'ouverture du fichier";
} else {
    while(!feof(file)) {
        $line = fgets($file);
        if(ferror($file)) {
            echo "Erreur lors de la lecture du fichier";
            break;
        }
        echo $line;
    }
    fclose(file);
}
```

Utilisation des exceptions pour la gestion des erreurs

PHP offre également un mécanisme d'exception pour gérer les erreurs de manière plus structurée. Vous pouvez utiliser les blocs `try`, `catch` et `finally` pour capturer et gérer les exceptions.

```php
try {
    file=fopen('file.txt','r');if(!file = fopen('file.txt', 'r');
    if(!file=fopen('file.txt','r');if(!file) {
        throw new Exception("Erreur lors de l'ouverture du fichier");
    }
    // Lecture du fichier...
} catch (Exception $e) {
    echo $e->getMessage();
} finally {
    if($file) {
        fclose($file);
    }
}
```

Fichiers en mode binaire

PHP permet également de lire et d'écrire des fichiers en mode binaire, ce qui est particulièrement utile pour manipuler des données non textuelles, comme des images ou des fichiers audio.

Lecture de fichiers binaires

Pour lire un fichier en mode binaire, nous ouvrons le fichier en mode binaire de lecture ('rb') et utilisons la fonction `fread()`. Cette fonction lit une quantité spécifiée d'octets à partir du fichier.

```php
file=fopen('image.jpg','rb');file = fopen('image.jpg', 'rb');
file=fopen('image.jpg','rb');content = fread(file,filesize('image.jpg'));fclose(file,
filesize('image.jpg'));
fclose(file,filesize('image.jpg'));fclose(file);
```

Écriture de fichiers binaires

Pour écrire des données binaires dans un fichier, nous ouvrons le fichier en mode binaire d'écriture ('wb') et utilisons la fonction `fwrite()`.

```php
file=fopen('imagecopy.jpg','wb');fwrite(file = fopen('image_copy.jpg', 'wb');
fwrite(file=fopen('imagecopy.jpg','wb');fwrite(file, content);fclose(content);
fclose(content);fclose(file);
```

Exercices

Exercices complémentaires

Journal de bord

Dans cet exercice, vous allez créer un journal de bord simple. Chaque fois que votre script PHP est exécuté, il doit ajouter une nouvelle entrée à un fichier journal. Chaque entrée doit contenir la date et l'heure actuelles, ainsi qu'un message de votre choix.

Pour cet exercice, vous devrez utiliser les fonctions `fopen()`, `fwrite()`, et `fclose()` pour manipuler le fichier journal. Vous devrez également utiliser la fonction `date()` pour obtenir la date et l'heure actuelles.

Gestionnaire de tâches

Dans cet exercice, vous allez créer un gestionnaire de tâches simple. Le gestionnaire de tâches doit permettre à l'utilisateur d'ajouter des tâches à une liste, de marquer des tâches comme terminées et de voir toutes les tâches non terminées. La liste des tâches doit être stockée dans un fichier.

Pour cet exercice, vous devrez utiliser les fonctions `fopen()`, `fwrite()`, `fgets()`, et `fclose()` pour lire et écrire dans le fichier de la liste de tâches. Vous devrez aussi gérer l'entrée de l'utilisateur pour ajouter des tâches et marquer les tâches comme terminées.

Copie de fichiers binaires

Dans cet exercice, vous allez écrire un script PHP qui copie un fichier image de votre choix et le sauvegarde sous un nouveau nom. Vous devrez vous assurer d'ouvrir les fichiers en mode binaire pour lire et écrire les données.

Pour cet exercice, vous devrez utiliser les fonctions `fopen()`, `fread()`, `fwrite()`, et `fclose()` pour lire le fichier image source, créer le fichier image de destination, et écrire les données de l'image source dans le fichier de destination. Vous devrez également utiliser la fonction `filesize()` pour déterminer combien d'octets lire du fichier source.

1.12 Dates

Introduction

Les dates jouent un rôle essentiel dans la manipulation du temps et de la chronologie en PHP. Dans ce chapitre, nous allons explorer les principales fonctionnalités liées aux dates en PHP. Nous apprendrons comment obtenir la date actuelle, formater les dates, manipuler les dates, comparer les dates, ajouter ou soustraire des intervalles de temps, et gérer les fuseaux horaires.

1. Obtenir la date actuelle

La fonction `date()` permet d'obtenir la date actuelle en PHP. Voici un exemple pour obtenir la date au format "année-mois-jour" :

```PHP
dateActuelle=date(’Y-m-d’);echodateActuelle = date('Y-m-d');
echo dateActuelle=date(’Y-m-d’);echodateActuelle;
```

Résultat :

```YAML
2023-09-15
```

2. Formater les dates

PHP offre une grande flexibilité pour formater les dates selon nos besoins. Voici un exemple pour afficher la date au format "jour/mois/année heure:minute:seconde" :

```PHP
dateActuelle=date(’d/m/YH:i:s’);echodateActuelle = date('d/m/Y H:i:s');
echo dateActuelle=date(’d/m/YH:i:s’);echodateActuelle;
```

Résultat :

```
15/09/2023 14:30:45
```

3. Manipuler les dates

Dans cette section, nous explorerons les fonctions PHP qui nous permettent de manipuler les dates. Voici un exemple pour convertir une chaîne de caractères en timestamp et formater le timestamp selon nos besoins :

```PHP
dateString=’2023-09-15’;dateString = '2023-09-15';
dateString=’2023-09-15’;timestamp = strtotime(dateString);dateString);
dateString);dateFormatee = date('d/m/Y', timestamp);echotimestamp);
echo timestamp);echodateFormatee;
```

Résultat :

```
15/09/2023
```

4. Comparer les dates

PHP nous permet de comparer les dates pour déterminer si une date est antérieure, postérieure ou égale à une autre. Voici un exemple pour comparer deux dates :

```PHP
date1=’2023-09-15’;date1 = '2023-09-15';
date1=’2023-09-15’;date2 = '2023-09-20';

if (date1<date1 < date1<date2) {
    echo "La date 1 est antérieure à la date 2";
} elseif (date1>date1 > date1>date2) {
    echo "La date 1 est postérieure à la date 2";
} else {
    echo "Les deux dates sont égales";
}
```

Résultat :

```BASH
La date 1 est antérieure à la date 2
```

5. Ajouter ou soustraire des intervalles de temps

Parfois, nous devons ajouter ou soustraire des intervalles de temps à une date donnée. Voici un exemple pour ajouter un jour à une date :

```PHP
date=’2023-09-15’;date = '2023-09-15';
date=’2023-09-15’;nouvelleDate = date('Y-m-d', strtotime('+1 day',
strtotime(date)));echodate)));
echo date)));echonouvelleDate;
```

Résultat :

```YAML
2023-09-16
```

6. Gérer les fuseaux horaires

Les fuseaux horaires sont importants lorsqu'il s'agit de travailler avec des dates. Voici un exemple pour définir le fuseau horaire par défaut et afficher la date dans ce fuseau horaire :

```PHP
date_default_timezone_set('Europe/Paris');
dateActuelle=date(’Y-m-dH:i:s’);echodateActuelle = date('Y-m-d H:i:s');
echo dateActuelle=date(’Y-m-dH:i:s’);echodateActuelle;
```

Résultat :

```yaml
2023-09-15 14:30:45
```

1.13 Exercices complémentaires

Advent of Code

https://adventofcode.com/

Exercism

1. opérateurs arithmétiques et boucles
    1. https://exercism.org/tracks/php/exercises/ordinal-number/
    2. https://exercism.org/tracks/php/exercises/sum-of-multiples
2. Chaines de caractères
    1. https://exercism.org/tracks/php/exercises/reverse-string
    2. https://exercism.org/tracks/php/exercises/binary
    3. https://exercism.org/tracks/php/exercises/rna-transcription
    4. https://exercism.org/tracks/php/exercises/atbash-cipher
3. Tableaux
    1. https://exercism.org/tracks/php/exercises/nucleotide-count
    2. https://exercism.org/tracks/php/exercises/word-count/edit
    3. https://exercism.org/tracks/php/exercises/flatten-array
    4. https://exercism.org/tracks/php/exercises/difference-of-squares
    5. https://exercism.org/tracks/php/exercises/accumulate
4. Chaines et tableaux
    1. https://exercism.org/tracks/php/exercises/hamming

À quoi sert ce programme ?

Dites pour chaque programme ce qu'il fait et à quoi il sert.

1.

```PHP
function process_data(a,a, a,b) {
        c=c = c=a + b;returnb;
        return b;returnc;
    }

    firstval=5;first_val = 5;
    firstval=5;second_val = 10;

    result=processdata(result = process_data(result=processdata(first_val,
secondval);echosecond_val);
    echo secondval);echoresult;
```

2.

```PHP
function calculate_values(input_array) {
    $output_array = [];
    $counter = 0;
    foreach($input_array as $value) {
        $output_array[$counter] = $value * 2;
        $counter++;
    }
    return $output_array;
}

function organize_data(input_array) {
    sort(inputarray);returninput_array);
    return inputarray);returninput_array;
}

originaldata=[5,2,8,1,9];original_data = [5, 2, 8, 1, 9];
originaldata=[5,2,8,1,9];processed_data = calculate_values(originaldata);original_data);
originaldata);organized_data = organize_data(processeddata);printr(processed_data);

print_r(processeddata);printr(organized_data);
```

3.

```PHP
function a(b) {
    $c = count($b);
    for ($d = 0; $d < $c / 2; $d++) {
        $e = $b[$d];
        $b[$d] = $b[$c - $d - 1];
        $b[$c - $d - 1] = $e;
    }
    return $b;
}
f = [1, 2, 3, 4, 5];
g=a(g = a(g=a(f);
print_r($g);
```

4.

```php
function a(b) {
    $c = str_split($b);
    $d = array_count_values($c);
    arsort($d);
    return key($d);
}

e = "hello world";
f=a(f = a(f=a(e);
echo $f;
```

5.

```php
function a(b) {
    $c = 0;
    foreach ($b as $d) {
        $c += $d;
    }
    return $c;
}

e = [1, 2, 3, 4, 5];
f=a(f = a(f=a(e);
echo $f;
```

6.

```php
function a(b) {
    $c = [];
    $d = explode(" ", $b);
    foreach ($d as $e) {
        if (strlen($e) > 3) {
            $f = str_split($e);
            $f[3] = "*";
            $g = implode("", $f);
            $c[] = $g;
        } else {
            $c[] = $e;
        }
    }
    return implode(" ", $c);
}
h = "Hello my name is John";
i=a(i = a(i=a(h);
echo $i;
```

7.

```php
function a(b) {
    for ($i = 0; $i < count($b); $i++) {
        $temp = $b[$i];
        $j = $i-1;
        while($j ⩾ 0 && $b[$j] > $temp) {
            $b[$j+1] = $b[$j];
            $j--;
        }
        $b[$j+1] = $temp;
    }
    return $b;
}

c = [3, 1, 4, 1, 5, 9, 2, 6, 5];
d=a(d = a(d=a(c);
print_r($d);
```

8.

```php
function a(b) {
    for ($i = 0; $i < count($b) - 1; $i++) {
        $min = $i;
        for ($j = $i + 1; $j < count($b); $j++) {
            if ($b[$j] < $b[$min]) {
                $min = $j;
            }
        }
        if ($i ≠ $min) {
            $temp = $b[$i];
            $b[$i] = $b[$min];
            $b[$min] = $temp;
        }
    }
    return $b;
}

c = [4, 9, 5, 3, 9, 2, 2, 1];
d=a(d = a(d=a(c);
print_r($d);
```

1.14 Implémentation de Brainfuck en PHP

Introduction

Les langages ésotériques sont des langages de programmation uniques qui repoussent les limites de la conception informatique. Ils sont souvent créés pour explorer des concepts théoriques ou simplement par défi intellectuel. Parmi ces langages, Brainfuck se distingue par son minimalisme extrême.

L'étude et l'implémentation d'un langage comme Brainfuck nous offrent une opportunité unique de plonger dans les fondamentaux de la programmation. Cet exercice nous permet de mieux comprendre comment fonctionnent les langages de programmation à un niveau très bas, et comment les concepts de base comme la gestion de la mémoire et les structures de contrôle sont mis en œuvre.

Découverte de Brainfuck

Brainfuck a été créé en 1993 par Urban Müller. Son objectif était audacieux : concevoir un langage Turing-complet (capable de réaliser n'importe quel calcul informatique) avec un compilateur le plus petit possible. Le résultat est un langage d'une simplicité déroutante, ne comportant que huit commandes.

Malgré (ou peut-être grâce à) sa simplicité, Brainfuck est capable d'exécuter n'importe quel algorithme. C'est un excellent exemple de la puissance qui peut résider dans un système minimaliste bien conçu.

Syntaxe de Brainfuck

Brainfuck utilise seulement huit commandes, chacune représentée par un seul caractère :

- `>` : Déplace le pointeur d'une cellule vers la droite
- `<` : Déplace le pointeur d'une cellule vers la gauche
- `+` : Incrémente la valeur de la cellule pointée
- `-` : Décrémente la valeur de la cellule pointée
- `.` : Affiche le caractère ASCII correspondant à la valeur de la cellule pointée
- `,` : Lit un caractère depuis l'entrée et le stocke dans la cellule pointée
- `[` : Début d'une boucle. Si la valeur de la cellule pointée est zéro, saute à la commande suivant le `]` correspondant
- `]` : Fin d'une boucle. Retourne à la commande suivant le `[` correspondant si la valeur de la cellule pointée n'est pas zéro

Tout autre caractère est considéré comme un commentaire et est ignoré lors de l'exécution. Cette caractéristique permet d'ajouter des explications ou de structurer visuellement le code sans affecter son fonctionnement.

Le concept de pointeur dans Brainfuck

Le concept de pointeur est central dans Brainfuck. Pour bien le comprendre, imaginons la mémoire de Brainfuck comme un long ruban divisé en cellules, chacune pouvant contenir une valeur numérique.

Un pointeur, dans ce contexte, est comme un curseur qui se déplace le long de ce ruban. Il indique à tout moment quelle cellule est "active", c'est-à-dire quelle cellule sera affectée par les opérations `+`, `-`, `.` et `,`.

Les commandes `>` et `<` sont utilisées pour déplacer ce pointeur. C'est un peu comme si vous déplaciez votre doigt le long d'une règle graduée, pointant différentes marques à mesure que vous avancez ou reculez.

Ce concept de pointeur est fondamental non seulement pour Brainfuck, mais aussi pour de nombreux aspects de la programmation de bas niveau. Comprendre comment Brainfuck utilise son pointeur vous donnera une base solide pour appréhender des concepts plus avancés dans d'autres langages.

Fonctionnement de Brainfuck

Brainfuck fonctionne sur un modèle de mémoire remarquablement simple :

1. Un tableau de cellules (généralement 30 000), toutes initialisées à zéro au début du programme.
2. Un pointeur, qui commence sur la première cellule (celle la plus à gauche).
3. Une série d'instructions (le programme Brainfuck) qui sont exécutées séquentiellement.

Les instructions sont traitées une par une, de gauche à droite, sauf dans le cas des boucles. Les boucles (`[ ]`) répètent les instructions qu'elles contiennent tant que la valeur de la cellule pointée n'est pas zéro.

Ce modèle simple permet pourtant de réaliser des calculs complexes. C'est un excellent exemple de comment des règles simples peuvent conduire à des comportements sophistiqués, un principe que l'on retrouve dans de nombreux domaines de l'informatique et des mathématiques.

Exemples simples en Brainfuck

Pour mieux comprendre comment fonctionne Brainfuck, examinons quelques exemples simples :

1. Hello World en Brainfuck :

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>+⟶»+[<]←]>>.>---.+++++++..+++.>>.←.<.+++.------.--------.>>+.>++.
```

Ce programme semble incompréhensible au premier abord, mais il imprime effectivement "Hello World!" à l'écran. Il illustre comment même des tâches simples peuvent sembler complexes en Brainfuck.

2. Addition de deux nombres :

```
,>,[−<+>]<.
```

Ce programme lit deux caractères (leurs valeurs ASCII), les additionne, et affiche le résultat. C'est un excellent exemple de la puissance de Brainfuck malgré sa simplicité.

Prenez le temps d'analyser ces exemples. Essayez de suivre le déplacement du pointeur et les changements de valeurs dans les cellules. Cet exercice vous aidera à mieux comprendre le fonctionnement de Brainfuck.

Préparation de l'environnement PHP

Notre objectif est de créer un interpréteur Brainfuck en PHP. Nous allons créer un script PHP qui pourra lire et exécuter des programmes Brainfuck à partir de fichiers.

Voici comment nous allons procéder :

1. Créez un fichier nommé `brainfuck.php`.
2. Ce script prendra le chemin vers un fichier Brainfuck comme argument de ligne de commande.
3. Il lira le programme depuis le fichier, l'interprétera, et exécutera les instructions Brainfuck.

Les extensions couramment utilisées pour les fichiers Brainfuck sont :

- `.b`
- `.bf`
- `.brainfuck`

Pour exécuter notre interpréteur, nous utiliserons la ligne de commande comme ceci :

```bash
php brainfuck.php chemin/vers/votre/programme.bf
```

Cette commande lance notre script PHP et lui passe le chemin vers le fichier Brainfuck à exécuter.

Pour lire l'input de l'utilisateur (nécessaire pour la commande `,` de Brainfuck), vous pouvez utiliser cette fonction PHP :

```php
function readChar() {
    return fgetc(STDIN);
}
```

Cette fonction lit un seul caractère depuis l'entrée standard (clavier).

Pour l'output (commande `.` de Brainfuck), vous pouvez simplement utiliser `echo` ou `print`.

Structure de base de notre interpréteur

Voici les étapes principales que votre interpréteur devra suivre :

1. Initialiser la mémoire (un tableau de 30000 cellules, toutes à 0).
2. Initialiser le pointeur (à 0, pointant sur la première cellule).
3. Lire le programme Brainfuck depuis l'argument de ligne de commande.
4. Parcourir chaque caractère du programme :
   - Si c'est une commande Brainfuck, l'exécuter.
   - Sinon, l'ignorer (c'est un commentaire).
5. Gérer les entrées/sorties quand nécessaire.

Réfléchissez à comment vous pourriez implémenter chacune de ces étapes en PHP.

Implémentation pas à pas

Voici quelques indications pour commencer l'implémentation :

1. Utilisez un tableau PHP pour représenter la mémoire.
2. Utilisez une variable pour le pointeur.
3. Utilisez une boucle pour parcourir chaque caractère du programme Brainfuck.
4. Utilisez une structure de contrôle (comme switch ou if-else) pour déterminer quelle action effectuer pour chaque caractère.
5. Pour les boucles (`[` et `]`), vous devrez peut-être utiliser une pile ou compter les crochets.

Pensez à comment vous pourriez implémenter chacune des huit commandes Brainfuck. Quelles variables devrez-vous modifier ? Comment gérerez-vous les entrées/sorties ?

Gestion des erreurs basiques

Votre interpréteur devrait gérer quelques erreurs de base :

1. Vérifiez que le pointeur ne sort pas des limites de la mémoire.
2. Assurez-vous que les crochets (`[` et `]`) sont bien équilibrés.
3. Gérez le cas où l'utilisateur n'entre pas de programme Brainfuck en argument.

Réfléchissez à comment vous pourriez détecter et signaler ces erreurs.

Exemples d'utilisation et tests

Une fois votre interpréteur implémenté, testez-le avec différents programmes Brainfuck. Commencez par des programmes simples, puis passez à des programmes plus complexes.

Exemples de tests :

1. Un programme qui affiche "A" (Indice : le code ASCII de 'A' est 65)
2. Un programme qui lit un caractère et l'affiche
3. Le programme "Hello World!" que nous avons vu plus tôt
4. Un programme qui additionne deux nombres entrés par l'utilisateur

Exercices et défis

Voici quelques défis pour approfondir votre compréhension :

1. Implémentez l'interpréteur de base capable d'exécuter les huit commandes Brainfuck.

2. Ajoutez la gestion des erreurs mentionnée précédemment.
3. Créez un petit programme Brainfuck qui fait quelque chose d'intéressant (par exemple, un calculateur simple) et testez-le avec votre interpréteur.
4. Comme défi final, créez un visualiseur de mémoire simple qui affiche l'état de la mémoire après chaque instruction.

Conclusion

L'implémentation d'un interpréteur Brainfuck en PHP nous offre une opportunité unique d'explorer plusieurs concepts fondamentaux de la programmation :

1. Gestion de la mémoire : Nous avons vu comment un langage peut manipuler directement la mémoire.
2. Structures de contrôle : Les boucles en Brainfuck nous montrent une forme basique mais puissante de contrôle de flux.
3. Entrées/Sorties : Nous avons appris à gérer les interactions basiques avec l'utilisateur.
4. Interprétation de code : Nous avons créé un programme qui lit et exécute un autre langage, une base pour comprendre comment fonctionnent les interpréteurs et les compilateurs.

Ces connaissances sont précieuses et transférables. Elles vous aideront à mieux comprendre le fonctionnement interne des langages de programmation plus complexes que vous utiliserez dans votre carrière.

Rappelez-vous, la programmation est un voyage d'apprentissage continu. Chaque défi comme celui-ci vous rapproche d'une compréhension plus profonde de l'informatique. Continuez à explorer, à expérimenter et à apprendre !

Tests

xmastree.b

```CSS
>>>--------<,[<[>++++++++++<-]>>[<------»—<+],]++>>++<—[<++[+>]>+<<+++<]<
<[>>+[[>>+<<—]<<]>>>>[[<<+>.>—]>>]<.<<<+<<—]>>[<.>--]>.>>.
```

tictactoe.b

```
---->---->>>>-<<>->>>>>-->>>>>>>>>>>>>>>>>>>>+>>++++++++++[
  <<++[
    --<+<<+<<+>>>>[
      >[<-»+++>>[-]+++<<<+[<++>>+<--]]+>+++++[>>+++++++++«<-]
      >>++++.[-]>>+[<<<<+>>+»-]<<<<<<[>+<-]<<
    ]++++++++++.[-]>++
  ]-->>[-->[-]>]<<[
    >>--[
      -[
        -[
          -----[>+>+++++++<<+]--»-.----->,[<->-]<[[<]+[->»]<-]<[<<,[-]]>>>>
        ]>
      ]<[
        >-[+<+++]+<+++[+[---->]+<<<<<<[>>]<[-]]
        >[<+[---->]++[<]<[>]>[[>]+>++++++++++«<-[<]]]>[>>>>]
      ]<[
        -[[>+>+«<-]>[<+>-]++>+>>]<[<<++[-->>[-]]>[[-]>[<<+»-]>]]
      ]<[
        [[<<]-[>>]<+<-]>[-<+]<<[<<]-<[>[+>>]>[>]>[-]]
        >[[+>>]<-<<>>[>]+>>>]
      ]<[
        -[
          --[+<<<<--[+>[-]>[<<+>+>-]<<[>>+«<-]]++[>]]
          <<[>+>+«<-]>--[<+>-]++>>>
        ]<[<<<[-]+++>[-]>[<+>>>+«<-]+>>>]
      ]<[
        +[[<]<<[<<]-<-»+>[>>]>[>]<-]+[-<+]<++[[>+<-]++<[«<-»>+]<++]<
        <<<<<<      +> > >+> > >+[
        <<<                 ->+>+>+[
        <<<<<<<   +>->+> > >->->+[
        <<<<<          ->+>+> >+>+[
        <<<<             ->->+>->+[
        <<<<<<<<<+>-> >+> > >->+>+[
        <<<<<          -> >+> >->+[
        <<<<             +>->+> >+]]]]]]]
        +++[[>+<-]<+++]---»>[-[<->-]<++>>]++[[<->-]>>]>[>]
      ]<
    ]
  ]<
]
```

Ressources utiles

- https://en.wikipedia.org/wiki/Brainfuck
- https://brainfuck.org/
- https://minond.xyz/brainfuck/
- https://copy.sh/brainfuck/

CHAPITRE 2

Pourquoi la POO avant Laravel

Séance 1 · ~1 période

Slides du chapitre (PDF) PDF

Vous êtes ici pour apprendre Laravel. Pourtant, les deux premières semaines de ce cours ne parlent pas de Laravel. Elles parlent de programmation orientée objet, la POO. Ce chapitre explique pourquoi, et comment les six séances qui viennent vont se passer.

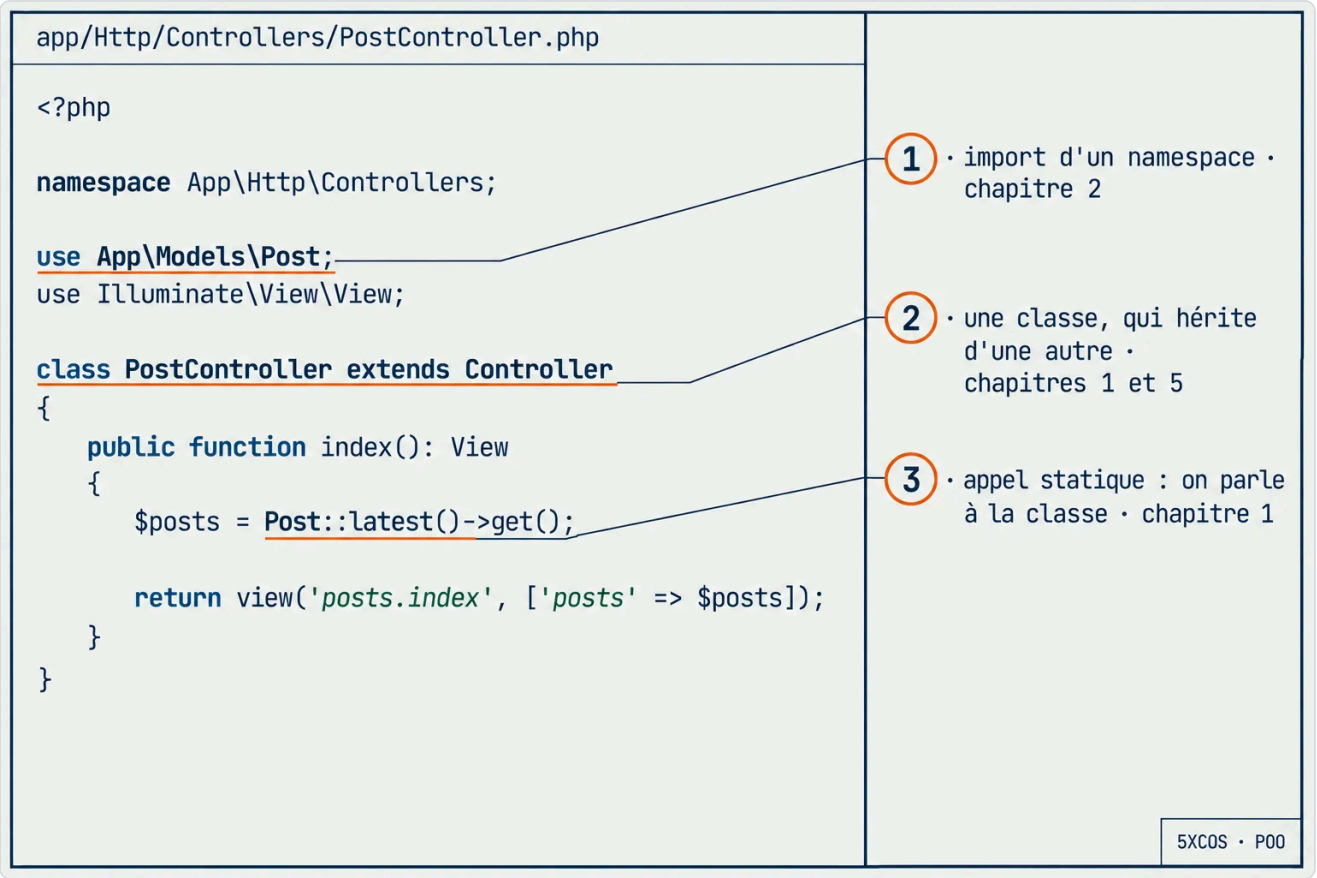
1. Vingt lignes de Laravel

Voici deux fichiers que vous écrirez en semaine 4. Ce sont de vrais fichiers Laravel, tels qu'ils sortent d'un projet neuf.

Le contrôleur, `app/Http/Controllers/PostController.php` :

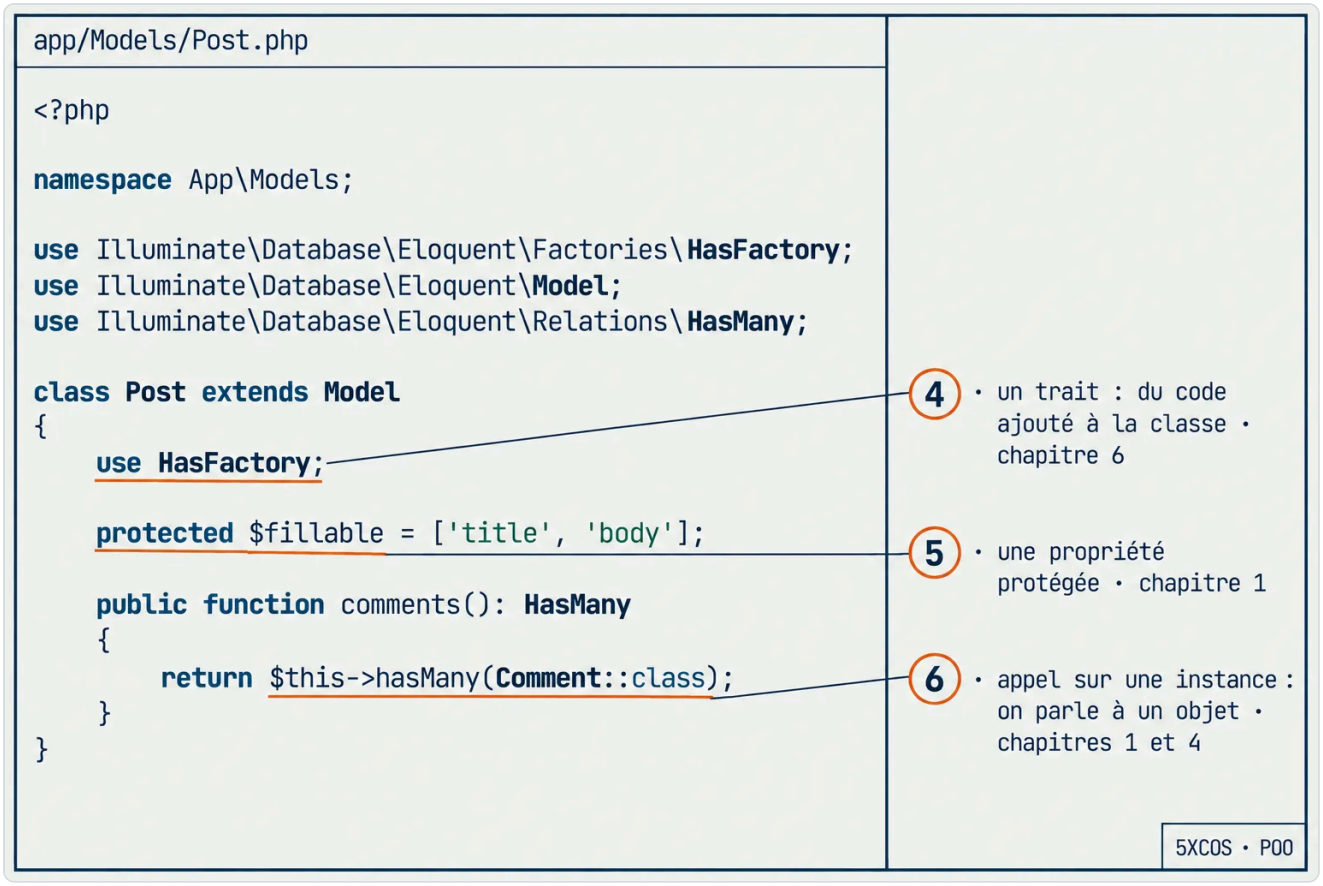
PHP

```php
<?php

namespace App\Http\Controllers;

use App\Models\Post;
use Illuminate\View\View;

class PostController extends Controller
{
    public function index(): View
    {
        posts=Post::latest()->get();returnview('posts.index',['posts'⇒posts =
Post::latest()→get();

        return view('posts.index', ['posts' ⇒ posts=Post::latest()
->get();returnview('posts.index',['posts'⇒posts]);
    }
}
```



Le modèle, `app/Models/Post.php` :

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Post extends Model
{
    use HasFactory;

    protected $fillable = ['title', 'body'];

    public function comments(): HasMany
    {
        return $this->hasMany(Comment::class);
    }
}
```



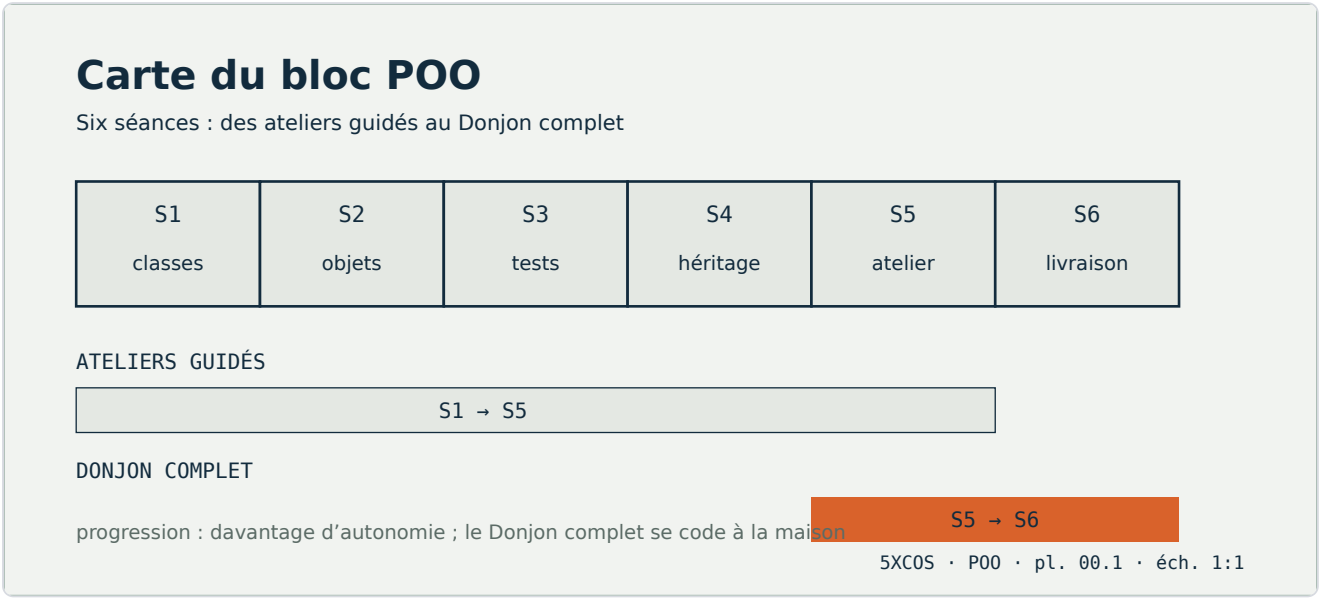
Ces vingt lignes ne calculent rien. Elles déclarent des objets et des liens entre eux. Pour les lire, il faut connaître huit mots.

Ce que vous voyez	Ce que c'est	Chapitre
`class PostController`	une **classe**, le moule dont on tire des objets	1
`extends Model`	l'**héritage** : Laravel a déjà écrit l'essentiel de la classe	6
`use HasFactory;` dans la classe	un **trait**, du code ajouté à la classe	7
`use App\Models\Post;` en haut du fichier	un **import** : même mot, autre chose	2
`Post::latest()`	un appel **statique**, adressé à la classe	1
`$this->hasMany(...)`	un appel sur une **instance**, adressé à un objet	1 et 5
`protected $fillable`	une propriété que la classe garde pour elle : l'**encapsulation**	1 et 3
`Comment::class`, `HasMany`	des **types** qui sont des classes	5 et 7
`new`	absent : alors qui fabrique ces objets ?	5 et 11

Si ce tableau vous paraît obscur, c'est normal. La documentation de Laravel suppose que tout cela est acquis. Le but des deux prochaines semaines est simple : rendre ces vingt lignes évidentes. Rien de plus.

2. Les six séances

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo



Tous les exemples et tous les katas racontent la même histoire : **le Donjon**, un petit jeu de rôle dans le terminal. Un héros, un dé, un sac, des monstres. Un seul décor, pour que votre attention aille aux concepts et pas au contexte. À la fin du bloc, au chapitre 10, vous assemblez toutes ces briques en un jeu complet, chez vous, sans code fourni.

3. Une séance type

Une séance dure cinq périodes. Elle se déroule presque toujours de la même façon.

1. **Un code qui sent mauvais.** Cinq minutes, en binôme. Un extrait de code fonctionne, mais quelque chose cloche. Vous cherchez quoi. Ce défaut est le sujet du jour.
2. **La théorie.** Une lecture courte, avec des exemples que vous pouvez lancer chez vous. Chaque chapitre tient en une heure de lecture.
3. **Le kata.** Un exercice court qu'on répète, comme en arts martiaux. Un dépôt GitHub contient des tests déjà écrits et des classes vides. Vous écrivez le code qui fait passer les tests au vert, niveau par niveau. Vous poussez, et GitHub relance les tests sur une machine neuve. Vert, vous continuez. Rouge, le message dit où regarder.
4. **La revue.** En fin de séance, on lit ensemble le code d'un binôme volontaire.

Quand vous bloquez, demandez. La première réponse sera une piste, pas la solution. Si la piste ne suffit pas après un vrai essai, vous aurez la solution, expliquée. Les premiers katas sont très guidés. Les suivants le sont de moins en moins, jusqu'au chapitre 10, où vous partez d'une feuille blanche.

4. L'IA

Vous pouvez utiliser ChatGPT, Claude ou Copilot pendant ce cours. Une seule règle : **vous devez pouvoir réécrire seul ce que vous rendez.** Une bonne façon de s'en servir est de lui demander d'expliquer un message d'erreur ou de critiquer votre code, puis de vérifier sa réponse en lançant les tests. Une mauvaise façon est de coller l'énoncé et de recopier la réponse. Au chapitre 10, que vous ferez seuls, notez les questions que vous lui avez posées et ce que vous en avez gardé.

5. Vos prérequis PHP

Huit questions, cinq minutes, aucune note. Répondez dans votre tête, puis retournez la carte. Trois erreurs ou plus : relisez le chapitre Introduction avant la séance 2, et dites-le au professeur.

Que renvoie `strlen('donjon')` et de quel type est cette valeur ?

`6`, un `int`. Une fonction renvoie une valeur. Si vous ne la stockez pas et ne l'affichez pas, elle est perdue.

Quelle est la différence entre `return` et `echo` dans une fonction ?

`echo` affiche tout de suite et ne rend rien. `return` rend une valeur à celui qui a appelé la fonction, et arrête la fonction.

Après $hp = 10;`hp -= 3;`, que vaut `$hp` ?

`7`. `-=` est un raccourci pour $hp =`hp - 3;`.

Quelle est la différence entre un tableau indexé et un tableau associatif ?

Le tableau indexé a des clés entières automatiques : `['épée', 'potion']`, on lit `$items[0]`. L'associatif a des clés que vous choisissez : `['name' ⇒ 'Arthur']`, on lit `$hero['name']`. En PHP, c'est le même type `array`.

Que fait `foreach (`$itemsas`key ⇒ $value)` de plus que `foreach (`$itemsas`value)` ?

Il donne la clé en plus de la valeur à chaque tour. On prend la forme courte tant qu'on n'a pas besoin de la clé.

Que vaut `'10' == 10` en PHP 8 ? Et `'10' === 10` ?

`true`, puis `false`. `==` compare après conversion, `===` compare aussi le type. Écrivez toujours `===`.

Que signifie le type `?int` dans `function findHero(string $name): ?int` ?

Un `int` ou `null`. Le `?` autorise l'absence de valeur. C'est la façon propre de dire « je n'ai rien trouvé ».

Quelle est la différence entre une variable qui vaut `null` et une variable qui n'existe pas ?

`null` est une valeur : la variable existe et elle est vide. Une variable qui n'existe pas déclenche `Warning: Undefined variable`. Les deux se testent avec `isset()` ou `??`.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo

À retenir

- Deux semaines de POO servent à lire Laravel, pas à faire de la théorie.
- Un seul décor pour les exemples et les katas : le Donjon.
- Chaque séance : un code qui cloche, une lecture courte, un kata testé, une revue.
- Les tests et la CI vous répondent tout de suite : vert, on avance ; rouge, le message dit où regarder.
- L'IA est permise, à condition de pouvoir refaire seul ce que vous rendez.

Exercices

1. **Faites le quiz** sans relire vos notes. Comptez vos erreurs.
2. **Relisez les vingt lignes de Laravel** et écrivez en français ce que fait, selon vous, `Post::latest()->get()`. Gardez votre réponse : on la ressortira au chapitre 11.
3. **Préparez votre machine** : un compte GitHub, et un terminal où `php --version` et `composer --version` répondent. Le chapitre 2 s'occupe du reste.

2.1 Objets et classes

Séance 1 · ~2 périodes

Slides du chapitre (PDF) PDF

À la fin de ce chapitre, vous serez capables de :

1. **définir** un objet comme la réunion d'un état et d'un comportement ;
2. **écrire** une classe PHP moderne : propriétés typées, constructeur avec promotion, `readonly`, `declare(strict_types=1)` ;
3. **expliquer** pourquoi une classe garde ses données pour elle et expose des actions comme `takeDamage()` ;
4. **distinguer** une méthode d'instance d'une méthode statique et écrire une fabrique `Dice::d6()`.

1. Le problème : un héros en tableau associatif

On veut un héros de jeu de rôle : un nom, des points de vie, une force. Il encaisse des coups, il se soigne, il peut mourir. Avec ce que vous savez déjà, cela donne un tableau et trois fonctions.

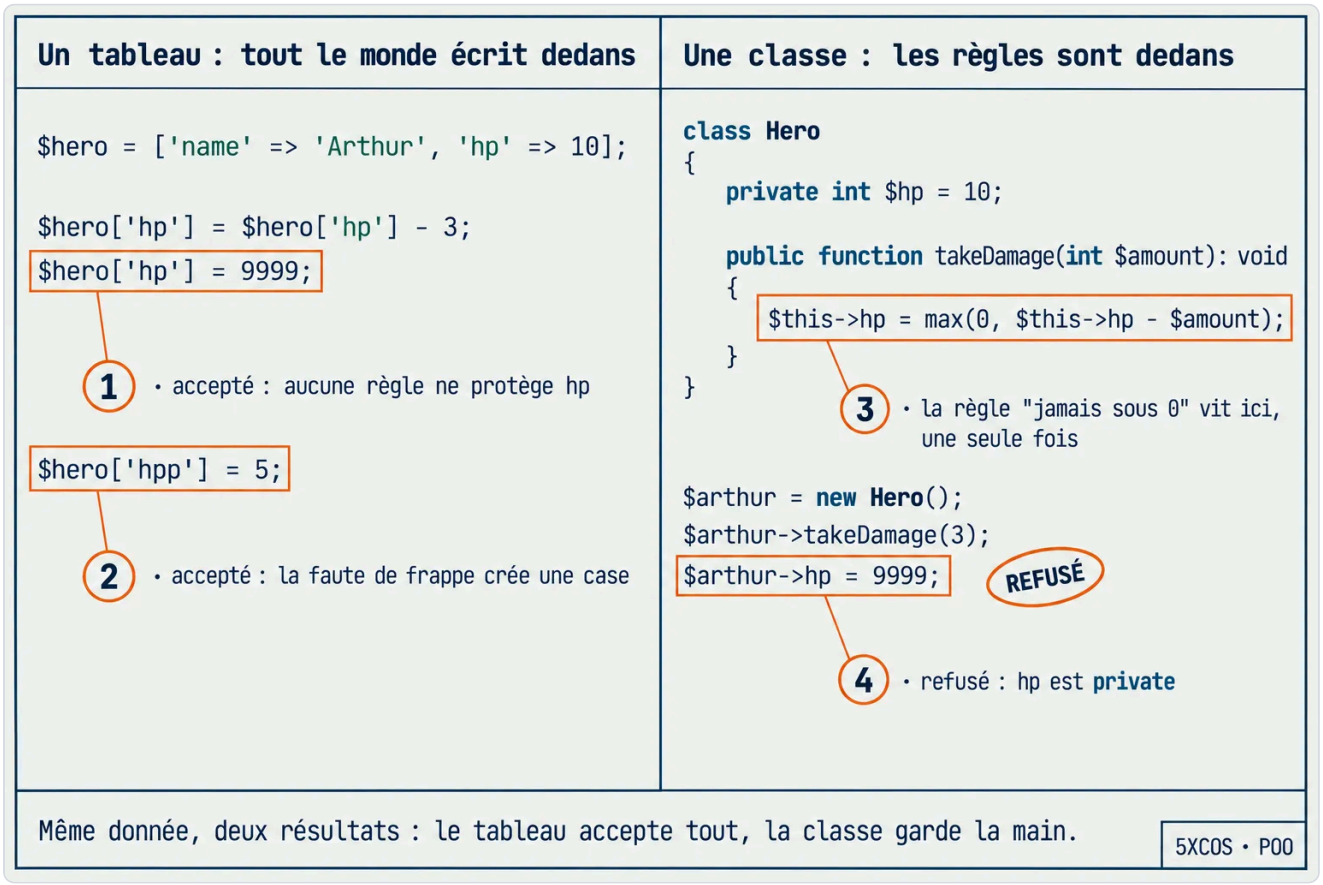
```php
<?php

declare(strict_types=1);

hero=[’name’⇒’Arthur’,’hp’⇒10,’maxHp’⇒10,’strength’⇒2];functiontakeDamage(arrayhero =
['name' ⇒ 'Arthur', 'hp' ⇒ 10, 'maxHp' ⇒ 10, 'strength' ⇒ 2];

function takeDamage(array hero=[’name’⇒’Arthur’,
’hp’⇒10,’maxHp’⇒10,’strength’⇒2];functiontakeDamage(arrayhero, int amount): array
{
    $hero['hp'] = max(0, $hero['hp'] - $amount);

    return $hero;
}

function heal(array hero, int amount): array
{
    $hero['hp'] = min($hero['maxHp'], $hero['hp'] + $amount);

    return $hero;
}

function describe(array hero): string
{
    return "{hero['name']} ({hero['hp']}/{hero['maxHp']} PV)";
}

hero = takeDamage(hero,3);hero, 3);
hero,3);hero = heal(hero,1);echodescribe(hero, 1);
echo describe(hero,1);echodescribe(hero), PHP_EOL;

hero[’hp’]=9999;hero['hp'] = 9999;
hero[’hp’]=9999;hero['maxHP'] = 12;
echo describe($hero), PHP_EOL;
```

```
Arthur (8/10 PV)
Arthur (9999/10 PV)
```

Le résultat est correct. Ce code a pourtant quatre défauts. Ce sont les quatre que la POO corrige.

1. **Les données et les fonctions qui les manipulent sont séparées.** Rien n'indique que `heal()` concerne le héros. Avec quarante fonctions dans le fichier, plus personne ne sait ce qui va avec quoi.
2. **Il faut réassigner à chaque appel** : $hero = takeDamage($hero, 3);$. Si vous oubliez `$hero =`, les points de vie ne changent pas et rien ne le signale.
3. **Rien n'est protégé.** La ligne `$hero['hp'] = 9999;` passe. Les règles du jeu sont : jamais sous 0, jamais au-dessus de `maxHp`. Elles ne tiennent que si tout le monde passe par ces trois fonctions.
4. **Rien n'est vérifié.** Une affectation à `$hero['maxHP']`, avec un `H` majuscule, ajoute une clé au tableau au lieu de modifier `maxHp`. Le bug apparaîtra plus tard, ailleurs.

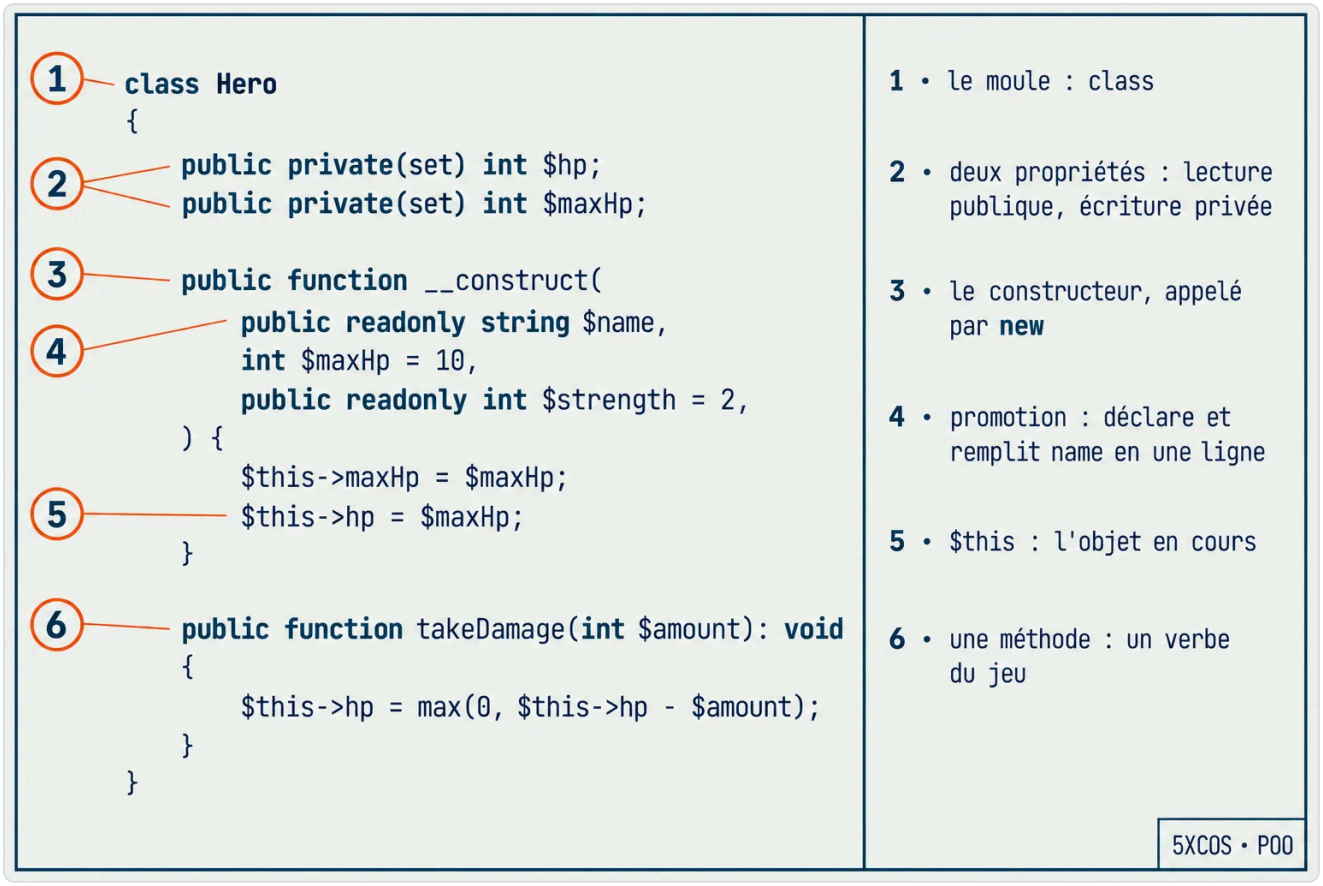
2. La même chose, en objet



Deux mots avant de lire le code qui suit. Une **propriété** est une variable qui appartient à l'objet : elle remplace ici une case du tableau. Une **méthode** est une fonction qui appartient à l'objet : elle remplace ici une des trois fonctions. Le reste n'est que de la syntaxe.

PHP

```php
<?php

declare(strict_types=1);

class Hero
{
    public private(set) int hp;publicprivate(set)inthp;
    public private(set) int hp;publicprivate(set)intmaxHp;

    public function __construct(
        public readonly string name,intname,
        int name,intmaxHp = 10,
        public readonly int strength = 2,
    ) {
        $this->maxHp = $maxHp;
        $this->hp = $maxHp;
    }

    public function takeDamage(int amount): void
    {
        this->hp=max(0,this->hp = max(0, this->hp=max(0,this->hp - amount);
    }

    public function heal(int amount): void
    {
        this->hp=min(this->hp = min(this->hp=min(this->maxHp, this->hp+this->hp +
this->hp+amount);
    }

    public function isAlive(): bool
    {
        return this->hp > 0;
    }

    public function __toString(): string
    {
        return sprintf('%s (%d/%d PV)', this->name, this->hp,this->hp, this->hp,this
->maxHp);
    }
}
```



On s'en sert comme ceci :

```php
hero=newHero(’Arthur’);hero = new Hero('Arthur');
hero=newHero(’Arthur’);hero→takeDamage(3);
hero->heal(1);echohero→heal(1);
echo hero->heal(1);echohero, PHP_EOL;

hero->takeDamage(50);echohero→takeDamage(50);
echo hero->takeDamage(50);echohero, PHP_EOL;
echo $hero→isAlive() ? 'vivant' : 'mort', PHP_EOL;
```

```
Arthur (8/10 PV)
Arthur (0/10 PV)
mort
```

Même résultat qu'avec le tableau. Mais `$hero` est maintenant un seul bloc : ses données et les méthodes qui les modifient sont au même endroit. On ne réassigne plus, car `$hero→takeDamage(3)` modifie le héros lui-même.

> NOTE
>
> **Un objet réunit un état et un comportement.** L'état, ce sont ses **propriétés** : `name`, `hp`, `maxHp`, `strength`. Le comportement, ce sont ses **méthodes** : `takeDamage()`, `heal()`, `isAlive()`. Tout le vocabulaire qui suit sert à décrire cette phrase.

3. Le vocabulaire

`class` : le moule, `new` : la pièce

`class Hero { … }` ne crée aucun héros. C'est un **moule** : la description des données et des méthodes que possède tout héros. `new Hero('Arthur')` fabrique une **instance**, c'est-à-dire un héros concret, avec ses propres valeurs, en mémoire.

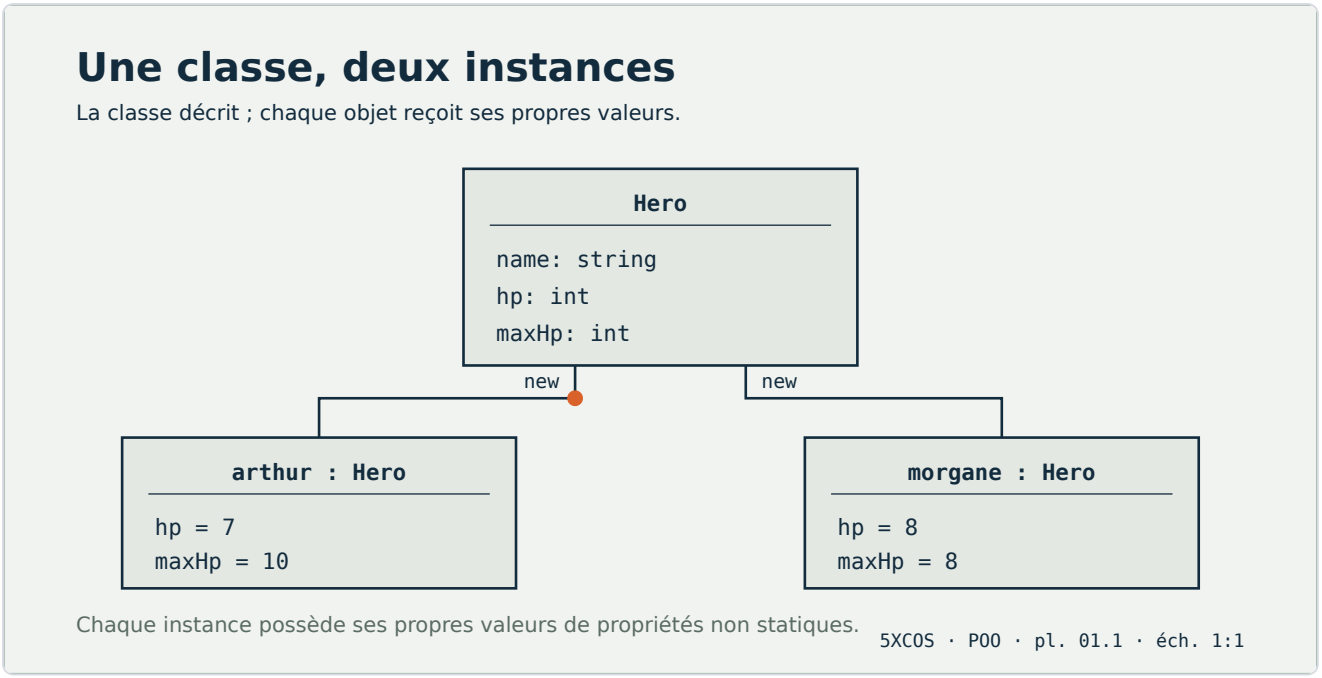
Avec un seul moule, vous fabriquez autant d'objets que vous voulez. Chacun a son propre état.

```php
arthur=newHero('Arthur');arthur = new Hero('Arthur');
arthur=newHero('Arthur');morgane = new Hero('Morgane', maxHp: 8, strength: 3);

arthur->takeDamage(3);echoarthur→takeDamage(3);

echo arthur->takeDamage(3);echoarthur, PHP_EOL;
echo $morgane, PHP_EOL;
```

```
Arthur (7/10 PV)
Morgane (8/8 PV)
```

Les points de vie de Morgane n'ont pas changé : chaque héros possède ses propres valeurs.

Au passage, `maxHp: 8` est un **argument nommé** : on écrit le nom du paramètre, puis sa valeur. Le constructeur de `Hero` attend trois paramètres. Le premier, `$name`, est obligatoire. Les deux autres ont une **valeur par défaut** : `$maxHp = 10` et `$strength = 2`. Si on ne les donne pas, PHP prend ces valeurs. Arthur n'a reçu que son nom : il a donc 10 points de vie et 2 de force. Pour Morgane, on veut 8 et 3, on nomme les deux. Et si on ne voulait changer que la force, `new Hero('Morgane', strength: 3)` suffit : `maxHp` garde sa valeur par défaut, sans qu'on ait à écrire `10` au milieu.



`$this` : l'objet en cours

Dans une méthode, `$this` désigne l'objet sur lequel la méthode a été appelée. Quand vous écrivez `$arthur→takeDamage(3)`, PHP exécute le corps de `takeDamage` avec `$this` qui vaut `$arthur`. C'est ce qui permet d'écrire la méthode une seule fois pour tous les héros du jeu.

Propriétés typées et `declare(strict_types=1)`

`private int $hp;` déclare une propriété qui n'accepte qu'un entier. La ligne `declare(strict_types=1)`, en tête de chaque fichier, dit à PHP de refuser les conversions silencieuses.

```php
declare(strict_types=1);

function takeDamage(int $amount): int
{
    return $amount;
}

takeDamage('3');
```

```
Fatal error: Uncaught TypeError: takeDamage(): Argument #1 ($amount)
must be of type int, string given
```

Sans cette ligne, PHP accepterait `'3'` et le convertirait en `3` sans rien dire. Le typage strict fait apparaître l'erreur là où elle est, pas cent lignes plus loin.

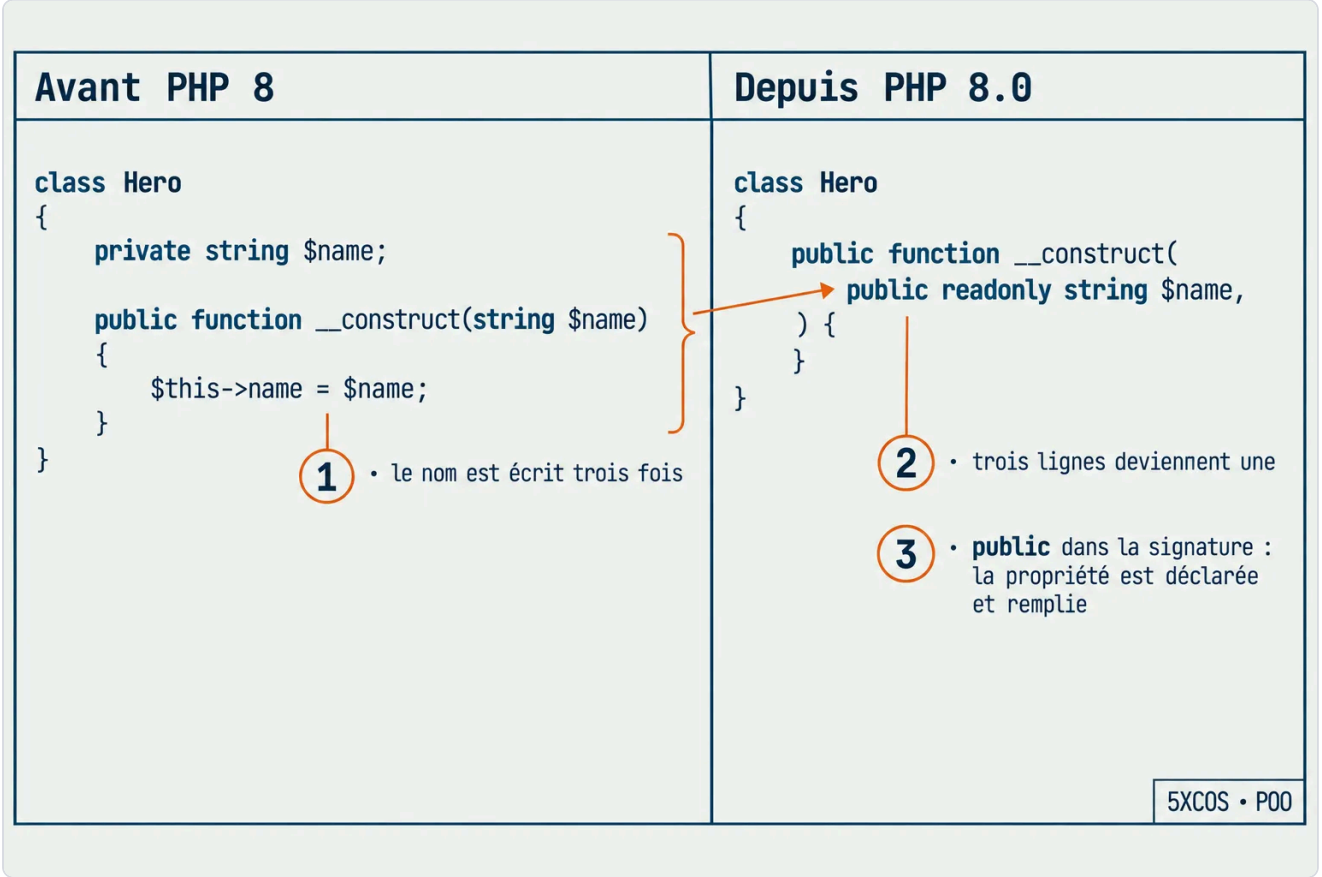
En PHP, les types ne sont pas obligatoires. Vous pouvez écrire `function takeDamage($amount)` et `private $hp;`, et ça tourne. Mais alors rien n'empêche de passer une chaîne, un tableau ou `null` à la place d'un entier. L'erreur ne se voit pas à l'appel : elle se voit plus tard, quand un calcul donne un

résultat absurde ou qu'un `null` fait planter une autre fonction, parfois dans un autre fichier. Un type, c'est une vérification gratuite que PHP fait pour vous à chaque appel, et une information pour celui qui lit : `int $amount` dit ce que la fonction attend sans qu'on ait à ouvrir son code. Votre éditeur s'en sert aussi pour vous prévenir avant même de lancer le programme. Dans ce cours, la règle est simple : un type sur chaque propriété, chaque paramètre et chaque valeur de retour, partout où c'est possible. Laravel est écrit comme ça.

Le constructeur, et la promotion

Le constructeur est la méthode appelée automatiquement par `new`. Il initialise les propriétés de l'objet. Avant PHP 8, il fallait écrire chaque propriété trois fois.

```php
// L'ancienne façon, celle que vous croiserez dans du code plus vieux.
class Hero
{
    private string name;

    public function __construct(string name)
    {
        this->name=this→name = this->name=name;
    }
}
```



> **NOTE**
>
> **Histoire de PHP.** Le langage s'est équipé par étapes. PHP 5.4, en 2012, ajoute la syntaxe courte des tableaux : `[]` remplace `array()`, que vous verrez encore dans du code ancien. PHP 7.0, en 2015, ajoute les types scalaires sur les paramètres et les retours, ainsi que `declare(strict_types=1)`. PHP 7.4, en 2019, ajoute les types sur les propriétés. PHP 8.0, en 2020, ajoute la promotion de constructeur et les arguments nommés. PHP 8.1, en 2021, ajoute `readonly`. Un fichier sans types ni promotion n'est pas du mauvais code : il est plus ancien que ces versions.

La **promotion de constructeur** fait les trois opérations en une.

```php
public function __construct(
    public readonly string name,intname,
    int name,intmaxHp = 10,
    public readonly int $strength = 2,
) {
    $this->maxHp = $maxHp;
    $this->hp = $maxHp;
}
```

Un **paramètre** est une valeur reçue par une méthode, le temps de son exécution. Une propriété, elle, reste dans l'objet. Un mot de visibilité devant un paramètre, comme `public` devant `$name`, crée la propriété correspondante et lui donne la valeur reçue. C'est le cas de `name` et de `strength`. Sans ce mot, `$maxHp` reste un simple paramètre, et le constructeur doit le recopier lui-même.

Le corps du constructeur sert à ce qui se calcule. Ici, `hp` reçoit la valeur de `maxHp` : le héros commence avec tous ses points de vie. C'est une règle du jeu, pas un paramètre. `hp` et `maxHp` sont l'affaire de la classe : tout le monde peut les lire, `$hero->hp`, mais seule la classe les écrit. C'est ce que dit `public private(set)`, expliqué juste après.

`readonly` : ce qui ne change jamais

Un héros ne change pas de nom en cours de partie. `readonly` l'écrit dans le code : la propriété reçoit sa valeur une fois, dans le constructeur, et ne peut plus être modifiée ensuite. Toute tentative est refusée par PHP, avec une erreur claire.

À quoi ça sert ? À rendre impossible une catégorie de bugs : la valeur qui change par accident, quelque part au milieu du programme, sans que personne ne l'ait voulu. Avec `readonly`, vous n'avez plus à vous demander « qui a modifié le nom ? » : personne, c'est garanti. Cela rend aussi la propriété publique sans risque. `name` est `public readonly` : tout le monde peut la lire avec `$arthur->name`, personne ne peut l'écrire. Réflexe à prendre : dès qu'une valeur est fixée à la création et ne doit plus bouger, elle est `readonly`. Le nombre de faces d'un dé, le nom d'un héros, la date de création d'une commande.

Visibilité : `public`, `private`, `protected`

Chaque propriété et chaque méthode porte un mot de **visibilité**. Il répond à une question : qui a le droit d'y toucher ?

« L'intérieur de la classe », c'est le code écrit entre les accolades de `class Hero { … }`, donc le corps de ses méthodes, là où on écrit `$this->hp`. « L'extérieur », c'est tout le reste : le script qui fait `new Hero('Arthur')`, une autre classe, un test.

```php
class Hero
{
    public string name='Arthur';privateintname = 'Arthur';
    private int name='Arthur';privateinthp = 10;

    public function takeDamage(int amount): void
    {
        $this->hp = max(0, $this->hp - $amount); // intérieur : autorisé
    }
}

arthur = new Hero();
echo arthur->name,PHPEOL;//exte´rieur,public:autorise´arthur->name, PHP_EOL; // extérieur, public : autorisé
arthur->name,PHPEOL;//exte´rieur,public:autorise´arthur->takeDamage(3);     // extérieur, méthode publique : autorisé
echo $arthur->hp;           // extérieur, private : refusé
```

```
Arthur
Fatal error: Uncaught Error: Cannot access private property Hero::$hp
```

- `public` : accessible de partout, à l'intérieur comme à l'extérieur. C'est le cas de `name` et de `takeDamage()`.
- `private` : accessible uniquement depuis l'intérieur de la classe. `hp` se lit et s'écrit dans les méthodes de `Hero`, nulle part ailleurs.
- `protected` : comme `private`, plus les classes qui héritent de celle-ci. On en reparle au chapitre 6, quand ces classes existeront.

Il manque un cas. L'affichage a besoin de lire `hp` : la barre de vie, le `Arthur (8/10 PV)`. Mais personne ne doit l'écrire. Depuis PHP 8.4, une propriété peut avoir deux visibilités, une pour la lecture et une pour l'écriture :

```php
public private(set) int $hp;
```

Se lit « publique en lecture, privée en écriture » : `$arthur->hp` fonctionne partout, `$arthur->hp = 9999` ne fonctionne que dans les méthodes de `Hero`. C'est la déclaration utilisée pour `hp` et `maxHp` dans la classe du chapitre. Le chapitre 3 y revient en détail.

Encapsulation : la classe garde la main sur ses données

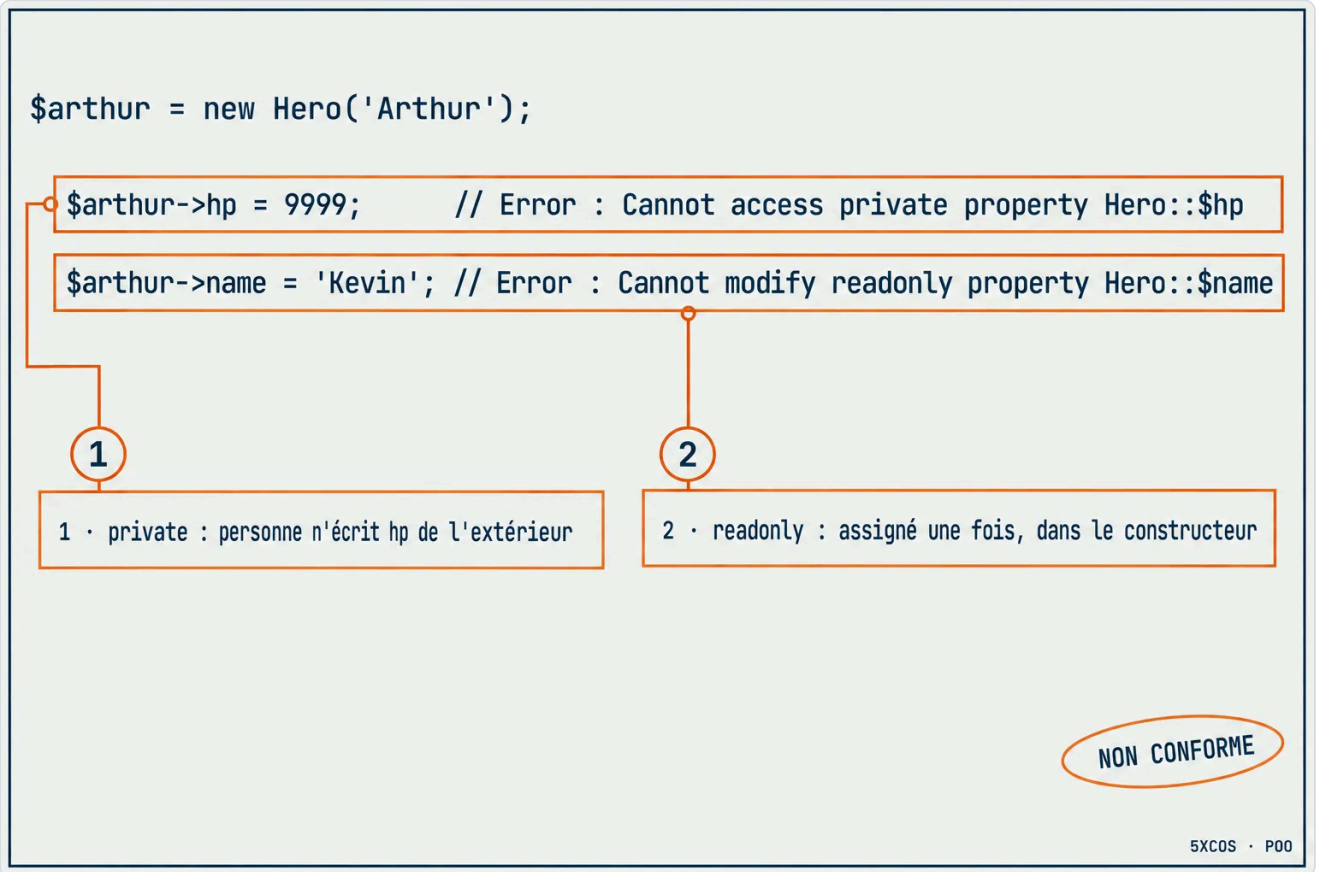
C'est le point le plus important du chapitre. Reprenez le tableau de la partie 1 : n'importe quelle ligne du programme pouvait écrire `$hero['hp'] = 9999;`, et la règle « jamais au-dessus de `maxHp` » ne tenait que si tout le monde pensait à passer par `heal()`. Dans un vrai projet, tout le monde n'y pense pas.

L'**encapsulation** règle ce problème. Les données sont `private`, donc inaccessibles de l'extérieur. La classe expose à la place des **actions** : `takeDamage()`, `heal()`, `isAlive()`. Chaque action contient sa règle, écrite une seule fois. Trois avantages concrets :

- **les règles sont respectées partout**, puisque le seul chemin vers `hp` passe par des méthodes qui les vérifient ;
- **une faute de frappe devient une erreur immédiate** : `$arthur→hpp` n'existe pas, PHP le dit tout de suite, au lieu de créer une case en silence comme avec le tableau ;
- **vous pouvez changer l'intérieur sans casser l'extérieur** : si demain les points de vie sont stockés autrement, seule la classe `Hero` change, pas les cinquante fichiers qui l'utilisent.

Voici deux écritures que PHP refuse maintenant.

```php
arthur->hp=9999;//proprie´te´prive´earthur→hp = 9999;      // propriété privée
arthur->hp=9999;//proprie´te´prive´earthur→name = 'Kevin';  // propriété readonly
```



Chaque essai doit être lancé seul, car la première erreur arrête le script. Voici les deux messages.

```
Fatal error: Uncaught Error: Cannot access private property Hero::$hp
```

```
Fatal error: Uncaught Error: Cannot modify readonly property Hero::$name
```

Le `9999` de la partie 1 est devenu impossible. La classe est le seul endroit où `hp` change : c'est cela, l'**encapsulation**.

ATTENTION

Vous croiserez souvent deux mots. Un **getter** est une méthode qui se contente de lire une propriété, comme `getHp()` ou `hp()`. Un **setter** est une méthode qui se contente de l'écrire, comme `setHp(int $hp)`. Le réflexe scolaire, hérité de Java, est d'en ajouter une paire pour chaque propriété. Ne le faites pas. Le getter est inutile depuis que `private(set)` ouvre la lecture sans ouvrir l'écriture. Le setter est dangereux : un `setHp()` sans vérification permettrait de redonner 9999 points de vie au héros, et on aurait recréé le tableau, avec plus de lignes.

```php
// Mauvais : la règle « jamais sous 0 » n'est plus nulle part.
public function setHp(int hp): void { $this→hp = $hp; }

// Bon : l'action du jeu, avec sa règle à l'intérieur.
public function takeDamage(int amount): void { this->hp=max(0,this→hp = max(0,
this->hp=max(0,this→hp - $amount); }
```

Ici, les méthodes publiques décrivent les actions du jeu : `takeDamage`, `heal`, plus tard `equip` et `drink`. En lisant vos méthodes publiques, on doit voir les règles du jeu, pas la liste de vos propriétés.

Lire une valeur de l'extérieur reste utile : l'affichage a besoin de la barre de vie. C'est pour cela que `hp` est `public private(set)` et pas `private`. Ce qu'on refuse, c'est l'écriture non contrôlée.

`__toString` : afficher un objet

`echo $hero` sur un objet est normalement une erreur. Déclarez `__toString(): string` et PHP l'appelle chaque fois que votre objet est utilisé comme du texte : `echo`, concaténation, interpolation dans une chaîne. C'est ce qui produit `Arthur (8/10 PV)` dans les sorties ci-dessus.

PHP réserve certains noms commençant par deux tirets bas, comme `__construct` et `__toString`, à des méthodes qu'il appelle automatiquement dans des situations précises. On les appelle les **méthodes magiques**. Ces deux-là suffisent pour l'instant.

`static` : parler à la classe plutôt qu'à un objet

Deuxième classe du Donjon, le dé.

```php
<?php

declare(strict_types=1);

class Dice
{
    public function __construct(
        public readonly int sides,
    ) {
    }

    public static function d6(): self
    {
        return new self(6);
    }

    public static function d20(): self
    {
        return new self(20);
    }

    public function roll(): int
    {
        return random_int(1, $this->sides);
    }
}

d6 = Dice::d6();
echo $d6->sides, PHP_EOL;
echo Dice::d20()->sides, PHP_EOL;
```

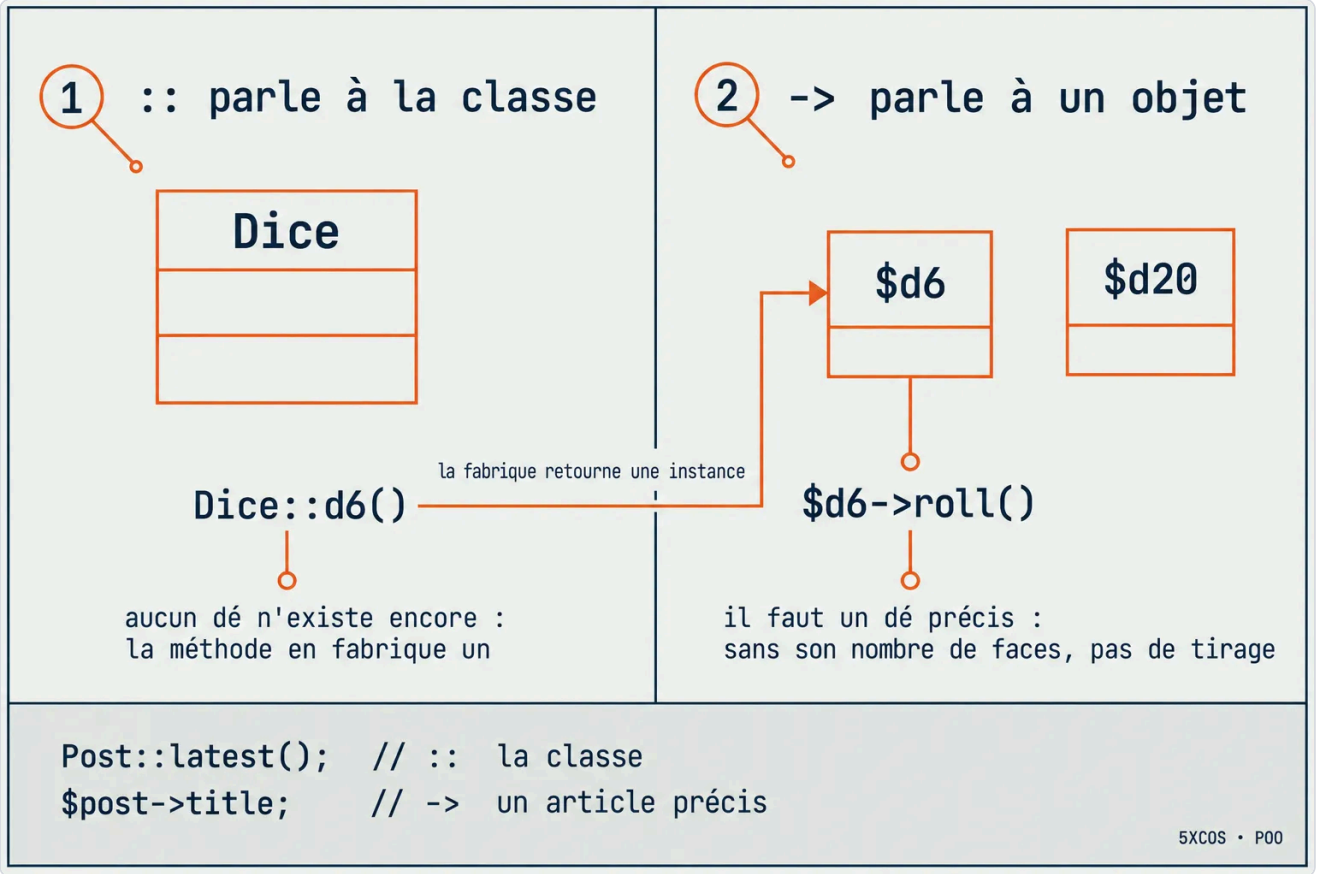
```
6
20
```

`roll()` a besoin d'un dé précis : sans le nombre de faces, impossible de tirer. C'est une **méthode d'instance**, on l'appelle avec la flèche `->` sur un objet. L'appel `$d6->roll()` renvoie un entier entre 1 et 6, différent à chaque exécution.

`d6()` n'a besoin d'aucun dé existant : elle en fabrique un. C'est une **méthode statique**, on l'appelle avec `::` sur la classe. Ici, `self` désigne `Dice` : écrire `new self(6)` revient à écrire `new Dice(6)`.

Pourquoi une fabrique plutôt que `new Dice(6)` partout ? Parce que `Dice::d6()` dit ce qu'on veut obtenir. Le d6 et le d20 sont des objets du jeu, pas deux nombres quelconques.

> **NOTE**
>
> Retenez la règle de reconnaissance : `::` **parle à la classe,** `->` **parle à un objet.** Vous retrouverez cette notation dans Laravel, avec `Post::latest()` d'un côté et `$post->title` de l'autre. Le mécanisme derrière `Post::latest()` est plus subtil que notre fabrique, mais la notation se lit de la même façon.

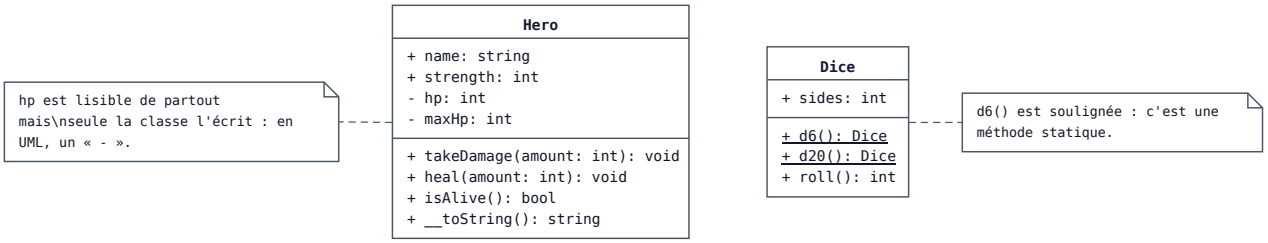


4. Pourquoi l'anglais dans le code ?

Les identifiants de ce cours sont en anglais, les commentaires et les explications en français. Laravel est écrit en anglais : ses classes `Model`, `Controller`, `User`, `Request`, ses méthodes `hasMany`, `latest`, `firstOrFail`, sa documentation et ses messages d'erreur. Il y a une raison plus concrète que le confort de lecture. Laravel devine beaucoup de choses à partir des noms : un modèle `Post` cherche une table `posts`, un modèle `Category` une table `categories`, une relation `comments()` une colonne `post_id`. Pour cela, il applique les règles du pluriel anglais. Un modèle `Article` deviendrait `articles`, ça passe par chance. Mais `Cheval` ne deviendra jamais `chevaux`, et `Categorie` donnera `categories` pendant que votre table s'appelle `categorie`. Résultat : des bugs entre le code et la base de données, difficiles à repérer. Une règle, pas de cas particulier : identifiants en anglais, commentaires et textes affichés en français.

5. `Hero` et `Dice` en UML

Un dessin se lit plus vite qu'un fichier. Voici les deux classes du chapitre en UML. Le chapitre 4 apprend à produire et à lire ces diagrammes.



À retenir

- Un objet réunit un **état**, ses propriétés, et un **comportement**, ses méthodes. La classe est le moule, `new` fabrique l'instance, `$this` désigne l'objet en cours.
- `declare(strict_types=1)`, types partout, promotion de constructeur, `readonly` : quatre habitudes qui transforment des bugs silencieux en erreurs immédiates.
- **Encapsulation** : personne n'écrit dans les propriétés de l'extérieur (`private`, ou `public private(set)` quand il faut pouvoir lire). La classe expose les actions du jeu comme `takeDamage()`. Les règles sont écrites une fois, au seul endroit où les données changent.
- `::` **parle à la classe,** `→` **parle à un objet.** Une fabrique statique comme `Dice::d6()` dit ce qu'on veut obtenir.
- Identifiants en anglais, commentaires en français : c'est la langue de Laravel.

Exercices

Les trois exercices tiennent dans un seul fichier. Créez `hero.php`, recopiez-y la classe `Hero` du chapitre, et lancez le tout avec `php hero.php`. Aucun outil supplémentaire n'est nécessaire : Composer et les tests arrivent au chapitre 2.

1. **Deux héros indépendants.** Créez Arthur avec ses valeurs par défaut et Morgane avec `maxHp: 8` et `strength: 3`. Infligez 4 dégâts à Arthur seulement, soignez Morgane de 5, puis affichez les deux. Vérifiez que les points de vie de l'un n'ont pas bougé quand l'autre a été touché, et expliquez pourquoi en une phrase.
2. **Une action de plus.** Ajoutez à `Hero` une méthode `attack(): int` qui renvoie la force du héros. Affichez le résultat pour Arthur et pour Morgane. Cette méthode deviendra intéressante au chapitre 6, quand le héros pourra équiper une arme.
3. **Deux fabriques de plus.** Ajoutez la classe `Dice` dans le même fichier, puis écrivez deux fabriques `Dice::d4()` et `Dice::d100()` sur le modèle de `d6()`. Lancez chaque dé dix fois et affichez les résultats. Puis répondez en deux phrases : `new Dice(4)` fait exactement la même chose que `Dice::d4()`, alors pourquoi garder les fabriques ? Indice : relisez la fin de la partie 3, et pensez à celui qui lira `Dice::d4()` dans six mois.

2.2 L'atelier : Composer, Pest et la CI

Séance 1 · ~2 périodes

Slides du chapitre (PDF) PDF

À la fin de ce chapitre, vous serez capables de :

1. **organiser** un projet PHP en un fichier par classe, avec `namespace` et `use` ;
2. **installer** des dépendances avec Composer et **expliquer** comment l'autoload PSR-4 retrouve vos classes ;
3. **lancer** une suite de tests Pest, la filtrer avec `--group` et `--filter`, et **lire** un test en échec ;
4. **lire** le résultat du workflow GitHub Actions fourni dans l'onglet Actions ;
5. **faire passer** le kata de niveau 1 du dépôt `poo-katas-26` et pousser un job vert.

Chaque kata, chaque atelier et le projet final se font dans un dépôt qui contient un `composer.json`, des tests et un fichier de workflow. D'où ce chapitre en premier.

1. Un fichier par classe

Au chapitre 1, `Hero` et `Dice` tenaient dans un seul fichier. Avec quinze classes, ce fichier devient difficile à parcourir, et deux personnes ne peuvent plus y travailler en même temps sans conflit.

La convention, en PHP, est donc : **un fichier par classe, et le fichier porte le nom de la classe**.

```
poo-katas-26/
├── composer.json
├── phpunit.xml
├── src/
│   ├── Dice.php          → class Dice
│   ├── Hero.php          → class Hero
│   ├── Inventory.php     → class Inventory
│   ├── Item.php          → class Item
│   └── …                 → une classe de plus à chaque niveau
└── tests/
    ├── Pest.php
    ├── Niveau1Test.php
    └── Niveau2Test.php
```

Dans `poo-katas-26`, les tests ne suivent pas les classes mais les niveaux : un fichier `NiveauNTest.php` par kata. Un niveau, un chapitre, un fichier.

2. `namespace` et `use` : le nom de famille des classes

Votre projet et Laravel peuvent chacun contenir une classe appelée `Collection`. PHP refuse deux classes du même nom, il faut donc pouvoir les distinguer. Le **namespace** est un préfixe qui s'en charge : un nom de famille pour vos classes.

`src/Dice.php` :

```php
<?php

declare(strict_types=1);

namespace Dungeon;

class Dice
{
    // …
}
```

Le nom complet de la classe est maintenant `Dungeon\Dice`. Ailleurs, on l'importe en haut du fichier avec `use`, et on l'utilise ensuite par son nom court.

```php
<?php

declare(strict_types=1);

require __DIR__ . '/vendor/autoload.php';   // fourni par Composer

use Dungeon\Dice;
use Dungeon\Hero;

hero=newHero('Arthur');hero = new Hero('Arthur');
hero=newHero('Arthur');hero->takeDamage(3);

echo $hero, PHP_EOL;
echo 'Le dé du donjon a ', Dice::d6()->sides, ' faces.', PHP_EOL;
```

```
Arthur (7/10 PV)
Le dé du donjon a 6 faces.
```

Cet import ne charge aucun fichier : il dit seulement à PHP quel nom complet se cache derrière le nom court `Dice`. Le chargement du fichier est le travail de l'autoload, à la section 4.

ATTENTION

Vous allez rencontrer deux usages de `use`, sans rapport entre eux :

- `use Dungeon\Dice;` **en haut du fichier**, avant la classe : c'est un import de namespace ;
- `use HasFactory;` **à l'intérieur d'une classe** : c'est un trait, du code copié dans la classe, vu au chapitre 7.

Pour les distinguer, regardez si le mot se trouve avant la classe ou à l'intérieur. C'est la confusion la plus fréquente à l'arrivée sur Laravel.

> **NOTE**
>
> **Histoire de PHP.** Les namespaces n'existent que depuis PHP 5.3, en 2009. Avant, on préfixait les noms de classes à la main, ce qui donnait des noms à rallonge comme `Zend_Db_Table_Abstract`. Composer arrive en 2012 ; avant lui, chaque fichier commençait par une pile de `require_once` qu'il fallait mettre dans le bon ordre. La norme PSR-4, qui lie le namespace au dossier, est publiée en 2013. Les trois ensemble expliquent pourquoi un projet PHP moderne n'a plus qu'une seule ligne de `require`.

3. Composer : le gestionnaire de dépendances

En première année, vous avez utilisé **npm** pour installer des bibliothèques JavaScript. Composer est son équivalent pour PHP. C'est un programme en ligne de commande, gratuit, qui fait trois choses : il télécharge les bibliothèques dont votre projet a besoin, il note leurs versions exactes, et il charge vos classes automatiquement, ce que la section suivante explique. Là où npm lit `package.json` et remplit `node_modules/`, Composer lit `composer.json` et remplit `vendor/`.

Les bibliothèques viennent de Packagist, l'annuaire public des paquets PHP, comme npmjs.com pour JavaScript. Chaque paquet y porte un nom en deux parties, `éditeur/paquet` : `pestphp/pest`, `laravel/framework`. Quand vous cherchez une bibliothèque PHP, c'est là que vous regardez, et `composer require éditeur/paquet` l'ajoute au projet en une commande. Laravel lui-même est un paquet Packagist, et un projet Laravel neuf en installe une centaine d'un coup.

Vous n'allez pas écrire votre propre outil de tests : Pest existe et il est gratuit. Composer installe Pest et les bibliothèques dont Pest a besoin, dans des versions compatibles avec le projet. Une **dépendance**, c'est cela : une bibliothèque écrite par quelqu'un d'autre et utilisée par votre projet.

Tout tient dans `composer.json`, à la racine du projet.

```json
{
    "name": "opmvpc/poo-katas-26",
    "require": {
        "php": "⩾8.4"
    },
    "require-dev": {
        "pestphp/pest": "^3"
    },
    "autoload": {
        "psr-4": {
            "Dungeon\\": "src/"
        }
    },
    "autoload-dev": {
        "psr-4": {
            "Tests\\": "tests/"
        }
    },
    "scripts": {
        "test": "pest"
    },
    "config": {
        "allow-plugins": {
            "pestphp/pest-plugin": true
        }
    }
}
```

Les clés qui comptent :

Clé	Ce qu'elle dit
`require`	Ce dont le projet a besoin pour tourner. Ici, PHP 8.4 minimum, parce que le cours utilise `private(set)` et les hooks, arrivés dans cette version : vérifiez la vôtre avec `php --version` avant de continuer.
`require-dev`	Ce dont vous avez besoin pour développer, ici les tests. Un serveur de production les exclut avec `composer install --no-dev`.
`^3`	La version 3 et ses mises à jour compatibles : 3.4 et 3.9 sont acceptées, 4.0 non.
`autoload`	Le lien entre vos namespaces et vos dossiers. Voir la section suivante.
`scripts`	Des raccourcis : `composer test` lancera `pest`.

Deux commandes, une différence qui compte :

```bash
composer install   # installe ce que dit composer.lock
composer update    # cherche des versions plus récentes et réécrit composer.lock
```

`composer.lock` note la version précise de chaque paquet installé. Il est **versionné avec le code**. C'est lui qui permet d'installer exactement les mêmes versions sur votre ordinateur, sur celui de votre binôme et sur GitHub. Pour les exercices fournis, utilisez toujours `composer install`.

Le dossier `vendor/`, lui, ne va **jamais** dans Git : c'est du code téléchargé, que `composer install` reconstruit à tout moment. Le `.gitignore` du dépôt s'en occupe.

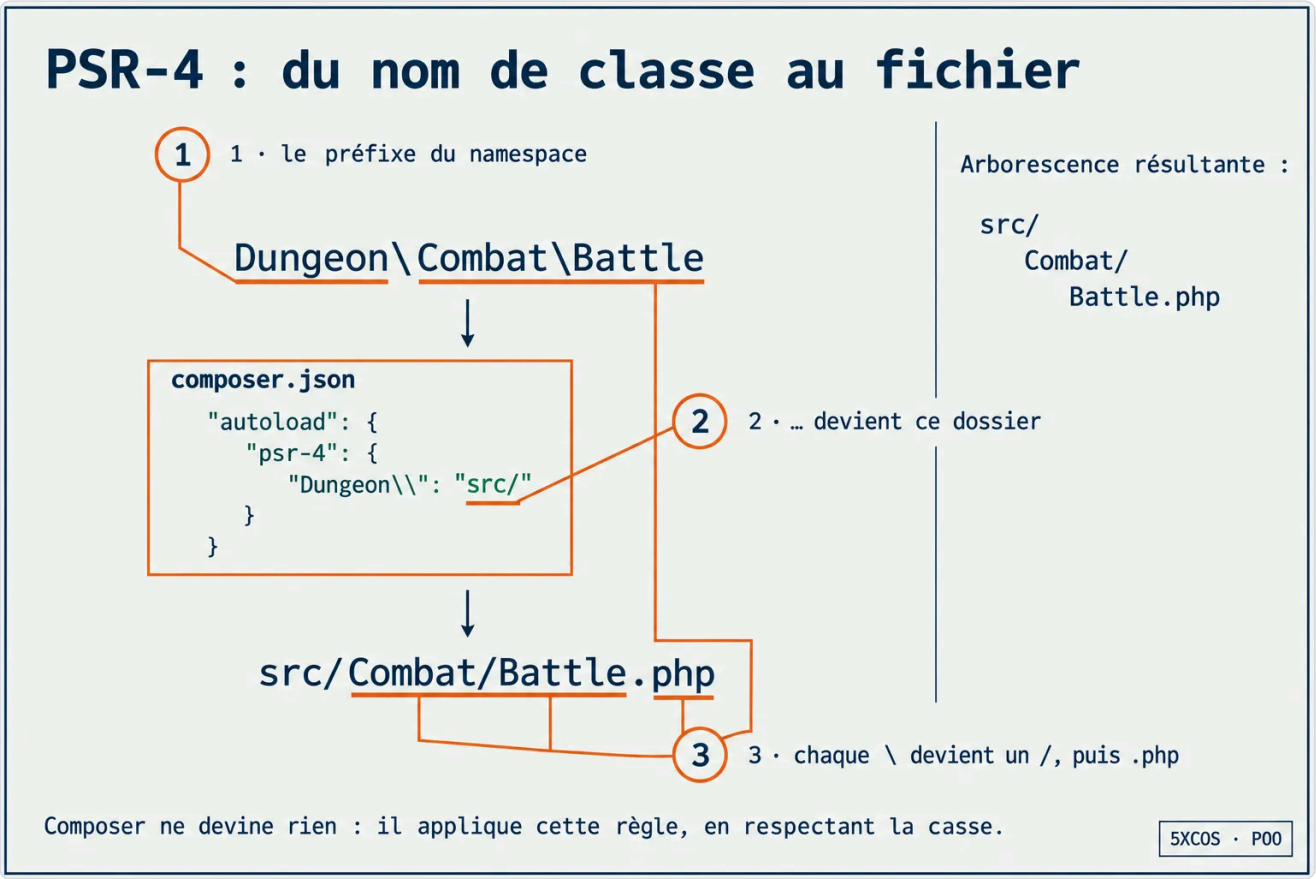
4. L'autoload PSR-4 : le namespace est le chemin

Une fois le chargement automatique configuré, vous n'écrivez plus de `require` pour vos propres classes. Une seule ligne, `require __DIR__ . '/vendor/autoload.php';`, et PHP va chercher le fichier d'une classe au moment où vous l'utilisez.

La règle est la norme **PSR-4**, et elle tient en une phrase : *à partir du préfixe configuré, le namespace est le chemin du dossier et le nom de la classe est le nom du fichier.*

Avec `"Dungeon\\": "src/"` dans `composer.json` :

Classe	Fichier attendu
`Dungeon\Dice`	`src/Dice.php`
`Dungeon\Hero`	`src/Hero.php`
`Dungeon\Combat\Battle`	`src/Combat/Battle.php`



Composer remplace `Dungeon\` par `src/`, puis chaque `\` restant par `/`. Il ajoute enfin `.php`. C'est tout : Composer ne fouille aucun dossier.

> **ASTUCE**
>
> Une classe « introuvable » alors que le fichier existe vient presque toujours d'une de ces trois causes :
>
> 1. le `namespace` du fichier ne correspond pas à son dossier ;
> 2. le nom de la classe ne correspond pas au nom du fichier. La majuscule compte : Linux la distingue, Windows non, d'où des erreurs qui n'apparaissent que sur GitHub ;
> 3. vous avez ajouté une correspondance dans `composer.json` sans relancer `composer dump-autoload`.

5. Pest : écrire et lire des tests

Un test est un petit programme qui exécute votre code et vérifie le résultat. Pest est l'outil qui fait tourner ces programmes : un **lanceur de tests**, ou *test runner*. Il cherche les fichiers de tests, les exécute un par un, compare ce qu'il obtient à ce qui était attendu, et affiche un rapport : une coche verte par test réussi, une croix rouge avec l'explication par test raté. Il n'a rien de magique. Sans lui, vous écririez un script avec des `if` et des `echo` pour vérifier chaque comportement à la main. Pest est le lanceur utilisé par Laravel ; la documentation est courte et bien faite.

Où vont les tests ? Dans le dossier `tests/`, à côté de `src/`. Un fichier de tests se termine par `Test.php` : `tests/Niveau1Test.php`, `tests/Niveau2Test.php`. C'est la convention que Pest suit pour les trouver, vous n'avez rien à déclarer. Le fichier `tests/Pest.php` contient la configuration, et vous n'y touchez pas. Dans `composer.json`, la clé `autoload-dev` donne au dossier `tests/` son propre namespace, `Tests\`, séparé de `Dungeon\` : le code des tests ne part pas en production.

Voici deux tests du kata de niveau 1.

```php
<?php

declare(strict_types=1);

use Dungeon\Dice;

test('un dé garde le nombre de faces qu\'on lui donne', function (): void {
    expect((new Dice(6))->sides)->toBe(6);
    expect((new Dice(20))->sides)->toBe(20);
})->group('niveau-1');

test('un lancer de dé reste entre 1 et le nombre de faces', function (): void {
    dice=newDice(6);for(dice = new Dice(6);

    for (dice=newDice(6);for(i = 0; i<100;i < 100; i<100;i++) {
        expect($dice->roll())->toBeGreaterThanOrEqual(1)->toBeLessThanOrEqual(6);
    }
})->group('niveau-1');
```

Trois éléments seulement :

- `test('…', function () { … })` : un nom en français, qui décrit le comportement attendu, et le code qui le vérifie.
- `expect(`$valeur$`)->toBe(attendu)` : une **assertion**, c'est-à-dire une vérification. Pest en propose une centaine. N'essayez pas de les retenir : consultez la documentation de Pest quand vous en avez besoin. Les tests des exercices vous montreront celles dont vous avez besoin.
- `->group('niveau-1')` : une étiquette, qui permet de ne lancer qu'une partie de la suite.

Le deuxième test vérifie cent tirages. C'est ce qu'on demande à un test sur du hasard : pas un tirage, cent. Notez sa limite : un `roll()` qui renverrait toujours 1 passerait quand même.

Lancer, et filtrer

Deux façons de lancer Pest, au choix.

```bash
composer test                          # toute la suite
composer test -- --group=niveau-1      # seulement les tests étiquetés niveau-1
composer test -- --filter=readonly     # seulement les tests dont le nom contient « readonly »
```

```bash
vendor/bin/pest                        # toute la suite
vendor/bin/pest --group=niveau-1
vendor/bin/pest --filter=readonly
```

`composer test` est un raccourci défini dans `composer.json` : il lance `vendor/bin/pest`. Les deux tirets seuls, `--`, disent à Composer « ce qui suit n'est pas pour toi, passe-le à Pest ». Sans eux, `--group` est ignoré et toute la suite tourne. Le raccourci a un défaut : la sortie perd ses couleurs. En appelant Pest directement, le rapport est en couleurs. Sous PowerShell, le chemin s'écrit `.\vendor\bin\pest`.

`--group` lance les tests d'un niveau, `--filter` sélectionne les tests dont le nom correspond au filtre. Les deux servent tout le temps : le premier pour le kata du jour, le second pour le test que vous êtes en train de réparer.

Lire un test en échec

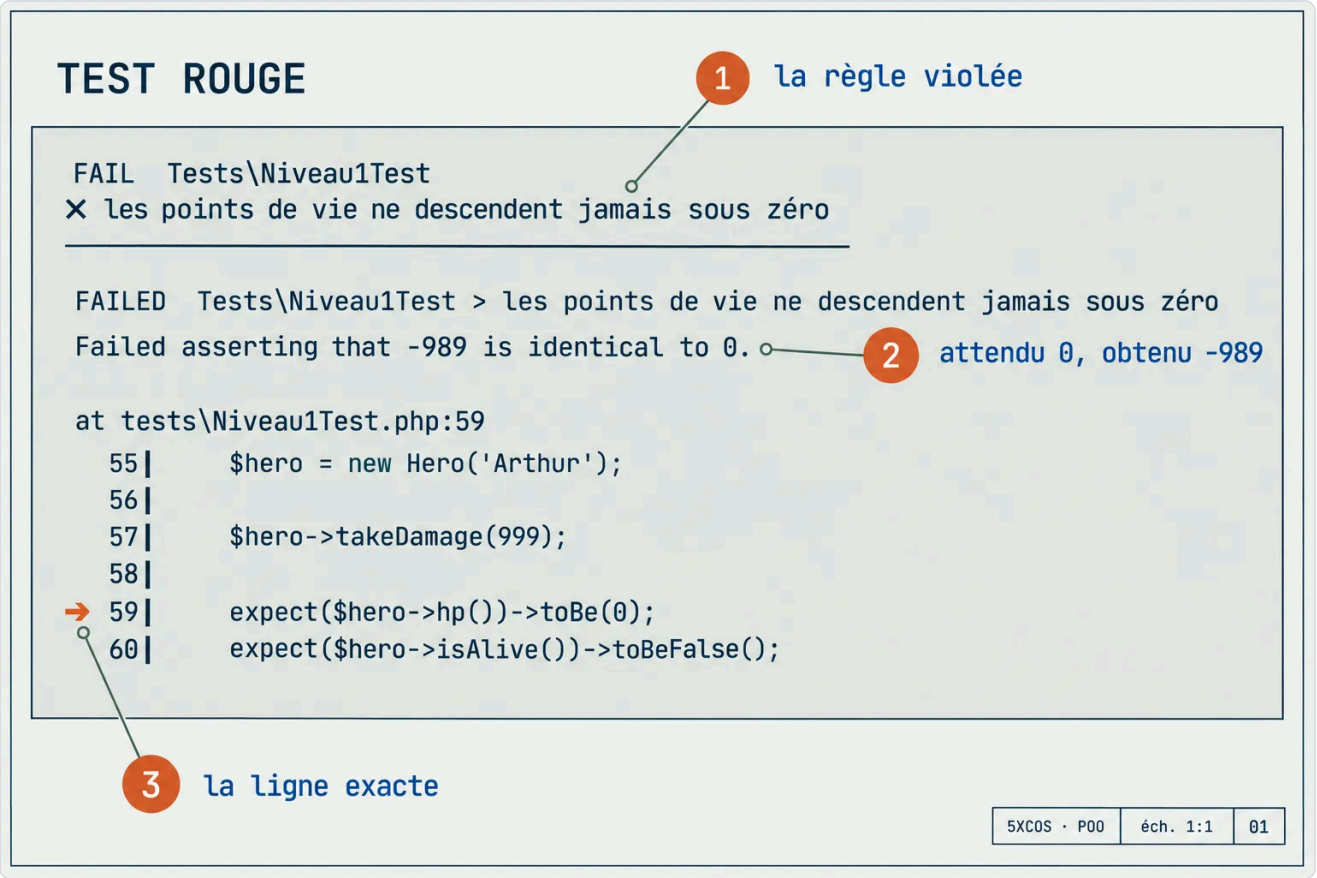
Un test rouge est une adresse. Voici ce que Pest affiche quand `takeDamage()` oublie de plafonner à zéro.

```
 FAIL  Tests\Niveau1Test
× les points de vie ne descendent jamais sous zéro                          0.01s
──────────────────────────────────────────────────────────────────────────────────

 FAILED  Tests\Niveau1Test > les points de vie ne descendent jamais sous zéro
Failed asserting that -989 is identical to 0.

at tests\Niveau1Test.php:59
   55 |     hero=newHero('Arthur');56 |57 |hero = new Hero('Arthur');
   56 |
   57 |     hero=newHero('Arthur');56 |57 |hero→takeDamage(999);
   58 |
➜  59 |     expect(hero->hp)->toBe(0);60 |expect(hero→hp)→toBe(0);
   60 |     expect(hero->hp)->toBe(0);60 |expect(hero→isAlive())→toBeFalse();
   61 | })→group('niveau-1');

1   tests\Niveau1Test.php:59

Tests:    1 failed, 9 passed (417 assertions)
```

Lisez-le dans cet ordre :

1. **le nom du test** : `les points de vie ne descendent jamais sous zéro`. Voilà la règle qui est violée.
2. **le message** : `Failed asserting that -989 is identical to 0`. Attendu `0`, obtenu `-989`.
3. **la flèche** `➜` : la ligne exacte de la vérification, avec le code autour.

`-989`, c'est `10 - 999` : la soustraction n'est pas bornée. Il faut empêcher les points de vie de descendre sous zéro, avec `max(0, …)`. Vous n'avez rien deviné : vous avez lu le message.

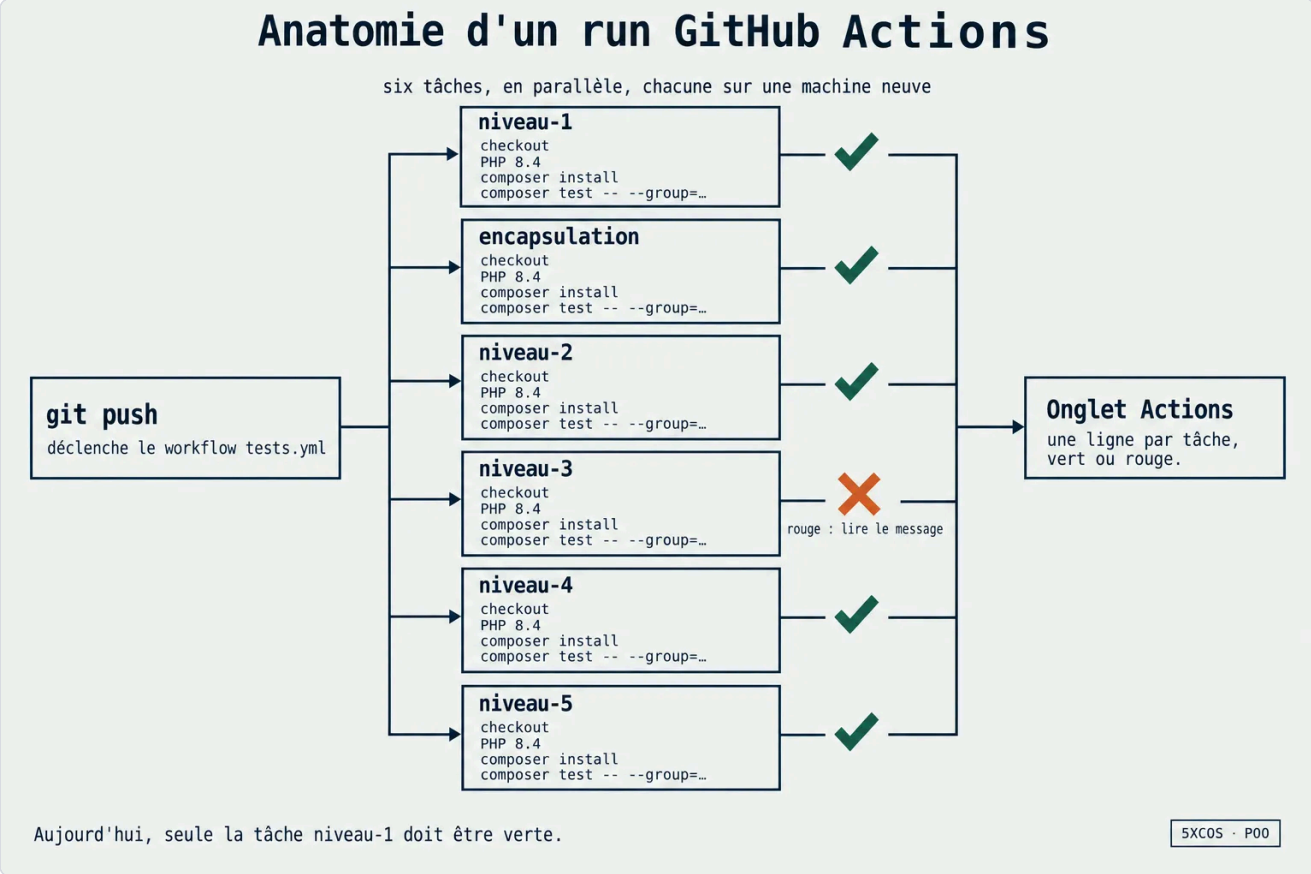


6. GitHub Actions : les tests relancés sur une machine neuve

Vos tests tournent chez vous. À chaque `git push`, GitHub peut les relancer sur une machine vierge et afficher une coche verte ou une croix rouge sur votre commit.

Un workflow est un fichier YAML dans `.github/workflows/`. Celui de `poo-katas-26` est fourni, vous n'avez rien à écrire. Une **tâche**, ou *job* dans l'interface, est une suite d'étapes exécutées sur une machine.

```yaml
name: Tests

on: [push, pull_request]

jobs:
  katas:
    name: {{ matrix.groupe }}
    runs-on: ubuntu-latest
    strategy:
      fail-fast: false        # si le niveau 1 casse, on veut quand même le verdict des
autres
      matrix:
        groupe: [niveau-1, encapsulation, niveau-2, niveau-3, niveau-4, niveau-5]
    steps:
      - uses: actions/checkout@v5            # cloner votre dépôt dans la machine

      - uses: shivammathur/setup-php@v2      # installer PHP
        with:
          php-version: "8.4"
          coverage: none

      - run: composer install --no-interaction --prefer-dist --no-progress

      - run: composer test -- --group={{ matrix.groupe }}
```



Trois repères :

- `on: [push, pull_request]` : ce qui déclenche l'exécution.

- `runs-on: ubuntu-latest` : une machine Ubuntu vierge, créée pour l'occasion et détruite à la fin. Rien de votre ordinateur n'y est. Ce que cela vérifie : le projet s'installe et se teste à partir du seul contenu du dépôt.
- `matrix: groupe: […]` : la tâche est lancée six fois en parallèle, une par groupe de tests. Dans l'onglet Actions, vous voyez six lignes, une par kata.

Une exécution complète n'est verte que si les six tâches passent. Tant que les autres groupes ne sont pas écrits, votre objectif est la coche verte de la tâche `niveau-1`, pas celle du dépôt entier. Les tests relancés sur GitHub arrivent deux minutes après le push et sont identiques pour tout le monde.

Le badge. Dans l'onglet Actions, votre workflow, menu « … », *Create status badge* : vous obtenez une ligne de Markdown à coller en haut de votre `README.md`. Votre dépôt affiche alors en permanence le résultat de la branche `main`.

7. La procédure, pas à pas

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/latelier-composer-pest-et-la-ci

Étape 5 : le premier rouge

Les classes de `src/` sont fournies avec leur signature, mais le code des méthodes reste à écrire. Pour l'instant, elles signalent « À implémenter ».

```php
public static function d6(): self
{
    throw new \LogicException('À implémenter');
}
```

En filtrant sur ce seul test, voici ce que Pest affiche.

```
 FAIL  Tests\Niveau1Test
× Dice::d6() et Dice::d20() fabriquent les dés classiques                    0.02s
─────────────────────────────────────────────────────────────────────────────────

 FAILED  Tests\Niveau1Test > Dice::d6() et Dice::d20()…          LogicException
À implémenter

at src\Dice.php:19
   15 |
   16 |     /** Fabrique statique : doit renvoyer un dé à 6 faces. */
   17 |     public static function d6(): self
   18 |     {
➜  19 |         throw new \LogicException('À implémenter');
   20 |     }
   21 |
   22 |     /** Fabrique statique : doit renvoyer un dé à 20 faces. */
   23 |     public static function d20(): self

1   src\Dice.php:19
2   tests\Niveau1Test.php:24

Tests:    1 failed (0 assertions)
```

Le message indique le fichier et la ligne où travailler.

Étape 7 : les premiers verts

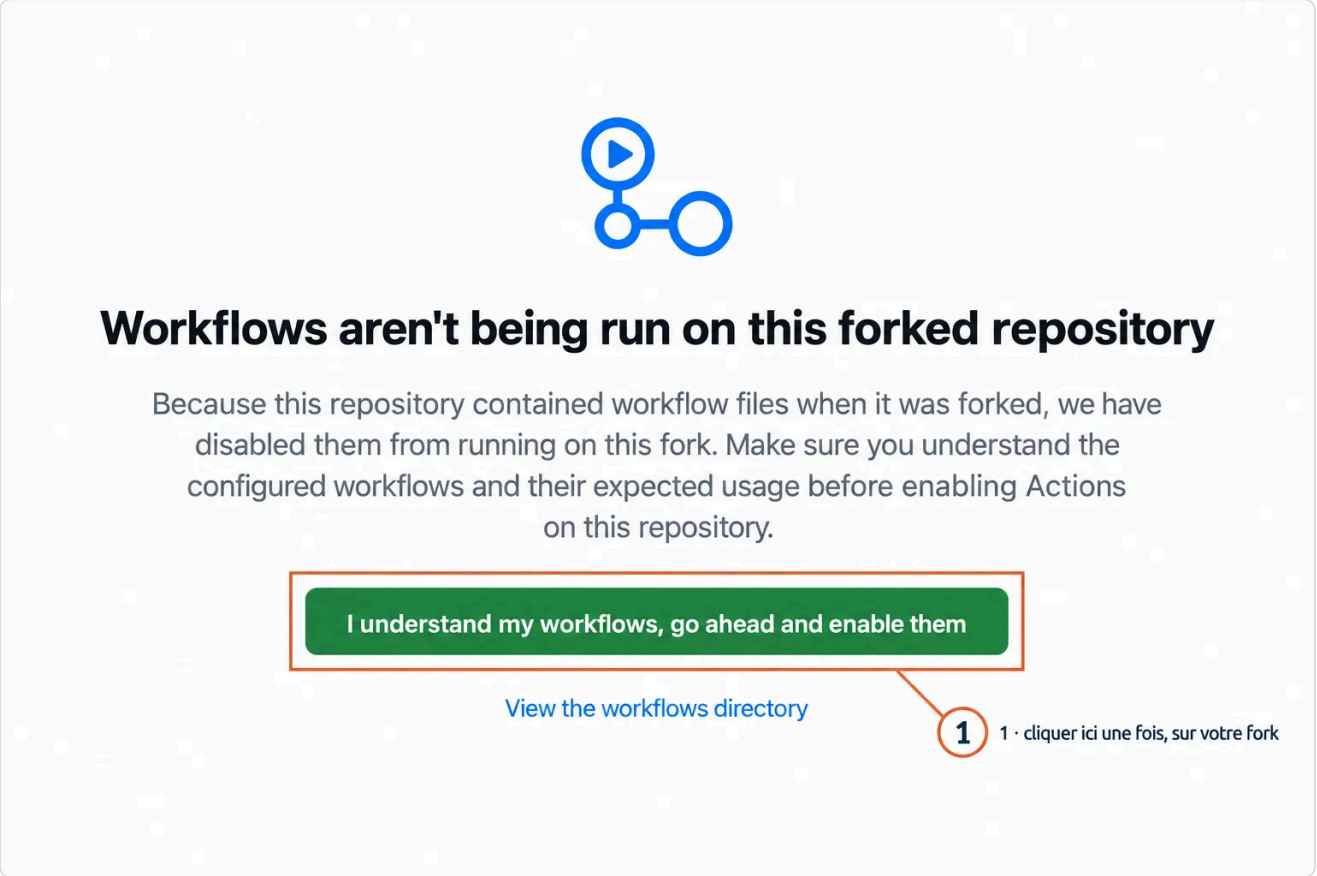
```php
public static function d6(): self
{
    return new self(6);
}

public static function d20(): self
{
    return new self(20);
}

public function roll(): int
{
    return random_int(1, $this->sides);
}
```

```
 FAIL  Tests\Niveau1Test
✓ un dé garde le nombre de faces qu'on lui donne                    0.01s
✓ un lancer de dé reste entre 1 et le nombre de faces               0.01s
✓ Dice::d6() et Dice::d20() fabriquent les dés classiques
× un héros démarre avec tous ses points de vie                      0.01s
× on peut choisir les points de vie et la force à la création
× takeDamage retire des points de vie
× les points de vie ne descendent jamais sous zéro
× heal ne dépasse jamais le maximum de points de vie
× un héros affiché donne "Arthur (7/10 PV)"
× le nom du héros est readonly : le réécrire lève une Error

Tests:    7 failed, 3 passed (405 assertions)
```

Trois tests verts sur dix : `Dice` est terminé, `Hero` reste à écrire. Un vert partiel est déjà une information utile.

Étape 9 : lire le résultat sur GitHub

Un détail d'abord. GitHub désactive les workflows sur un dépôt forké, par prudence : un fork pourrait contenir un workflow que vous n'avez pas lu. La première fois que vous ouvrez l'onglet **Actions** de votre fork, il affiche ce message. Cliquez sur le bouton vert, une seule fois. Tant que ce n'est pas fait, aucun push ne lance les tests.



Après le push, ouvrez l'onglet **Actions** de votre fork. Une exécution apparaît, d'abord en cours, puis son résultat s'affiche. Cliquez dessus : six tâches, une par groupe de tests.

Si la tâche `niveau-1` est rouge alors que les mêmes tests passent chez vous, la différence vient de l'environnement. Le plus souvent : un fichier oublié au moment du commit, ou une majuscule dans un nom de fichier, comme `src/dice.php` au lieu de `src/Dice.php`. Windows ne fait pas la différence, Ubuntu si.

> **ASTUCE**
>
> **Mettre à jour votre fork.** Si une erreur est trouvée dans les tests ou les classes fournies pendant la semaine, le dépôt d'origine sera corrigé. Votre fork ne suit pas ces corrections tout seul. Sur la page GitHub de votre fork, un bandeau indique alors « This branch is N commits behind » avec un bouton **Sync fork** : cliquez dessus, puis récupérez les changements sur votre machine avec `git pull`. Tant que la correction touche des fichiers que vous n'avez pas modifiés, comme les tests, tout se passe sans question. Si elle touche une classe de `src/` que vous avez déjà écrite, Git signale un conflit : ne corrigez rien vous-même, montrez-le, on le règle ensemble en deux minutes.

> **ASTUCE**
>
> **Dépanner Composer sous Windows.** Commencez toujours par `php --version`, `composer --version` et `php --ini`, puis partez du message exact.
>
> - `composer : le terme « composer » n'est pas reconnu` : Composer n'est pas dans le `PATH`. Fermez et rouvrez complètement votre terminal après l'installation ; si le message persiste, réinstallez depuis https://getcomposer.org/Composer-Setup.exe.
> - `your php version (8.1.x) does not satisfy that requirement` : votre PHP est trop ancien, le dépôt demande 8.4 minimum. Sous Windows, le plus simple est Herd, gratuit, qui installe PHP et Composer proprement.
> - `The openssl extension is required` ou `zip extension is missing` : ouvrez le `php.ini` indiqué par `php --ini` et retirez le `;` devant `extension=openssl`, `extension=zip` et `extension=mbstring`.
> - Utilisez PowerShell ou le terminal intégré de VS Code, pas `cmd.exe`. Ne travaillez pas dans un dossier synchronisé par OneDrive : la synchronisation bloque des fichiers et `composer install` échoue avec des messages incompréhensibles.
> - Après une modification de `composer.json`, rechargez l'autoload comme indiqué à la section 4. En dernier recours, supprimez `vendor/` et relancez `composer install`.
> - Rien de tout cela ? Copiez le message d'erreur complet et montrez-le, avec la commande qui l'a produit.

8. Kata niveau 1 : `Dice` et `Hero`

Le dépôt : https://github.com/opmvpc/poo-katas-26. Vous venez de le forker, de le cloner et d'y écrire `Dice`. Il reste `Hero`.

```bash
composer test -- --group=niveau-1
```

Les classes sont dans `src/`. Le code des méthodes reste à écrire ; pour l'instant, elles signalent « À implémenter ». Votre travail est de les remplir jusqu'à ce que le groupe `niveau-1` passe au vert. Les tests sont écrits, ne les modifiez pas : lisez-les, ils sont l'énoncé.

Ce qu'ils attendent, exactement :

`Dungeon\Dice`

- `new Dice(int $sides)` ;
- `roll(): int` renvoie un entier dans `[1, sides]`, bornes comprises (le test tire un grand nombre de fois) ;
- `Dice::d6()` et `Dice::d20()` : fabriques **statiques** qui renvoient un dé à 6 et à 20 faces.

`Dungeon\Hero`

- `new Hero(string` $name, int$`maxHp = 10, int $strength = 2)` ;
- `name` et `strength` sont des propriétés **publiques** : le test lit `$hero->name` et `$hero->strength` ;
- `hp` et `maxHp` se lisent de l'extérieur, `$hero->hp`, mais ne s'écrivent que dans la classe : elles sont déclarées **`public private(set) int`**, comme au chapitre 1, et le squelette les déclare déjà ainsi ;
- `hp` démarre à `maxHp` ;
- `takeDamage(3)` retire 3 points de vie, **sans jamais descendre sous 0** ;
- `heal()` remonte les points de vie, **sans jamais dépasser `maxHp`** ;
- `isAlive(): bool` : vrai tant que `hp > 0` ;
- `__toString()` renvoie exactement `"Arthur (7/10 PV)"`, respectez l'espace, les barres obliques et le `PV` ;
- `name` est **`readonly`** et `hp` est **`private(set)`** : deux tests vérifient qu'une tentative d'écriture de l'extérieur lève une `Error`.

> **ASTUCE**
>
> Le dépôt contient déjà les fichiers de tous les niveaux : `Monster.php`, `Rarity.php` et les autres ne vous concernent pas aujourd'hui. Le groupe `niveau-1` ne touche que `src/Dice.php` et `src/Hero.php`. Ignorez `HasHealth.php`, il sert au niveau 4.

Quand tout est vert, vous verrez ceci :

```
 PASS  Tests\Niveau1Test
✓ un dé garde le nombre de faces qu'on lui donne                    0.01s
✓ un lancer de dé reste entre 1 et le nombre de faces               0.01s
✓ Dice::d6() et Dice::d20() fabriquent les dés classiques
✓ un héros démarre avec tous ses points de vie                      0.01s
✓ on peut choisir les points de vie et la force à la création
✓ takeDamage retire des points de vie
✓ les points de vie ne descendent jamais sous zéro
✓ heal ne dépasse jamais le maximum de points de vie
✓ un héros affiché donne "Arthur (7/10 PV)"
✓ le nom du héros est readonly : le réécrire lève une Error
✓ les points de vie ne s'écrivent pas de l'extérieur : private(set) lève une Error

Tests:    11 passed (419 assertions)
```

Poussez, puis vérifiez que la tâche `niveau-1` est verte dans l'onglet Actions. Comptez 30 minutes. Si vous bloquez plus de dix minutes sur un test, montrez le test et votre code.

À retenir

- Un fichier par classe, un `namespace` qui suit le dossier : c'est PSR-4, et c'est ce qui permet à Composer de charger vos classes tout seul.
- `use` en haut du fichier importe un namespace ; `use` dans une classe colle un trait. Même mot, rien à voir.
- `composer install` suit `composer.lock`, qui est versionné avec le code ; `vendor/` ne va jamais dans Git.
- Un test en échec se lit dans l'ordre : le nom du test, le message attendu et obtenu, la ligne pointée par `➜`.
- Sur GitHub, les mêmes tests sont relancés sur une machine neuve à chaque push. Aujourd'hui, seule la tâche `niveau-1` doit être verte.

Exercices

1. **Ajoutez le badge** de statut dans le `README.md` de votre fork.
2. **Cassez volontairement** votre `Dice::d6()` en renvoyant un dé à 7 faces, poussez, puis lisez l'exécution rouge dans l'onglet Actions. Repérez la première ligne d'erreur utile. Remettez en état, poussez, revérifiez.
3. **Renommez** `src/Dice.php` en `src/dice.php` en local, relancez `composer test`. Que se passe-t-il chez vous, et que se passerait-il sur GitHub ? Expliquez en deux phrases, puis remettez la majuscule.

2.3 L'encapsulation

Séance 2 · ~1 période

Slides du chapitre (PDF) PDF

Depuis le chapitre 1, personne ne peut écrire `$hero->hp = 9999` : `hp` est `public private(set)`. Pour chaque propriété, trois questions. Qui lit, qui écrit, où vit la vérification.

À la fin de ce chapitre, vous serez capables de :

1. **expliquer** ce que l'encapsulation protège, et quels bugs elle rend impossibles ;
2. **choisir** pour chaque propriété entre `private`, `public readonly` et `public private(set)` ;
3. **attacher** une règle d'écriture à une propriété, et une valeur calculée à sa lecture ;
4. **refuser** un objet invalide dès le constructeur, avec une exception.

Le code qui sent mauvais

Cinq minutes, en binôme. La classe `Hero` du chapitre 1, et deux appels un peu tordus. Lisez, et répondez à une seule question : **qu'affichent les deux `echo` ?**

```php
class Hero
{
    public private(set) int hp;publicprivate(set)inthp;
    public private(set) int hp;publicprivate(set)intmaxHp;

    public function __construct(
        public readonly string name,intname,
        int name,intmaxHp = 10,
        public readonly int strength = 2,
    ) {
        $this->maxHp = $maxHp;
        $this->hp = $maxHp;
    }

    public function takeDamage(int amount): void
    {
        this->hp=max(0,this->hp = max(0, this->hp=max(0,this->hp - amount);
    }

    public function heal(int amount): void
    {
        this->hp=min(this->hp = min(this->hp=min(this->maxHp, this->hp+this->hp +
this->hp+amount);
    }

    public function __toString(): string
    {
        return sprintf('%s (%d/%d PV)', this->name,this->name, this->name,this->hp, this
->maxHp);
    }
}

arthur = new Hero('Arthur');
arthur->takeDamage(-5);echoarthur->takeDamage(-5);
echo arthur->takeDamage(-5);echoarthur, PHP_EOL;

morgane=newHero('Morgane');morgane = new Hero('Morgane');
morgane=newHero('Morgane');morgane->heal(-20);
echo $morgane, PHP_EOL;
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/encapsulation

1. Ce que la classe promet

`Hero` tient une promesse qui n'est écrite nulle part : **`hp` reste entre 0 et `maxHp`**. Le reste du programme y compte. L'affichage `Arthur (8/10 PV)` n'a de sens que si cette promesse tient. Une règle qui doit rester vraie pendant toute la vie d'un objet s'appelle un **invariant**.

Cette façon de faire s'appelle l'**encapsulation** : la classe cache ses données et garantit ses règles. Ailleurs dans le programme, vous écrivez `$arthur->takeDamage(3)` sans rien vérifier. Dans `Hero`, la méthode s'occupe des bornes.

- **La règle est écrite une seule fois.** La borne « jamais sous 0 » est dans `Hero`, pas dans les vingt endroits du jeu qui infligent des dégâts.
- **Un bug se cherche à un seul endroit.** Si `hp` vaut 9999, l'erreur est dans `Hero`. Avec le tableau du chapitre 1, elle pouvait être n'importe où.
- **Les méthodes publiques suffisent à comprendre la classe.** `takeDamage()`, `heal()`, `isAlive()` : on lit ce qu'un héros sait faire sans ouvrir le reste du programme.

Le code qui sent mauvais montre la limite du chapitre 1 : l'écriture est fermée, mais la vérification est éparpillée.

2. Choisir la visibilité

Vous connaissez `public private(set)` depuis le chapitre 1 : lecture publique, écriture réservée à la classe.

Déclaration	Qui lit	Qui écrit	Quand
`private int $turn`	la classe	la classe	un détail interne dont personne dehors n'a besoin
`public readonly string $name`	tout le monde	personne, après le constructeur	une valeur fixée à la création, comme le nom
`public private(set) int $hp`	tout le monde	la classe	une valeur qui change, mais seulement par les actions de la classe
`public int $hp`	tout le monde	tout le monde	jamais, dans ce cours

La troisième ligne, à l'essai :

```php
arthur=newHero('Arthur');echoarthur = new Hero('Arthur');
echo arthur=newHero('Arthur');echoarthur->hp, PHP_EOL;
$arthur->hp = 9999;
```

```
10
Fatal error: Uncaught Error: Cannot modify private(set) property Hero::$hp from global scope
```

La règle, dans l'ordre :

1. **Écrivez `private`.** Tout autre choix demande une raison. Un compteur de tours dans `Dungeon`, que seul le donjon utilise, reste `private int $turn = 0`.

2. **Ouvrez la lecture quand quelqu'un en a besoin.** L'affichage doit lire `hp` : `public private(set)`. Le nom ne change jamais : `public readonly`.
3. **N'ouvrez jamais l'écriture.** Quand le reste du programme veut modifier un héros, c'est qu'il y a une action du jeu derrière. Le nom de cette action contient un verbe : `takeDamage()`, `heal()`, `equip()`.

> **NOTE**
>
> `protected`, vu au chapitre 1, attend le chapitre 6 : il sert quand une classe hérite d'une autre, et ces classes ont rarement besoin d'écrire dans les propriétés de leur parent.

> **ATTENTION**
>
> PHP accepte `public private(set) readonly`, mais cette écriture n'ajoute rien à `readonly` : une propriété `readonly` refuse déjà toute réécriture, même depuis la classe. Choisissez l'un ou l'autre. La valeur change-t-elle pendant la vie de l'objet ? Non : `readonly`. Oui : `private(set)`.

3. Le hook `set` : la règle vit sur la propriété

Relisez `takeDamage()` et `heal()`. La règle « entre 0 et `maxHp` » est coupée en deux : la borne basse dans une méthode, la borne haute dans l'autre. Ajoutez demain une potion empoisonnée qui fait `$this->hp -= 3` : il faudra penser à réécrire `max(0, …)` une troisième fois.

Depuis PHP 8.4, une propriété peut porter du code. C'est un **hook** : du code exécuté à chaque lecture ou à chaque écriture de la propriété. Il y en a deux, `set` et `get`. Voici le `set` de `hp`.

```php
public private(set) int $hp = 0 {
    set => max(0, min($this->maxHp, $value));
}
```

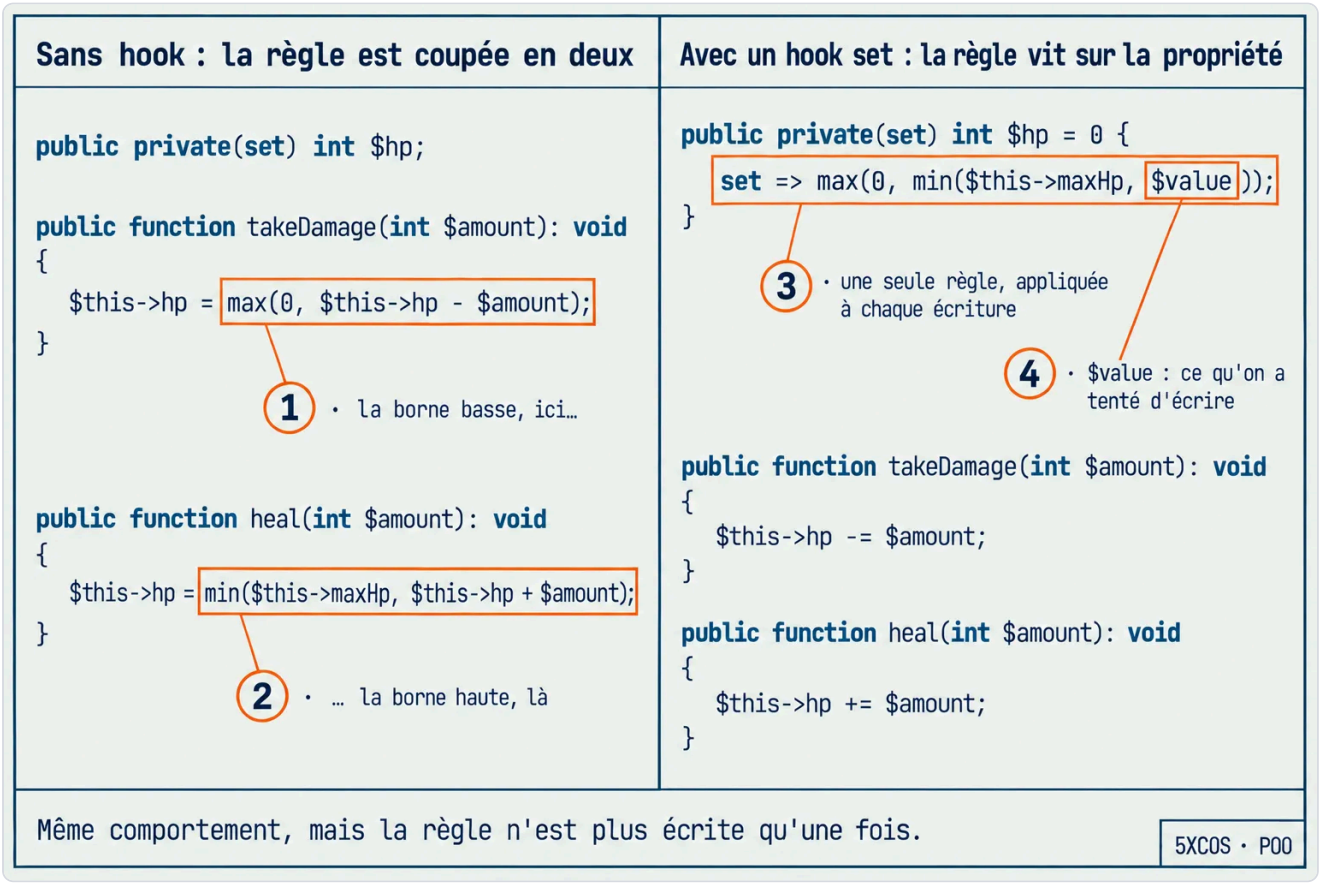
`$value` est une variable fournie par PHP : elle contient la valeur qu'on a tenté d'écrire. L'expression après `=>` donne la valeur réellement stockée. Avec `maxHp` à 10, après 50 dégâts, la valeur proposée est `-40`. `min(10, -40)` donne `-40`, puis `max(0, -40)` donne `0`.

Les affectations à `hp` dans les méthodes passent toutes par ce hook. Les deux méthodes se réduisent à une ligne :

```php
public function takeDamage(int amount): void
{
    $this->hp -= $amount;
}

public function heal(int amount): void
{
    this->hp+=this->hp += this->hp+=amount;
}
```

```php
arthur=newHero('Arthur');arthur = new Hero('Arthur');
arthur=newHero('Arthur');arthur->takeDamage(50);
echo arthur,PHPEOL;arthur, PHP_EOL;
arthur,PHPEOL;arthur->heal(999);
echo $arthur, PHP_EOL;
```

```
Arthur (0/10 PV)
Arthur (10/10 PV)
```



Les deux bornes sont maintenant vérifiées dans le hook de `hp`. La potion empoisonnée pourra écrire `$this->hp -= 3` sans rien vérifier. Seule exception : la valeur par défaut écrite dans la déclaration, ici `= 0`, ne passe pas par le hook.

Le même hook, écrit avec des accolades :

```php
public private(set) int $hp = 0 {
    set {
        $this->hp = max(0, min($this->maxHp, $value));
    }
}
```

Avec les accolades, vous écrivez vous-même `$this->hp = …` pour enregistrer la valeur : le hook remplace l'écriture, il ne vient pas s'y ajouter. Le résultat est le même, `Arthur (0/10 PV)` après 50 dégâts. Cette forme sert quand il faut plusieurs instructions, par exemple un `throw` pour refuser une valeur.

> **ATTENTION**
>
> Un hook `set` n'est pas un setter, comme le `setHp()` refusé au chapitre 1. Cette méthode ouvrirait l'écriture à tout le programme. Ici, `private(set)` réserve l'écriture à `Hero`, et le hook vérifie la valeur que `Hero` écrit.

4. Le hook `get` : une valeur calculée

Un héros est en pleine forme quand `hp` vaut `maxHp`. Vous pourriez stocker cette information dans un booléen, c'est-à-dire une valeur vraie ou fausse, `$isFullHealth`, et le mettre à jour dans `takeDamage()` et dans `heal()`. Un jour, une méthode oubliera cette mise à jour, et le booléen mentira. Un hook `get` calcule la valeur à chaque lecture :

```php
public bool $isFullHealth {
    get => $this->hp === $this->maxHp;
}
```

```php
arthur=newHero('Arthur');vardump(arthur = new Hero('Arthur');
var_dump(arthur=newHero('Arthur');vardump(arthur->isFullHealth);
arthur->takeDamage(1);vardump(arthur->takeDamage(1);
var_dump(arthur->takeDamage(1);vardump(arthur->isFullHealth);
```

```
bool(true)
bool(false)
```

Quand vous lisez `$arthur->isFullHealth`, PHP exécute le code du `get`. La valeur n'est pas stockée : elle est calculée à partir de `hp` et de `maxHp`. La propriété est **virtuelle**. Après un coup ou un soin, la prochaine lecture utilise les nouveaux points de vie. Et comme il n'y a pas de `set`, personne ne peut l'écrire :

```php
$arthur->isFullHealth = true;
```

```
Fatal error: Uncaught Error: Property Hero::$isFullHealth is read-only
```

Un hook `get` fait le même travail qu'une méthode, sans les parenthèses.

- **une propriété** pour ce qui se lit comme un état : `hp`, `name`, `isFullHealth` ;
- **une méthode** pour une action, dont le nom contient un verbe et qui change quelque chose : `takeDamage()`, `heal()`, `roll()` ;
- **une méthode** pour un calcul qui a besoin d'un paramètre : `distanceTo(Point $other)`.

Entre `isAlive()` et `isAlive`, les deux écritures se valent. Le Donjon garde la méthode. Ce qui compte, c'est qu'aucune des deux ne s'écrit de l'extérieur.

5. Valider à l'entrée : le constructeur

Une propriété `readonly` ne peut pas avoir de hook. Sa règle se vérifie donc au moment où le constructeur lui donne sa valeur. Un dé à zéro face ne doit pas exister :

```php
class Dice
{
    public function __construct(
        public readonly int sides,
    ) {
        if ($sides < 2) {
            throw new \InvalidArgumentException("Un dé a au moins 2 faces, $sides reçu.");
        }
    }
}

d0 = new Dice(0);
```

```
Fatal error: Uncaught InvalidArgumentException: Un dé a au moins 2 faces, 0 reçu.
```

Une **exception** est un objet qui signale un problème et interrompt le code en cours. `throw` la lance : le constructeur s'arrête, `new` ne renvoie aucun dé, et le programme s'arrête aussi, sauf si quelqu'un attrape l'exception. `InvalidArgumentException` est une classe fournie par PHP pour ce cas précis : l'argument reçu n'est pas acceptable. Le chapitre 6 revient sur les exceptions. Retenez la règle : **un objet impossible ne se construit pas**. On vérifie à l'entrée, une fois, et tout le reste du programme peut faire confiance.

Même chose pour un héros. Un nom vide ou un maximum de points de vie nul n'ont pas de sens. Dans le constructeur de `Hero`, avant les deux affectations, `trim()` enlève les espaces aux deux bouts, pour qu'un nom fait d'espaces soit refusé aussi :

```php
if (trim(name)===’’)thrownew\InvalidArgumentException(’Unhe´rosaunnom.’);if(name) === '') {
    throw new \InvalidArgumentException('Un héros a un nom.');
}

if (name)===’’)thrownew\InvalidArgumentException(’Unhe´rosaunnom.’);if(maxHp < 1) {
    throw new \InvalidArgumentException("maxHp doit valoir au moins 1, $maxHp reçu.");
}
```

Chaque essai se lance seul, puisque le premier arrête le script.

```php
new Hero('   ');
```

```
Fatal error: Uncaught InvalidArgumentException: Un héros a un nom.
```

```php
new Hero('Arthur', 0);
```

```
Fatal error: Uncaught InvalidArgumentException: maxHp doit valoir au moins 1, 0 reçu.
```

6. `Hero` au complet

Voici la classe entière. Elle fait quinze lignes de plus que celle du chapitre 1, et trois règles y sont écrites une seule fois : les bornes de `hp`, le nom non vide, le maximum positif. Pour l'essayer seule, copiez-la dans un fichier avec les lignes qui suivent. Dans le kata, vous modifiez la classe existante de `src/Hero.php`, vous ne remplacez pas le fichier.

PHP

```php
<?php

declare(strict_types=1);

class Hero
{
    public private(set) int maxHp;publicprivate(set)intmaxHp;

    public private(set) int maxHp;publicprivate(set)inthp = 0 {
        set => max(0, min(this->maxHp,this→maxHp, this->maxHp,value));
    }

    public bool isFullHealth {
        get => $this→hp === $this→maxHp;
    }

    public function __construct(
        public readonly string name,
        int maxHp=10,publicreadonlyintmaxHp = 10,
        public readonly int maxHp=10,publicreadonlyintstrength = 2,
    ) {
        if (trim(name)===’’)thrownew\InvalidArgumentException(’Unhe´rosaunnom.’);if(name)
=== '') {
            throw new \InvalidArgumentException('Un héros a un nom.');
        }

        if (name)===’’)thrownew\InvalidArgumentException(’Unhe´rosaunnom.’);if(maxHp < 1) {
            throw new \InvalidArgumentException("maxHp doit valoir au moins 1, maxHp
reçu.");
        }

        this→maxHp = maxHp;maxHp;
        maxHp;this→hp = maxHp;
    }

    public function takeDamage(int amount): void
    {
        this->hp-=this→hp -= this->hp-=amount;
    }

    public function heal(int amount): void
    {
        $this→hp += $amount;
    }

    public function isAlive(): bool
    {
        return $this→hp > 0;
    }

    public function __toString(): string
    {
        return sprintf('%s (%d/%d PV)', $this→name, $this→hp, $this→maxHp);
    }
}
```

```
arthur = new Hero('Arthur');
arthur->takeDamage(50);echoarthur→takeDamage(50);
echo arthur->takeDamage(50);echoarthur, ' ',
var_export(arthur->isFullHealth,true),PHPEOL;arthur→isFullHealth, true), PHP_EOL;
arthur->isFullHealth,true),PHPEOL;arthur→heal(50);
echo arthur, ' ', var_export(arthur→isFullHealth, true), PHP_EOL;
```

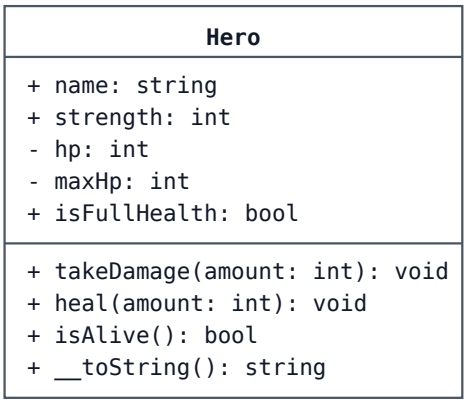
```
Arthur (0/10 PV) false
Arthur (10/10 PV) true
```

L'ordre des deux dernières lignes du constructeur compte. Le hook de `hp` lit `maxHp`. Si `hp` reçoit sa valeur en premier, `maxHp` n'en a pas encore, et PHP s'arrête :

```
Fatal error: Uncaught Error: Typed property Hero::$maxHp must not be accessed before
initialization
```

`maxHp` reste `private(set)` et non `readonly`, alors que rien ne la modifie ici. C'est un choix pour la suite du kata : un objet pourra un jour augmenter le maximum. Une propriété `readonly` l'interdirait.

En UML, la convention du cours est simple : une propriété que seule la classe écrit se note avec un `-`, qu'elle soit lisible de dehors ou non. Une propriété calculée se note comme les autres.



Dans Laravel, vous écrirez `$post→title` pour lire un titre et `$post→title = 'Le donjon'` pour le changer, avant d'enregistrer avec `save()`. Ces propriétés n'existent pas dans le fichier `Post.php` : le modèle les fabrique à partir des colonnes de la table. Il le fait avec les méthodes magiques `__get()` et `__set()`, cousines du `__toString()` du chapitre 1. C'est le modèle qui décide ce qui se lit, ce qui s'écrit et ce qui se convertit au passage. Le chapitre 11 le montre sur du vrai code.

À retenir

- L'encapsulation : la classe cache ses données et garantit ses règles. Une règle qui reste toujours vraie est un invariant, et il s'écrit à un seul endroit.

- Tout est `private` par défaut. On ouvre la lecture avec `readonly` (fixée à la création) ou `private(set)` (la classe la modifie). On n'ouvre jamais l'écriture.
- Un hook `set` applique une règle à chaque écriture faite par la classe. Un hook `get` fait une propriété virtuelle : `isFullHealth` est calculée à chaque lecture, sans rien stocker.
- Ce qui est `readonly` se valide dans le constructeur. Un objet impossible ne se construit pas : `throw new \InvalidArgumentException(…)`.
- Une propriété pour un état, une méthode pour une action.

Exercices

Kata : encapsulation

Dépôt : https://github.com/opmvpc/poo-katas-26, le même qu'au niveau 1. Lancez ce groupe :

```bash
composer test -- --group=encapsulation
```

Lancez les tests et repérez ceux qui échouent encore. Ils décrivent le travail à faire dans `src/Hero.php` et `src/Dice.php`. Après chaque étape, lancez aussi `composer test -- --group=niveau-1` : les tests déjà réussis doivent encore réussir.

Ce que les tests attendent, exactement :

`Dungeon\Dice`

- `new Dice(1)` et `new Dice(0)` lèvent une `InvalidArgumentException` ; `new Dice(2)` est accepté.

`Dungeon\Hero`

- `new Hero(' ')` et `new Hero('Arthur', 0)` lèvent une `InvalidArgumentException` ;
- `maxHp` ne s'écrit pas de l'extérieur : `$hero->maxHp = 99` lève une `Error` ;
- `isFullHealth` vaut `true` à la création, `false` après `takeDamage(1)`, `true` à nouveau après `heal(100)` ;
- `isFullHealth` ne s'écrit pas : `$hero->isFullHealth = true` lève une `Error` ;
- les tests du niveau 1 passent toujours : `takeDamage()` et `heal()` gardent `hp` entre 0 et `maxHp`.

1. **Le hook sur `hp`.** Ajoutez le hook `set` de la partie 3 sur la propriété `hp`, puis réduisez `takeDamage()` et `heal()` à une ligne chacune. Relancez le niveau 1, puis relisez votre hook : les deux bornes y sont, et nulle part ailleurs.
2. **`isFullHealth`.** Dans `src/Hero.php`, la propriété est déjà déclarée avec un hook `get` qui lève « À implémenter ». Remplacez-le par le calcul.
3. **Valider à l'entrée.** `new Dice(1)` et `new Dice(0)` doivent lever une `InvalidArgumentException`. Même chose pour `new Hero(' ')` et `new Hero('Arthur', 0)`.
4. **Refuser les montants négatifs.** Dans `Hero`, un dégât négatif soigne encore, et un soin négatif blesse encore : le hook borne les points de vie, il ne juge pas le sens de l'action. Refusez les montants négatifs dans `takeDamage()` et `heal()` avec la technique de la partie 5. Puis écrivez le test qui le

vérifie, à la fin de `tests/EncapsulationTest.php`. C'est le premier test que vous écrivez : prenez modèle sur ceux du fichier, et terminez-le par `->group('encapsulation')`, sinon la commande ci-dessus ne le lance pas.

5. **Poussez**, et vérifiez que la tâche `encapsulation` est verte dans l'onglet Actions, à côté de `niveau-1`.

Si vous bloquez plus de dix minutes sur un test, montrez le test et votre code.

Sur papier

Une salle du donjon a un nom, qui ne change jamais. Elle contient au plus un objet posé au sol : un `Item`, ou rien du tout. Ce type s'écrit `?Item`, et le point d'interrogation veut dire « ou `null` ». Elle compte ses visites, un nombre qui augmente à chaque passage du héros et que l'affichage doit pouvoir lire.

Pour chacune des trois propriétés, écrivez la ligne de déclaration complète : visibilité de lecture, visibilité d'écriture ou `readonly`, type, valeur de départ s'il en faut une. Puis nommez la ou les méthodes qui ont le droit de modifier chacune.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/encapsulation

2.4 Modéliser : du texte au diagramme de classes

Séance 2 · ~2 périodes

Slides du chapitre (PDF) PDF

Vous savez écrire une classe et faire tourner des tests. Reste une question qui arrive avant le code : **quelles classes écrire ?** Comment passer d'un énoncé en français à une liste de classes ?

C'est le rôle du **diagramme de classes**. Il est écrit en UML, un langage de dessin normalisé qui sert à représenter un programme. Ce dessin, vous le referez au tableau avec un collègue, et vous le retrouverez dans la documentation de la plupart des projets. Quand la documentation de Laravel dira qu'un `Post` a plusieurs `Comment`, vous saurez le dessiner.

À la fin de ce chapitre, vous serez capables de :

1. **extraire** d'un énoncé en français les classes, les attributs et les méthodes candidates ;
2. **dessiner** une classe en notation UML : compartiments, visibilités, types ;
3. **relier** deux classes avec la bonne relation, ses deux rôles et ses multiplicités ;
4. **critiquer** un modèle : repérer la classe qui fait tout et les tableaux qui cachent des classes.

Le code qui sent mauvais

Cinq minutes, en binôme. Ce code sent mauvais : un détail d'écriture y trahit un défaut de conception. Lisez-le et répondez à une seule question : **combien de classes manquent ici ?**

```php
<?php

declare(strict_types=1);

class Game
{
    public array hero=['name'⇒'Arthur','hp'⇒10];publicarrayhero = ['name' ⇒ 'Arthur',
'hp' ⇒ 10];
    public array hero=['name'⇒'Arthur','hp'⇒10];publicarrayrooms = [['name' ⇒ 'Entrée',
'monster' ⇒ null]];
    public array $items = [['name' ⇒ 'Épée', 'weight' ⇒ 2.0, 'room' ⇒ 0]];

    public function play(): void
    {
        $this→hero['hp'] -= 3;
        $this→rooms[0]['monster'] = ['name' ⇒ 'Gobelin', 'hp' ⇒ 5];
        echo $this→hero['name'], ' : ', $this→hero['hp'], " PV\n";
        echo count($this→items), " objet(s) au sol\n";
    }
}

(new Game())→play();
```

```
Arthur : 7 PV
1 objet(s) au sol
```

Ce code tourne.

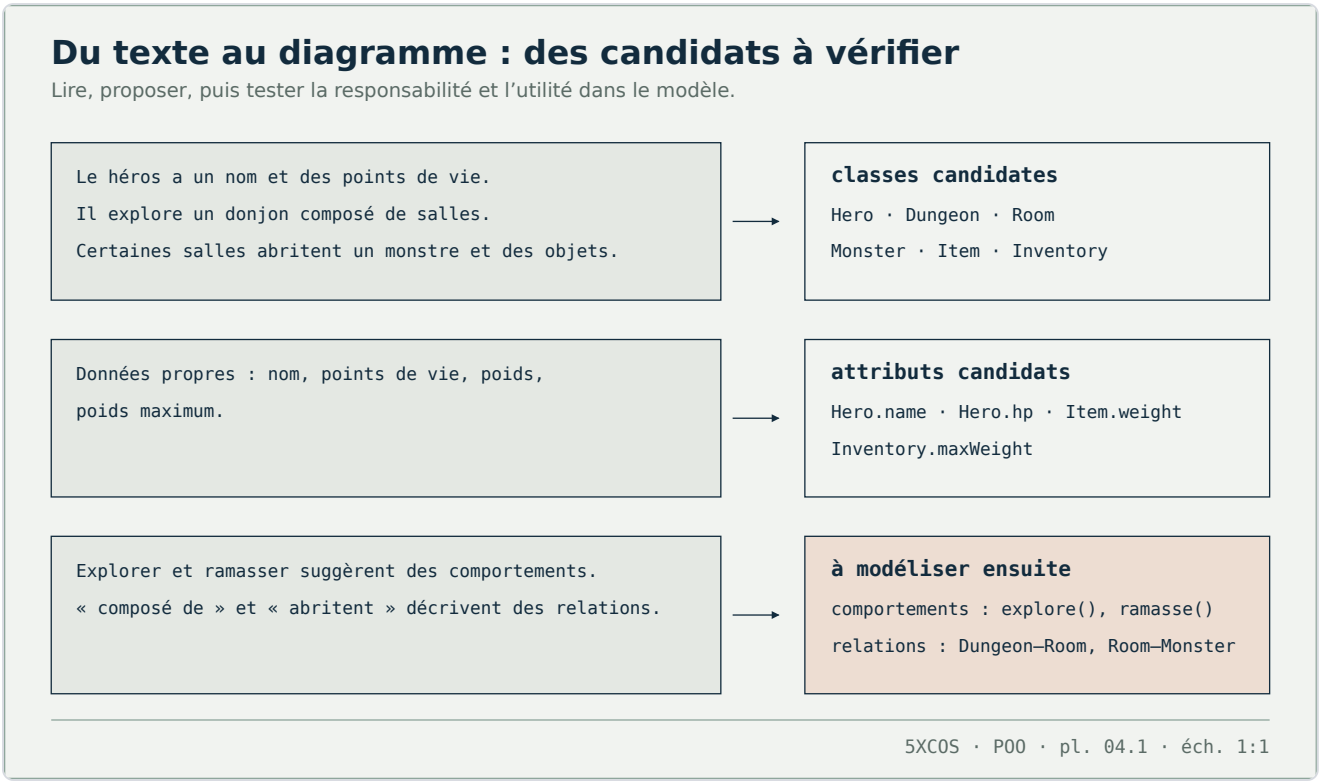
Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/modeliser-le-diagramme-de-classes

1. Lire un énoncé avec un crayon

La méthode est mécanique : vous relisez l'énoncé trois fois, avec un crayon.

Passage	Ce que vous soulignez	Ce que ça devient
1	Les **noms communs** importants, ceux qui reviennent	des **classes** candidates
2	Les **données** : adjectifs, nombres, qualificatifs	des **attributs**
3	Les **verbes** : ce que le système doit savoir faire	des **méthodes**

L'UML dit **attribut** là où PHP dit propriété. Même chose, deux mots.



Trois questions ensuite, pour réduire la liste :

- **Ce nom a-t-il des données à lui ?** Sinon, c'est un attribut. « Le héros a un nom » : `name` est une chaîne de caractères, pas une classe `Name`.
- **Ce nom sert-il à autre chose qu'à l'affichage ?** L'écran de jeu ne fait pas partie du modèle du jeu.
- **Quelle classe détient les données de ce verbe ?** `takeDamage()` modifie `hp` : il va dans `Hero`, pas dans `Game`.

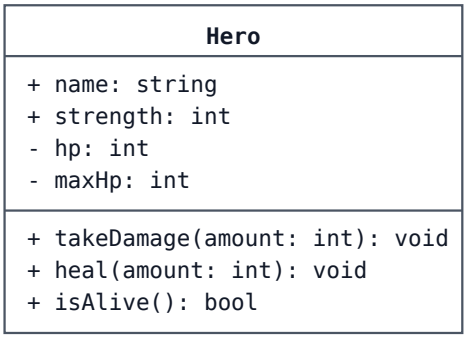
Appliquez la méthode à une phrase du Donjon : « le héros explore un donjon composé de salles ; certaines salles abritent un monstre et contiennent des objets que le héros peut ramasser dans son sac ». Les noms donnent six candidats : héros, donjon, salle, monstre, objet, sac.

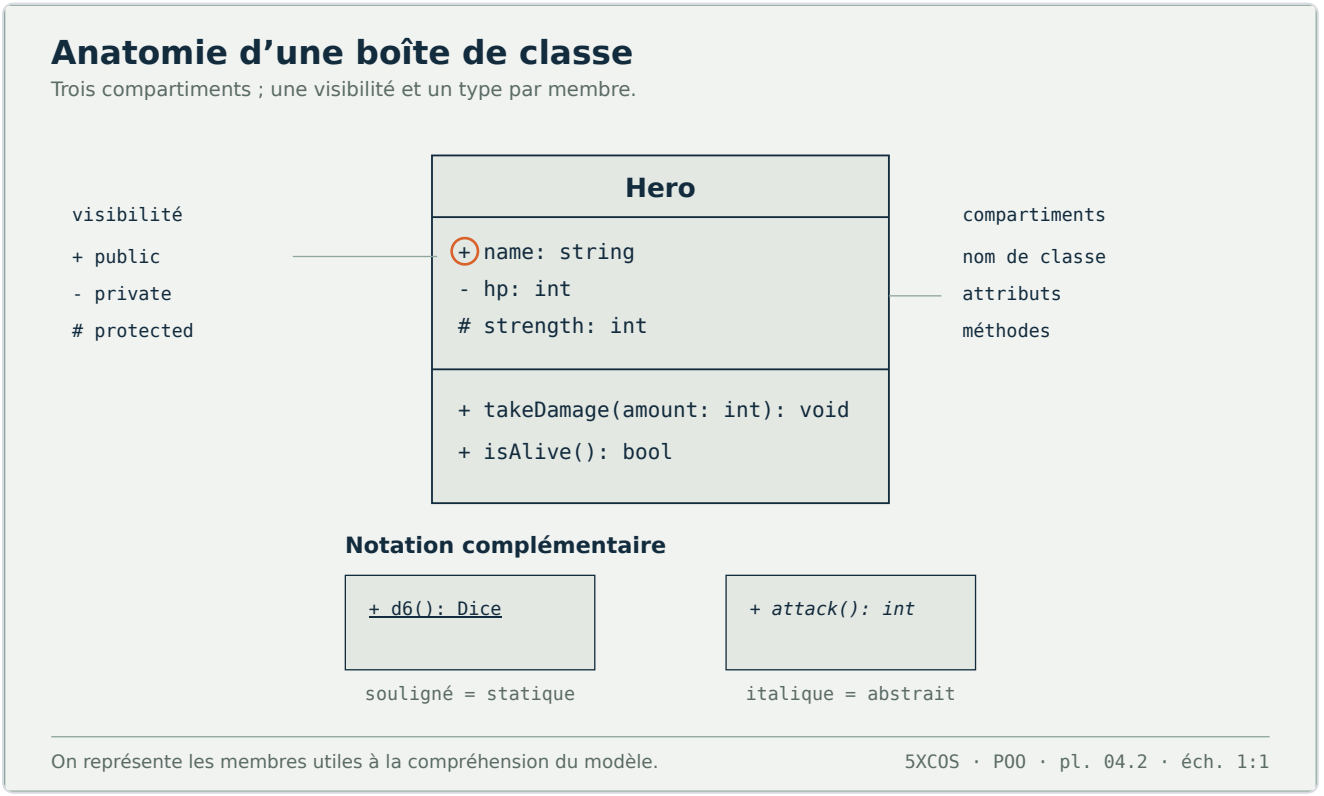
Le sac a des données à lui, un poids maximum précisé plus loin dans l'énoncé complet, et une liste d'objets. Il a aussi des actions : ajouter, retirer, peser. Ce sac devient donc une classe, `Inventory`. « Exploration » n'a aucune donnée propre : ce sera une méthode. Les verbes se placent ensuite. « Ramasser » a besoin du sac et de l'objet, donc il va dans `Hero`, qui possède le sac.

Ces six noms sont des pistes, pas encore un modèle. Le dessin les confirme ou les élimine.

2. La boîte de classe

Une classe se dessine dans un rectangle à **trois compartiments** : le nom, les attributs, les méthodes.



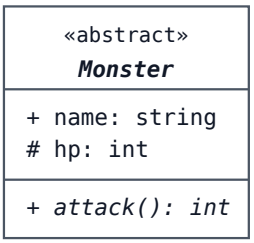


Les symboles :

Symbole	Signification
`+`	`public` : visible de partout
`-`	`private` : visible seulement dans la classe. Convention du cours : une propriété `public private(set)` se note aussi `-`, ce qui compte pour le modèle est que seule la classe l'écrit
`#`	`protected` : la classe et ses classes filles, au chapitre 6
`~`	visible dans le même paquet. PHP n'a pas cette visibilité, vous la croiserez dans d'autres cours
`nom: type`	un attribut et son type
`nom: type [0..1]`	un attribut facultatif, qui peut valoir `null`

Symbole	Signification
`methode(param: type): retour`	une méthode, ses paramètres typés et son type de retour
souligné	membre **statique**. PlantUML le note `{static}`

Une classe **abstraite** est une classe dont on ne fabrique jamais d'objet directement. Deux notations existent : le nom en italique, ou le stéréotype `«abstract»` au-dessus du nom. Le cours recommande le stéréotype : il se lit sans hésiter, au tableau comme à l'écran.



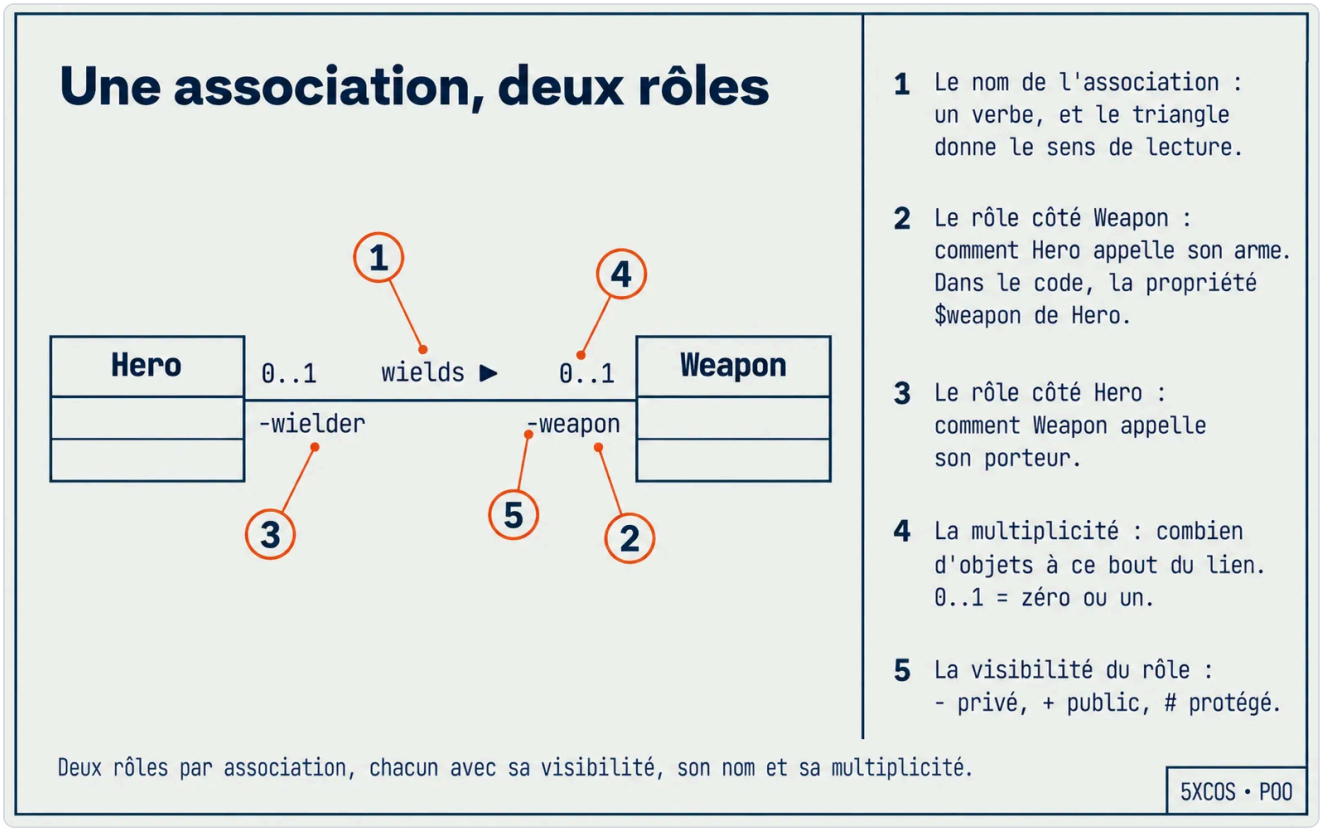
Deux autres boîtes existent, l'interface et l'énumération. Elles arrivent au chapitre 7 avec leur code.

> **ASTUCE**
>
> Un diagramme de classes n'est pas une recopie de votre code. Vous y mettez ce qui aide à comprendre : les données importantes, les méthodes publiques, les relations. Les méthodes de lecture, d'écriture et d'affichage, comme `__toString()`, peuvent rester en dehors du dessin.

3. Les relations

Association : « est en lien avec »

Un trait plein entre deux boîtes. Une association représente un lien durable entre les objets de deux classes.



Une association porte toujours quatre choses :

1. **Un nom**, un verbe, avec un petit triangle qui donne le sens de lecture : « un héros *wields* une arme ».
2. **Deux rôles**, un à chaque bout. Le rôle dit comment la classe d'en face appelle celle-ci : côté `Weapon`, le rôle `weapon` est le nom que `Hero` donne à son arme. Dans le code, c'est la propriété `$weapon` de `Hero`.
3. **Une visibilité par rôle**, `-`, `+` ou `#`, la même que pour un attribut. Un rôle privé devient une propriété privée.
4. **Deux multiplicités**, une à chaque bout : combien d'objets participent au lien de ce côté.

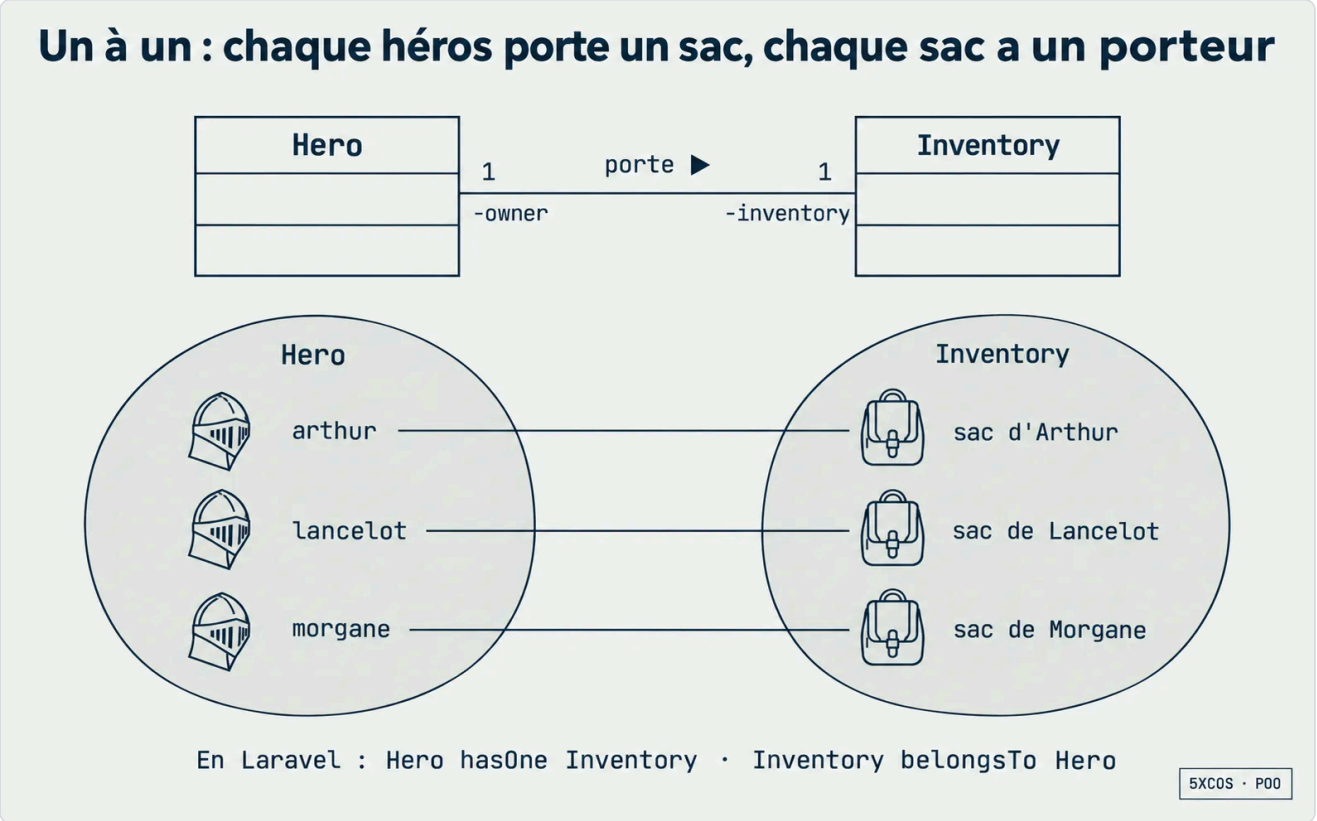
Un rôle par côté, sans exception, agrégation et composition comprises. Dans le code, vous n'êtes pas obligé de programmer les deux sens du lien. Le chapitre 5 en code un seul : la salle connaît l'objet posé au sol, l'objet ne connaît pas la salle. Le modèle, lui, nomme les deux.

Les multiplicités :

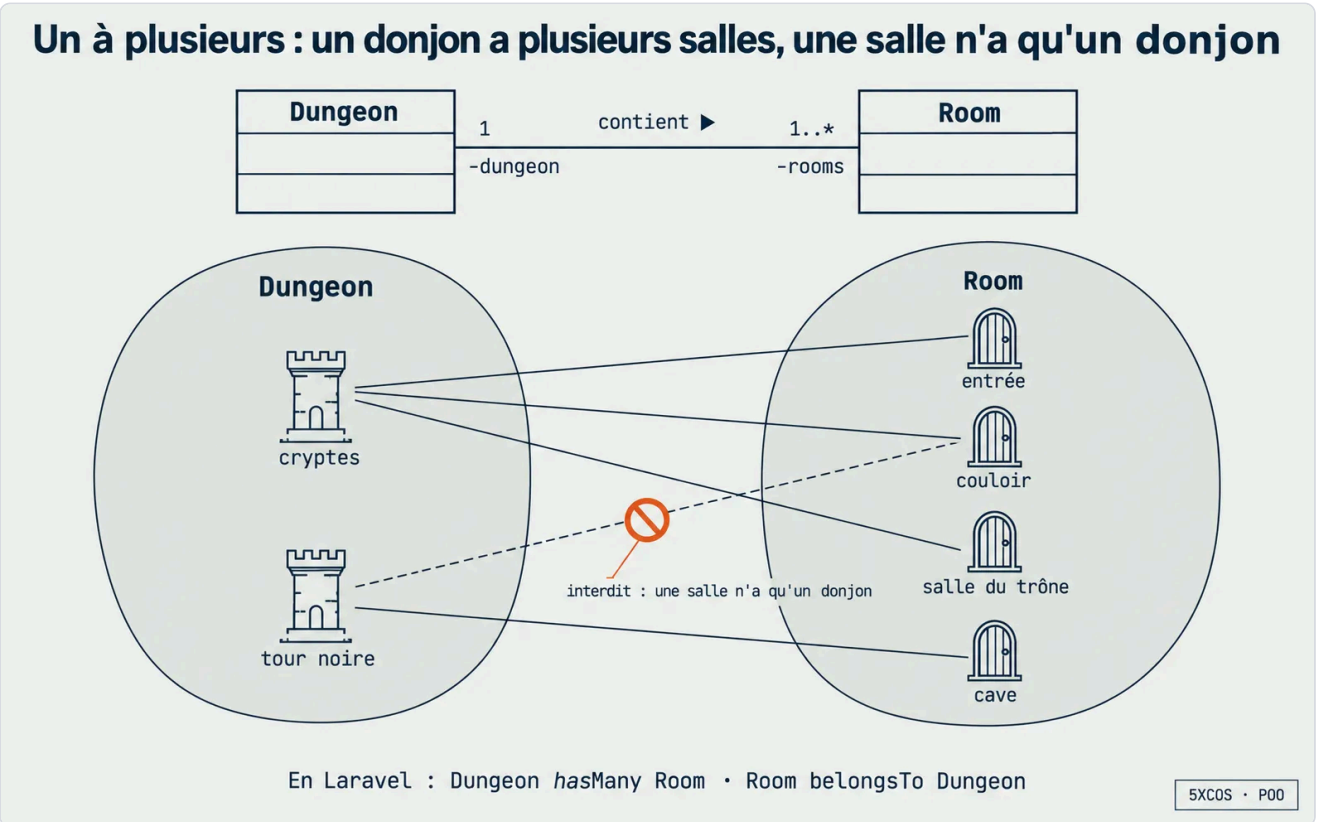
Multiplicité	Se lit	En PHP
`1`	exactement un	une propriété obligatoire
`0..1`	zéro ou un	une propriété qui peut valoir `null`
`*` ou `0..*`	zéro ou plusieurs	un tableau, éventuellement vide
`1..*`	au moins un	un tableau jamais vide

Trois combinaisons reviennent tout le temps. Vous les retrouverez telles quelles dans Laravel, où chaque relation entre deux modèles porte un nom de méthode.

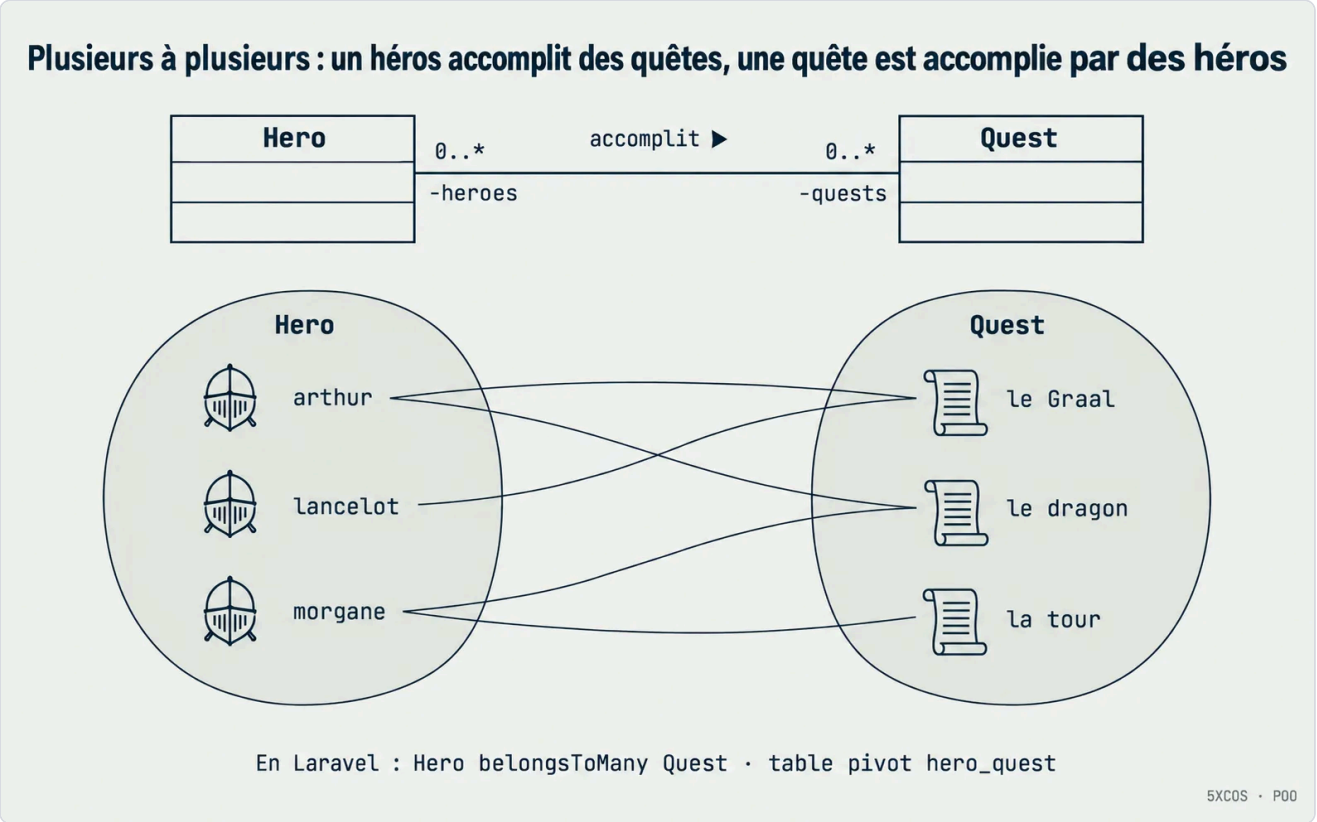
Un à un. Chaque héros porte un sac, chaque sac a un porteur.



Un à plusieurs. Un donjon contient plusieurs salles, une salle n'a qu'un donjon. C'est la relation la plus fréquente : un `Post` a plusieurs `Comment`, un `Comment` appartient à un seul `Post`.



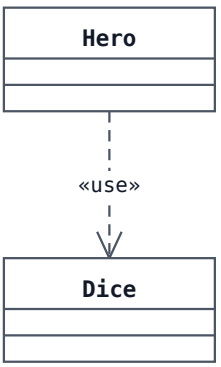
Plusieurs à plusieurs. Un héros accomplit des quêtes, une quête est accomplie par plusieurs héros. En base de données, ce lien demande une table intermédiaire, que Laravel appelle table pivot.



Multiplicités	Exemple du Donjon	Dans Laravel
`1` — `1`	`Hero` porte `Inventory`	`hasOne` d'un côté, `belongsTo` de l'autre
`1` — `0..*`	`Dungeon` contient `Room`	`hasMany` du côté 1, `belongsTo` du côté plusieurs
`0..*` — `0..*`	`Hero` accomplit `Quest`	`belongsToMany` des deux côtés, avec une table pivot

Dépendance : « utilise »

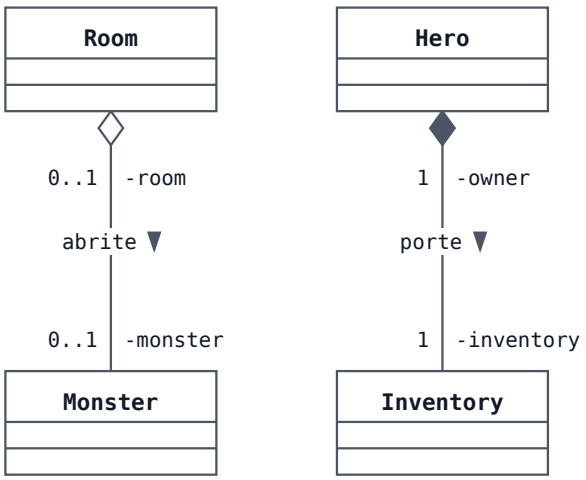
Un trait pointillé fléché. Une classe a besoin d'une autre le temps d'un appel, sans la garder.

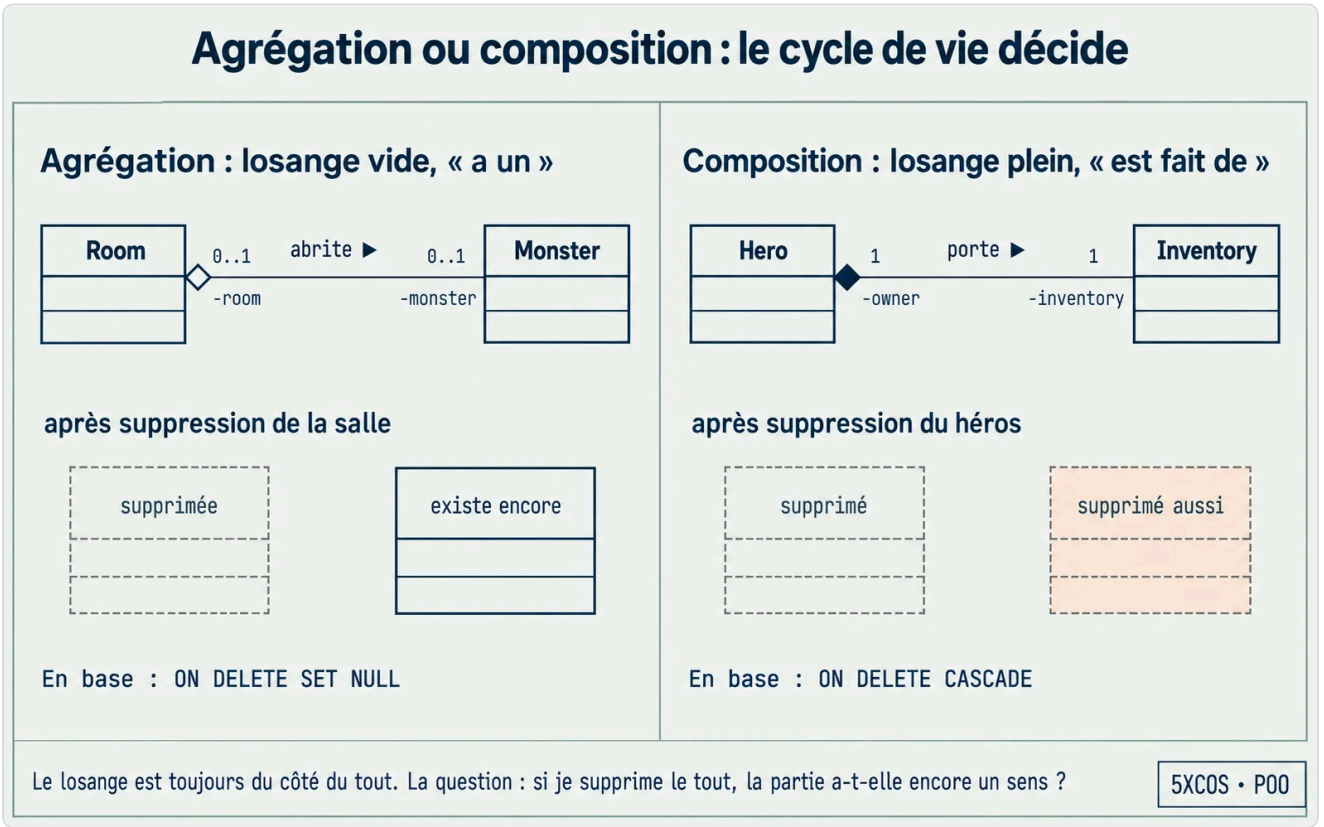


Dans le Donjon, une méthode de `Hero` reçoit un `Dice` en paramètre pour lancer un jet, puis l'oublie : le dé n'est rangé dans aucune propriété. C'est la relation la plus faible du lot. Dessinez-la seulement si elle aide à comprendre la méthode.

Agrégation et composition : « fait partie de »

Les deux disent qu'un objet fait partie d'un autre. Le losange se place **du côté du tout**. Elles ne disent pas la même chose sur les durées de vie.





Agrégation ◇——— : losange vide, « a un ». Le tout et la partie ont des vies indépendantes. Dans le Donjon, une `Room` **a un** `Monster`. Le gobelin existait avant qu'on ouvre la porte, et si la salle s'effondre il ira ailleurs. La norme UML ne garantit rien de plus qu'une annotation « tout et partie ». Convention du

cours : la partie survit au tout et peut changer de tout.

Composition ◆——— : losange plein, « est fait de ». La partie est créée par le tout et **meurt avec** lui. Deux contraintes sont fixées par UML : la multiplicité côté tout vaut toujours `1`, et une partie n'entre que dans une seule composition à la fois. Dans le Donjon, un `Hero` **est fait**, entre autres, d'un `Inventory`. Le sac est créé dans le constructeur du héros, personne d'autre ne le possède, et il disparaît avec lui.

En base de données, la même question se pose au moment de supprimer une ligne. La réponse est déjà dans le losange :

Losange	Quand le tout est supprimé	Clé étrangère
vide, agrégation	la partie reste, son lien est vidé	`ON DELETE SET NULL`
plein, composition	la partie est supprimée avec lui	`ON DELETE CASCADE`

> **ATTENTION**
>
> Beaucoup de supports intervertissent les deux définitions et écrivent que l'agrégation est le lien fort. C'est l'inverse. Le test qui tranche : si je supprime le tout, la partie a-t-elle encore un sens toute seule ? Oui, c'est une agrégation. Non, c'est une composition.

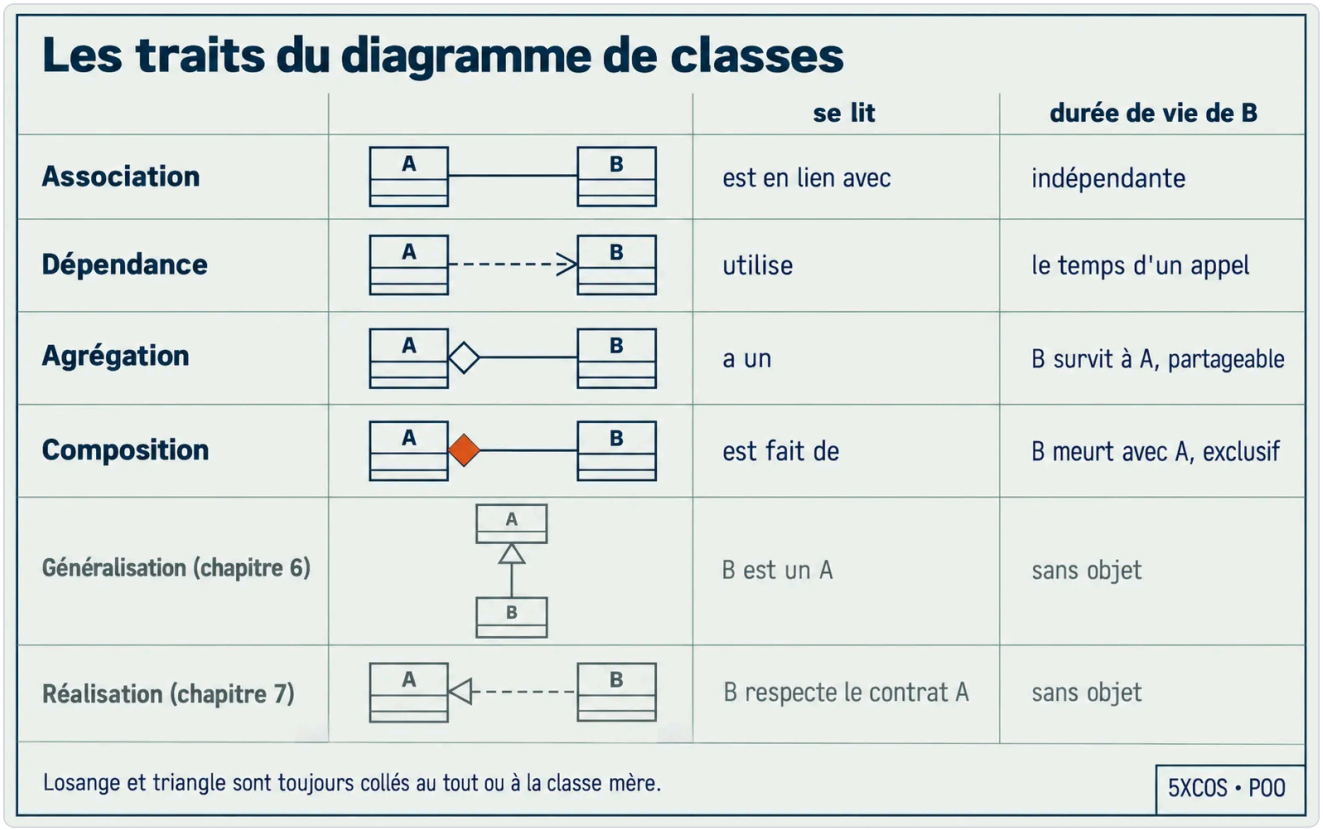
Un indice fiable la plupart du temps : **qui fabrique l'objet ?** Si le tout fabrique la partie dans son constructeur, c'est une composition. Si la partie arrive de l'extérieur, c'est une agrégation ou une association. Cet indice se trompe parfois. Quand il contredit la phrase « est fait de », suivez la phrase. Le chapitre 5 code exactement ces deux cas.

Ce qui vient plus tard

Deux relations arrivent dans les chapitres suivants :

- la **généralisation** ◁———, trait plein et triangle vide collé à la classe mère, qui dit « est un » : `Weapon` est un `Item`. Chapitre 6.
- la **réalisation** ◁- - - -, trait pointillé et triangle vide collé à l'interface, qui dit « respecte le contrat » : `Hero` réalise `Fighter`. Chapitre 7.

Au chapitre 6, vous verrez que beaucoup d'héritages auraient dû être une simple propriété.



Les quatre relations de ce chapitre, avec ce qu'il faut taper en PlantUML :

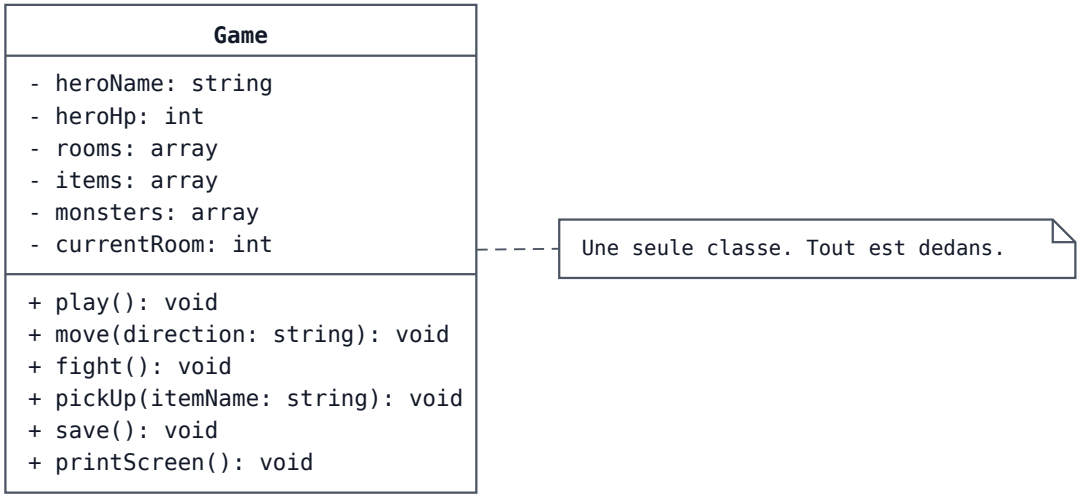
Relation	Trait	Se lit	Durée de vie de la partie	En PlantUML
Association	———	« est en lien avec »	indépendante	`A -- B`
Dépendance	- - - ->	« utilise »	le temps d'un appel	`A ..> B`
Agrégation, losange vide	◇———	« a un »	survit au tout, peut changer de tout	`A o-- B`, losange contre `A`
Composition, losange plein	◆———	« est fait de »	meurt avec le tout, exclusif	`A *-- B`, losange contre `A`

En PlantUML, le tout s'écrit à gauche et le losange se dessine contre lui. Les multiplicités et les rôles se mettent entre guillemets, de chaque côté : `Hero "1 -owner" *-- "1 -inventory" Inventory`.

4. Deux modèles du Donjon

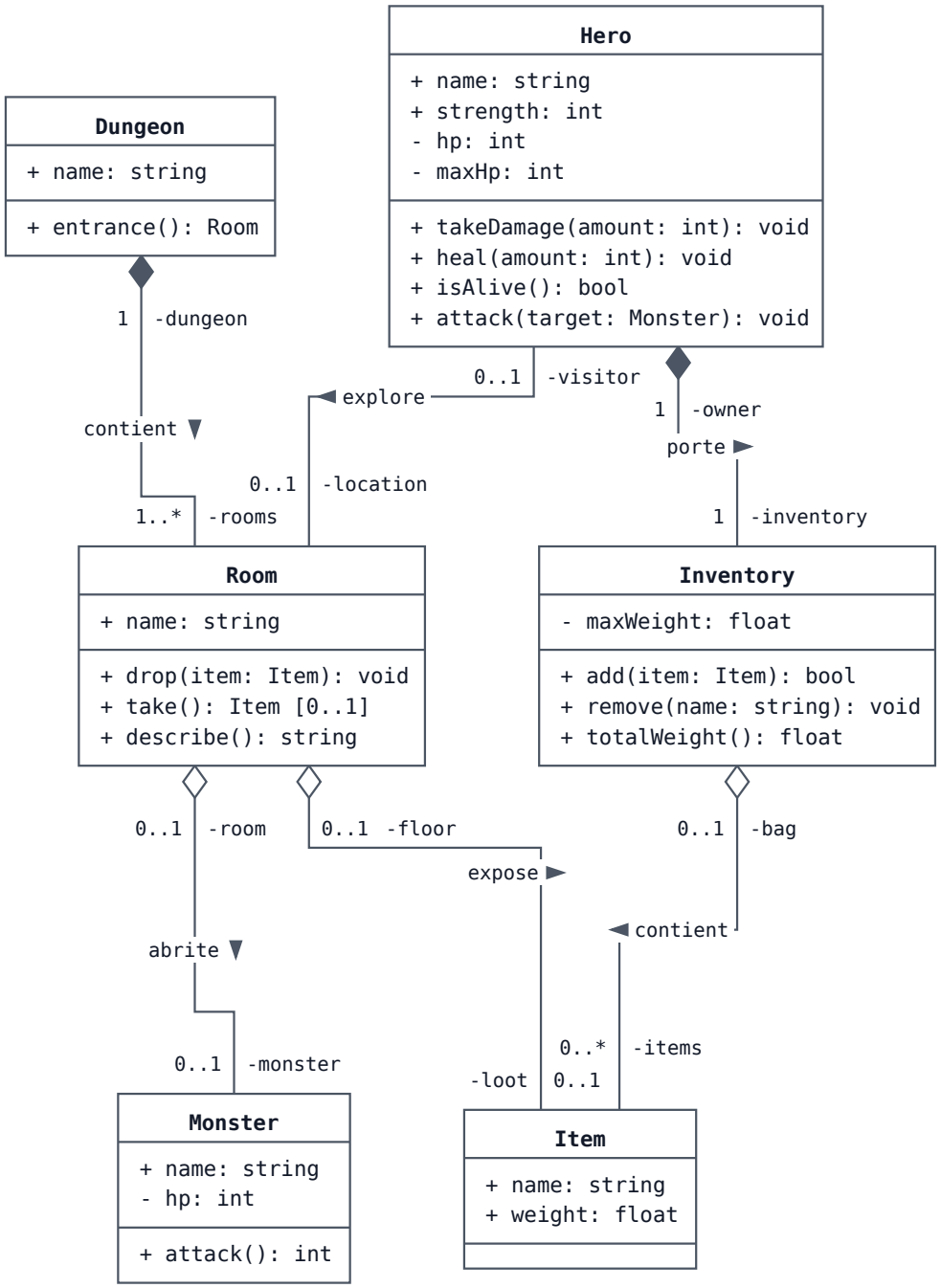
Même énoncé, deux diagrammes. Prenez trois minutes pour lister ce qui cloche dans le premier avant d'ouvrir la carte.

Un modèle à améliorer



Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/modeliser-le-diagramme-de-classes

Un meilleur modèle



Lisez-le en français : « un donjon **est fait** d'au moins une salle ; une salle **a** au plus un monstre et **a** au plus un objet au sol ; un héros **est fait** d'un inventaire ; un inventaire **a** zéro ou plusieurs objets ; un héros explore au plus une salle à la fois. »

Le combat n'est pas un trait : un héros attaque un monstre le temps d'un appel, `attack(target: Monster)`, puis l'oublie. C'est une dépendance, et elle n'a pas besoin d'être dessinée.

`currentRoom: int` a disparu. Il est remplacé par l'association `explore` entre `Hero` et `Room` : le héros connaît sa salle, pas un numéro de case. `Item` apparaît des deux côtés, en agrégation. C'est voulu : une épée posée au sol passe dans le sac sans changer d'identité. Le diagramme n'interdit pas qu'un objet

soit au sol et dans le sac en même temps. C'est une règle du jeu, et c'est le code qui la fera respecter. Un diagramme montre la structure, pas toutes les règles.

5. Dessiner

Commencez toujours sur papier, au crayon. Une feuille, une gomme, des boîtes vides, puis les traits. Vous effacez un losange en deux secondes, et vous voyez tout le modèle d'un coup. C'est le meilleur outil pour une première version, et c'est ce que vous ferez au tableau avec votre binôme.

Pour mettre le dessin au propre, trois outils :

Outil	Ce que c'est	Quand
Modelio	un logiciel libre avec une interface graphique, qui produit des diagrammes conformes à UML 2	**recommandé** pour vos rendus : rôles, multiplicités et losanges sont posés à la souris et toujours corrects
StarUML	un autre logiciel graphique, plus léger	si Modelio ne s'installe pas chez vous
PlantUML	un diagramme écrit en texte, comme les blocs de ce chapitre, rendu par un site ou une extension de votre éditeur	quand vous voulez versionner le diagramme avec le code

Mermaid est **interdit** dans ce cours : il ne respecte pas le standard UML 2. Ses associations n'ont ni rôles ni sens de lecture corrects, et ses losanges sont approximatifs. Un diagramme Mermaid sera refusé.

En PlantUML, le squelette minimal tient en trois lignes :

```
@startuml
Hero "1 -owner" *-- "1 -inventory" Inventory : porte >
@enduml
```

Le rendu attendu pour l'exercice ci-dessous est au choix : une photo nette de votre dessin au propre, un export Modelio ou StarUML, ou un fichier PlantUML.

À retenir

- Les **noms** de l'énoncé donnent les classes candidates, les **données** les attributs, les **verbes** les méthodes. Un nom sans donnée ni comportement n'est pas une classe.
- Une boîte UML a trois compartiments, `+ - #` pour la visibilité, `nom: type` partout, souligné pour statique, `«abstract»` pour une classe abstraite.
- Toute relation porte **deux rôles** avec leur visibilité et **deux multiplicités**. Une association porte en plus un nom et son sens de lecture.
- **Agrégation, losange vide, « a un »** : la partie survit au tout, en base `ON DELETE SET NULL`. **Composition, losange plein, « est fait de »** : la partie meurt avec le tout, multiplicité `1` côté tout, en base `ON DELETE CASCADE`.

- Une classe qui sait tout faire et un `array` sans contenu déclaré sont deux signaux qu'il manque des classes.

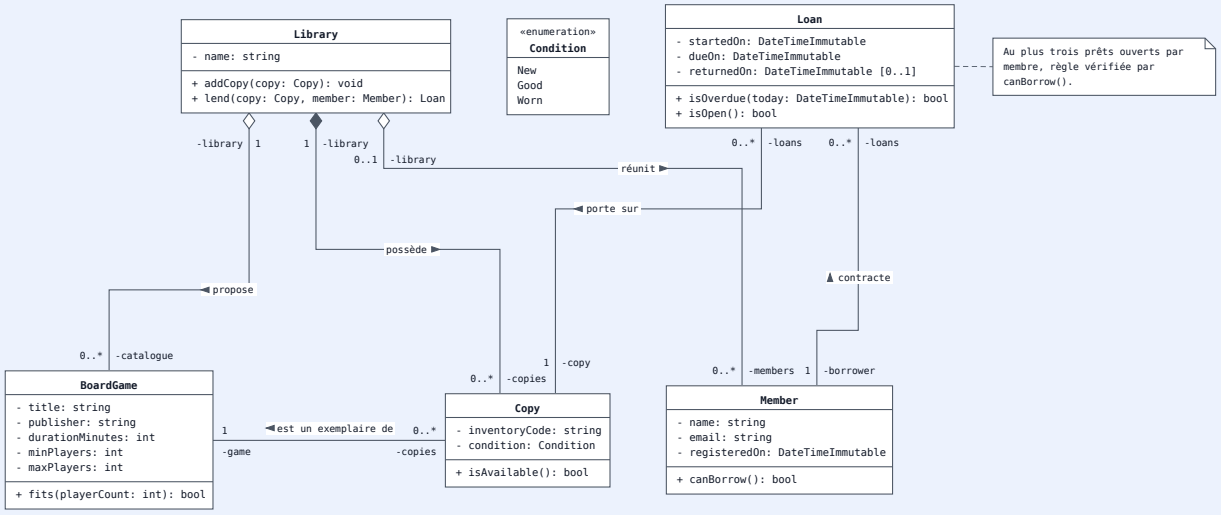
Exercice : la ludothèque

Modélisez le système suivant. Rendu attendu : **un diagramme de classes** avec les attributs typés, les méthodes publiques utiles, les relations avec **leurs deux rôles** et **toutes** les multiplicités. Pas de code. Papier d'abord, mise au propre ensuite. Comptez 40 minutes.

> *Un club de quartier gère une ludothèque. Elle possède un catalogue de **jeux de société** : chaque jeu a un titre, un éditeur, une durée moyenne en minutes, un nombre de joueurs minimum et un nombre maximum. La ludothèque possède plusieurs **exemplaires** physiques d'un même jeu ; chaque exemplaire a un code d'inventaire et un état, neuf, bon ou usé. Les **membres** du club ont un nom, un e-mail et une date d'inscription. Un membre peut emprunter jusqu'à trois exemplaires à la fois. Un **prêt** enregistre l'exemplaire emprunté, le membre, la date de sortie et la date de retour prévue ; il permet de savoir s'il est en retard. Un exemplaire ne peut être prêté qu'à un seul membre à la fois. Si la ludothèque ferme, ses exemplaires n'ont plus de raison d'être ; les membres, eux, existaient avant elle.*

Cinq questions à vous poser pendant le travail, dans l'ordre :

1. Combien de classes ? Comptez-les avant de dessiner.
2. Où sont les compositions ? Cherchez dans l'énoncé les parties qui disparaissent avec le tout.
3. Le prêt est-il une classe ou un simple trait ? Regardez s'il a des données à lui.
4. Comment traduisez-vous « jusqu'à trois exemplaires à la fois » ?
5. Où mettez-vous la méthode qui dit si un prêt est en retard ?



Cinq points de vigilance :

1. **`Loan` est une classe.** Un prêt a trois dates à lui et un comportement, `isOverdue()`. Dès qu'un lien porte des données, il devient une classe.
2. **`Library *-- Copy` est une composition** : l'énoncé dit que les exemplaires n'ont plus de raison d'être sans la ludothèque. Multiplicité `1` côté tout, et `Copy` n'entre dans aucune autre composition. **`Library o-- Member` est une agrégation** : les membres existaient avant.
3. **`Copy -- BoardGame` est une association.** Un jeu est un modèle, un exemplaire est un objet physique, et chaque exemplaire connaît son modèle. Confondre les deux est l'erreur classique. Avec un simple `quantity: int`, on ne sait plus quel exemplaire est chez qui.
4. **« Trois exemplaires à la fois » n'est pas une multiplicité.** La relation garde `0..*` parce qu'un membre accumule des prêts au fil des années. La limite porte sur les prêts encore ouverts, et elle vit dans `canBorrow()`. Une règle de gestion se code dans une méthode, elle ne se dessine pas comme multiplicité.
5. **`returnedOn` vaut `null` tant que l'exemplaire n'est pas rendu.** C'est ce que lit `isOpen()`. L'état de l'exemplaire est une liste fermée, neuf, bon ou usé : c'est une énumération, `Condition`. Vous la coderez au chapitre 7.

`isOverdue()` est sur `Loan`, parce que c'est `Loan` qui détient `dueOn`. Une méthode va toujours là où sont ses données.

2.5 Relations entre objets : posséder, partager, pointer

Séance 2 · ~3 périodes

Slides du chapitre (PDF) PDF

Au chapitre précédent, vous avez dessiné des traits entre des boîtes. Ce chapitre écrit ces traits en PHP. Un losange plein, un losange vide, un trait : chacun a sa traduction en code. Et le choix se joue presque toujours sur une seule ligne, celle où l'objet est fabriqué.

En PHP, **deux variables peuvent désigner le même objet**. La section 4 le montre sur `Hero`.

À la fin de ce chapitre, vous serez capables de :

1. **écrire** une propriété typée par une classe et un tableau d'objets correctement annoté ;
2. **traduire** en PHP une composition et une agrégation, et justifier votre choix ;
3. **expliquer** pourquoi deux variables peuvent modifier le même objet, et le prouver ;
4. **manipuler** une propriété qui peut valoir `null`, avec `?Type` et `?->` ;
5. **comparer** deux objets avec `==` et `===`.

Le code qui sent mauvais

Cinq minutes, en binôme. Ce code tourne, et il sent mauvais : un détail d'écriture y trahit un défaut de conception. Lisez-le et répondez à une seule question : qu'affiche la dernière ligne ?

```php
<?php

declare(strict_types=1);

class Inventory
{
    /** @var string[] */
    public array items = [];
}

class Hero
{
    public function __construct(
        public readonly string name,
        public Inventory inventory,
    ) {}
}

sac = new Inventory();
arthur=newHero('Arthur',arthur = new Hero('Arthur', arthur=newHero('Arthur',sac);
morgane=newHero('Morgane',morgane = new Hero('Morgane', morgane=newHero('Morgane',sac);

arthur->inventory->items[]='E´pe´ecourte';printr(arthur→inventory→items[] = 'Épée courte';

print_r(arthur->inventory->items[]='E´pe´ecourte';printr(morgane→inventory→items);
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/relations-entre-objets

1. Une propriété typée par une classe

Jusqu'ici, vos propriétés étaient des `int`, des `string`, des `float`. Elles peuvent aussi contenir un **objet**. C'est le cœur de la POO.

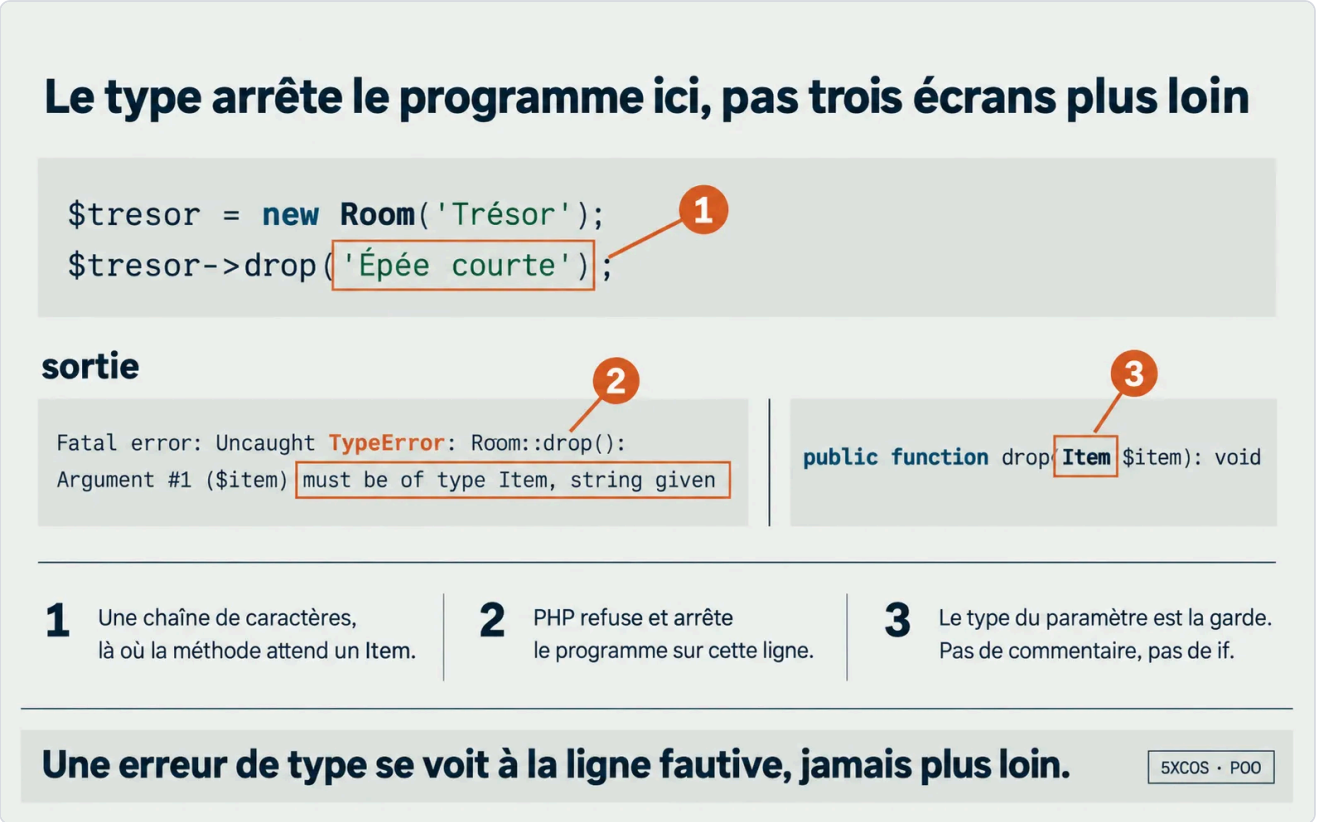
Voici une salle du Donjon, réduite à l'objet posé sur son sol et à la méthode qui le pose :

```php
class Room
{
    /** L'objet posé au sol. Au plus un, d'où le point d'interrogation. */
    public private(set) ?Item loot = null;

    public function __construct(
        public readonly string name,
    ) {}

    /** Pose un objet au sol. */
    public function drop(Item $item): void
    {
        $this->loot = $item;
    }
}
```

`?Item $loot` se lit ainsi : cette propriété contient un objet `Item`, ou `null`. Le paramètre de `drop()` porte le même type, sans le point d'interrogation : poser `null` au sol n'aurait pas de sens. Essayez de poser une chaîne de caractères à la place d'un objet :

```php
tresor=newRoom('Tre´sor');tresor = new Room('Trésor');
tresor=newRoom('Tre´sor');tresor->drop('Épée courte');
```

```
Fatal error: Uncaught TypeError: Room::drop(): Argument #1 ($item) must be of type Item,
string given
```



PHP arrête le programme sur la ligne fautive. Le bug ne va pas plus loin.

Une propriété typée par une classe **est** une relation. Le type dit avec quelle classe, le point d'interrogation dit combien. Ici, c'est le trait entre `Room` et `Item` du chapitre 4, le losange vide ◇——— avec `0..1` de chaque côté, écrit en PHP.

2. Une liste d'objets

Quand la multiplicité passe à `0..*`, la propriété devient un tableau. Le type `array` ne précise pas ce que ce tableau contient. Vous l'écrivez en commentaire, dans une forme que l'éditeur sait lire :

```php
/** @var Item[] Les objets transportés. */
private array $items = [];
```

Ce commentaire indique à l'éditeur que le tableau contient des `Item`. L'éditeur propose alors leurs méthodes dès que vous tapez `$item->`. L'analyse statique, l'outil qui relit le code sans le lancer, signale certains usages faux avant l'exécution. PHP, lui, ne lit pas ce commentaire. Ce qui protège le tableau, c'est le type du paramètre d'`add()`. Prenez quand même l'habitude de l'écrire à chaque tableau d'objets.

Voici le cœur de l'`Inventory`, le sac du héros, avec l'`Item` qu'il transporte :

PHP

```php
<?php

declare(strict_types=1);

class Item
{
    public function __construct(
        public readonly string name,publicreadonlyfloatname,
        public readonly float name,publicreadonlyfloatweight,
    ) {
        if (weight < 0) {
            throw new \InvalidArgumentException("Un poids n'est pas négatif, $weight
reçu.");
        }
    }
}

class Inventory
{
    /** @var Item[] Les objets transportés. */
    private array items = [];

    public function __construct(
        public readonly float maxWeight=20.0,)/
**Ajouteunobjet,saufsilesacde´borde.*/publicfunctionadd(ItemmaxWeight = 20.0,
    ) {}

    /** Ajoute un objet, sauf si le sac déborde. */
    public function add(Item maxWeight=20.0,)/
**Ajouteunobjet,saufsilesacde´borde.*/publicfunctionadd(Itemitem): bool
    {
        if (this->totalWeight()+this→totalWeight() + this->totalWeight()+item→weight >
this->maxWeight)returnfalse;this→maxWeight) {
            return false;
        }

        this->maxWeight)returnfalse;this→items[] = item;

        return true;
    }

    /** Le nombre d'objets dans le sac. */
    public function count(): int
    {
        return count(this→items);
    }

    /** La somme des poids, en kilos. */
    public function totalWeight(): float
    {
        total=0.0;foreach(total = 0.0;
        foreach (total=0.0;foreach(this→items as item) {
            $total += $item→weight;
        }
```

```
        return total;
    }
}
```

- `$this->items[] = $item;` ajoute l'objet à la fin du tableau. Aucune écriture dans `$items` ne vient de l'extérieur : la propriété est `private`.
- `add()` renvoie `false` quand le sac déborde, et n'ajoute rien. C'est provisoire. Si le programme ne regarde pas cette valeur, le refus passe inaperçu. Au chapitre 6, une exception signalera l'échec à la place.
- Le contrôle du poids vit dans `add()`, à un seul endroit. Personne ne peut le contourner : il n'y a pas d'autre chemin vers la liste.

Il manque `has()` et `remove()`. Vous les écrirez au kata.

Le héros fabrique son sac dans son constructeur, à la section 3. Essayez :

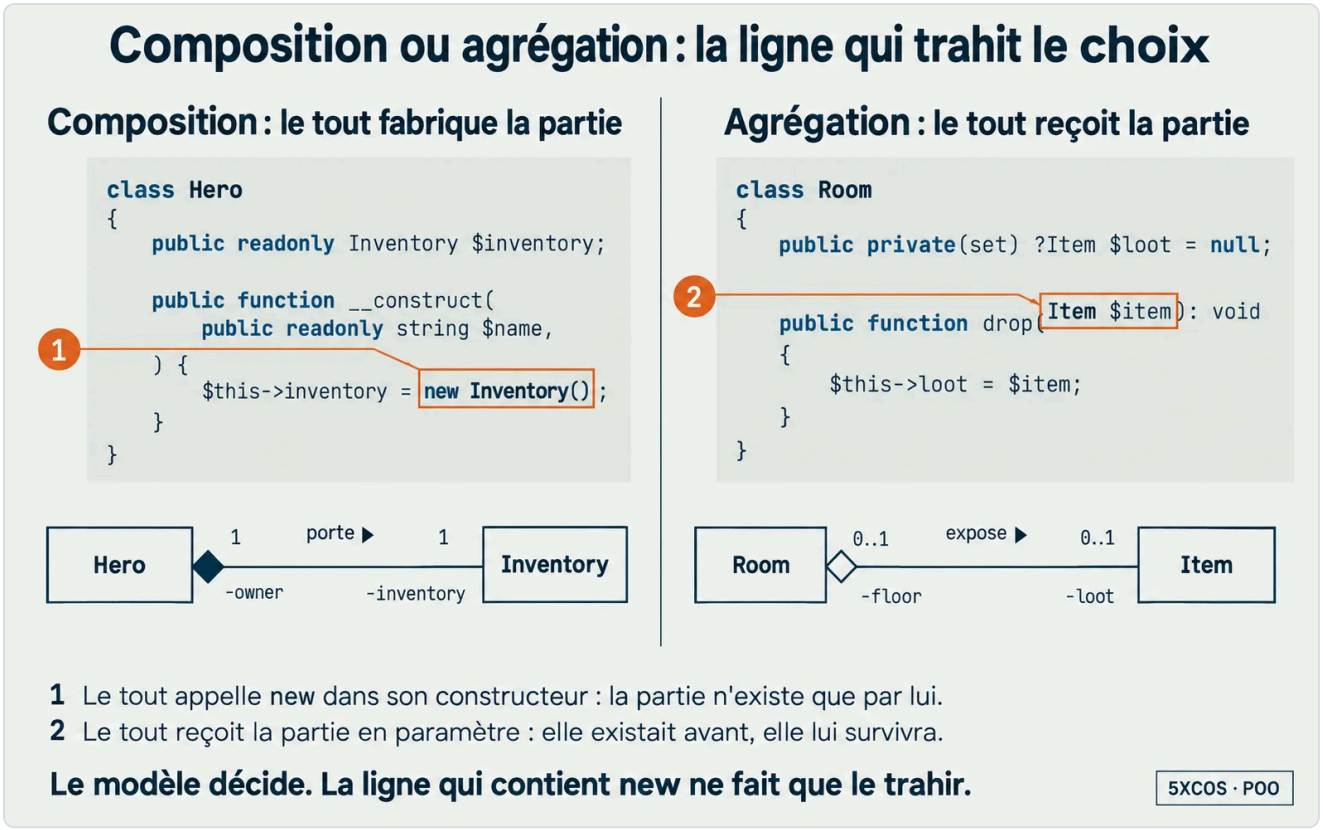
```php
arthur=newHero('Arthur');arthur = new Hero('Arthur');
arthur=newHero('Arthur');arthur->inventory->add(new Item('Épée courte', 2.0));
arthur->inventory->add(newItem('Potion',0.5));echo'Enclumeaccepte´e?',varexport(arthur
->inventory->add(new Item('Potion', 0.5));

echo 'Enclume acceptée ? ',
var_export(arthur->inventory->add(newItem('Potion',0.5));echo'Enclumeaccepte´e?',vare
xport(arthur->inventory->add(new Item('Enclume', 80.0)), true), "\n";
echo arthur->inventory->count(),'objets,',arthur->inventory->count(), ' objets, ',
arthur->inventory->count(),'objets,',arthur->inventory->totalWeight(), " kg\n";
```

```
Enclume acceptée ? false
2 objets, 2.5 kg
```

L'enclume est refusée : 2,5 plus 80 dépasse les 20 kg du sac.

3. Créer ou recevoir : la ligne qui trahit le choix

Le chapitre 4 a posé les deux relations. La **composition** dit que la partie est fabriquée par le tout et n'appartient qu'à lui. L'**agrégation** dit que la partie vient d'ailleurs et lui survit. C'est le modèle du jeu qui tranche, pas le code. Une fois la décision prise, un indice la donne : **qui appelle `new` ?**



Composition : le tout fabrique la partie

PHP

```php
class Hero
{
    /** Le sac : créé ici, et nulle part ailleurs. Lisible de partout, jamais remplacé. */
    public readonly Inventory inventory;publicprivate(set)intinventory;

    public private(set) int inventory;publicprivate(set)intmaxHp;

    public private(set) int hp = 0 {
        set => max(0, min($this->maxHp, $value));
    }

    public function __construct(
        public readonly string name,
        int maxHp=10,publicreadonlyintmaxHp = 10,
        public readonly int maxHp=10,publicreadonlyintstrength = 2,
    ) {
        this->maxHp=this->maxHp = this->maxHp=maxHp;
        this->hp=this->hp = this->hp=maxHp;
        $this->inventory = new Inventory();
    }

    // isFullHealth, les deux validations du constructeur, takeDamage(), heal(),
    // isAlive() et __toString() : inchangés depuis le chapitre 3.
}
```

Cette classe `Hero` est celle du chapitre 3, avec une propriété de plus : `inventory`. Le hook de `hp` borne toujours les points de vie.

Le `new Inventory()` est **dans le constructeur**. Trois conséquences suivent. Personne ne peut donner à un héros un sac déjà rempli. Deux héros ne peuvent pas recevoir le même sac. Et le sac n'a aucune raison d'exister ailleurs que chez son héros. C'est le losange plein ◆——— entre `Hero` et `Inventory` du chapitre 4 : un héros porte un sac, et un seul.

Le code d'ouverture faisait l'inverse : il recevait le sac en paramètre. D'où deux héros avec la même épée.

« La partie meurt avec le tout » est une règle du modèle, pas une garantie du langage. PHP libère un objet quand plus aucune variable ne le désigne. Si vous écrivez $sac = \texttt{arthur->inventory;}$ puis que vous oubliez Arthur, le sac reste accessible par `$sac`. Le losange plein dit une intention. C'est à votre code de la tenir.

Agrégation : le tout reçoit la partie

Une épée posée au sol n'appartient pas à la salle. Elle a été fabriquée ailleurs, elle passera dans le sac du héros, et ce sera toujours la même épée :

```php
class Room
{
    /** L'objet posé au sol : reçu de l'extérieur, et parfois absent. */
    public private(set) ?Item loot = null;

    public function __construct(
        public readonly string name,
    ) {}

    /** Pose un objet au sol. Vue à la section 1. */
    public function drop(Item $item): void
    {
        $this->loot = $item;
    }

    /** Retire l'objet du sol et le rend. */
    public function take(): ?Item
    {
        $item = $this->loot;
        $this->loot = null;

        return $item;
    }

    public function describe(): string
    {
        return $this->name.' : '.($this->loot?->name ?? 'rien au sol');
    }
}
```

Aucun `new Item` dans cette classe. La salle reçoit un objet fabriqué ailleurs, elle le garde un moment, elle le rend. C'est le losange vide.

Le donjon, lui, **fabrique** ses salles :

```php
class Dungeon
{
    /** @var Room[] Les salles, fabriquées ici. Lisibles de partout, modifiées seulement ici. */
    public private(set) array rooms=[];/**@paramstring[]rooms = [];

    /** @param string[] rooms=[];/**@paramstring[]names */
    public function __construct(array names)
    {
        foreach ($names as $name) {
            $this→rooms[] = new Room($name);
        }
    }

    public function room(int index): Room
    {
        return this->rooms[this→rooms[this->rooms[index];
    }
}
```

Un `Dungeon` se construit avec une liste de noms, pas avec une liste de salles. Personne ne peut lui passer une salle déjà peuplée. C'est une composition, comme le sac.

Le code :

```php
dungeon=newDungeon(['Entre´e','Salledesgardes','Tre´sor']);dungeon = new Dungeon(['Entrée', 'Salle des gardes', 'Trésor']);

dungeon=newDungeon(['Entre´e','Salledesgardes','Tre´sor']);epee = new Item('Épée courte', 2.0);
dungeon->room(1)->drop(dungeon→room(1)→drop(dungeon->room(1)->drop(epee);

foreach (dungeon->roomsasdungeon→rooms as dungeon->roomsasroom) {
    echo room→describe(), "\n";
}

echo 'même objet que epee ? ', var_export(dungeon->room(1)->loot⩵⩵dungeon→room(1)→loot ⩵⩵ dungeon->room(1)->loot⩵⩵epee, true), "\n";
```

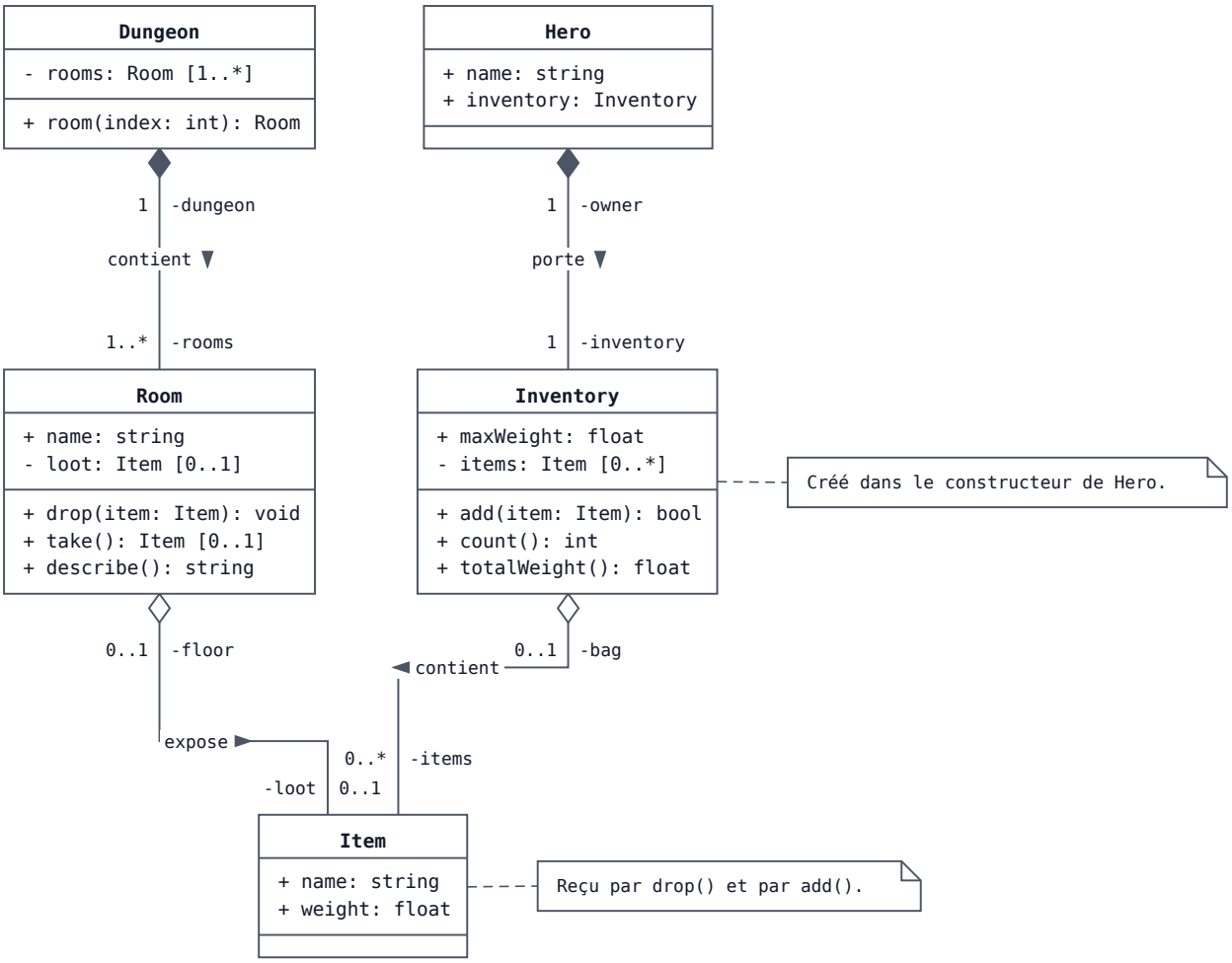
```
Entrée : rien au sol
Salle des gardes : Épée courte
Trésor : rien au sol
même objet que $epee ? true
```

La dernière ligne est le point important. La salle ne détient pas une copie de l'épée : elle détient **votre** épée. Sa propriété `loot` ne contient pas l'objet lui-même, mais une **référence** : de quoi retrouver l'objet en mémoire. Le héros peut donc ramasser l'épée sans qu'elle change d'identité :

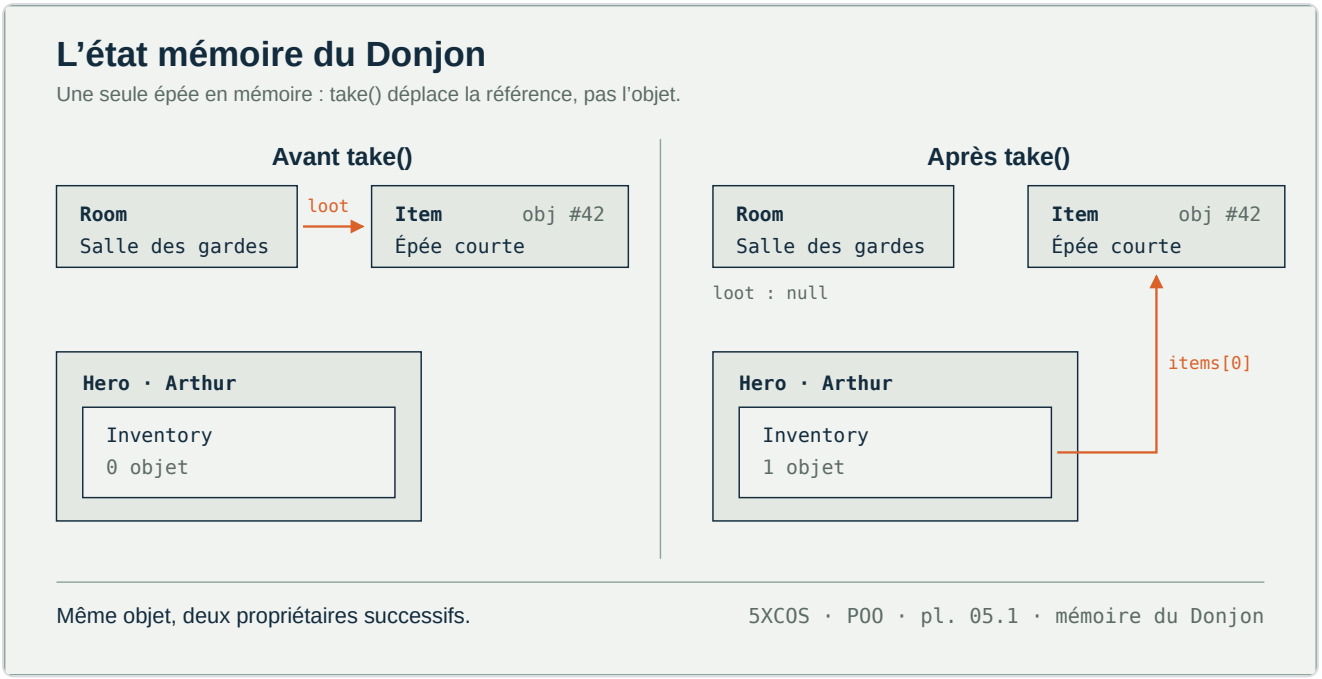
```php
arthur=newHero(’Arthur’);arthur = new Hero('Arthur');
arthur=newHero(’Arthur’);ramasse = dungeon->room(1)->take();if(dungeon→room(1)→take();

if (dungeon->room(1)->take();if(ramasse !== null) {
    arthur->inventory->add(arthur→inventory→add(arthur->inventory->add(ramasse);
}

echo dungeon->room(1)->describe(),"\n";echodungeon→room(1)→describe(), "\n";
echo dungeon->room(1)->describe(),"\n";echoarthur→inventory→count(), " objet dans le
sac\n";
echo 'toujours la même épée ? ', var_export(ramasse===ramasse === ramasse===epee, true),
"\n";
```

```
Salle des gardes : rien au sol
1 objet dans le sac
toujours la même épée ? true
```

Le modèle du Donjon, tel qu'il est codé ci-dessus :



Chaque relation se lit deux fois : sur le trait, et comme propriété dans la boîte du côté qui la porte. `rooms`, `loot` et `items` sont notées `-` parce que seule leur classe les écrit. La boîte `Hero` ne montre que `name` et `inventory`. Ses points de vie et ses méthodes du chapitre 3 existent toujours, le diagramme les laisse de côté pour ne garder que les relations.



Un héros **est fait d'un** sac : il le fabrique, et le sac n'a pas de sens sans lui. Composition. Une salle **a** une épée : l'épée existait avant. Agrégation.

4. Deux variables, un seul objet

Avec les types de base, PHP copie la valeur :

```php
a=5;a = 5;
a=5;b = a;a;
a;b = 10;

echo $a, "\n";
```

```
5
```

Avec les objets, non. Une variable ne contient pas l'objet : elle contient une référence, de quoi le retrouver. L'affectation recopie cette référence, pas l'objet.

```php
arthur=newHero(’Arthur’);arthur = new Hero('Arthur');
arthur=newHero(’Arthur’);leHeros = arthur;arthur;

arthur;leHeros→takeDamage(3);

echo arthur,"\n";echoarthur, "\n";
echo arthur,"\n";echoleHeros, "\n";
```

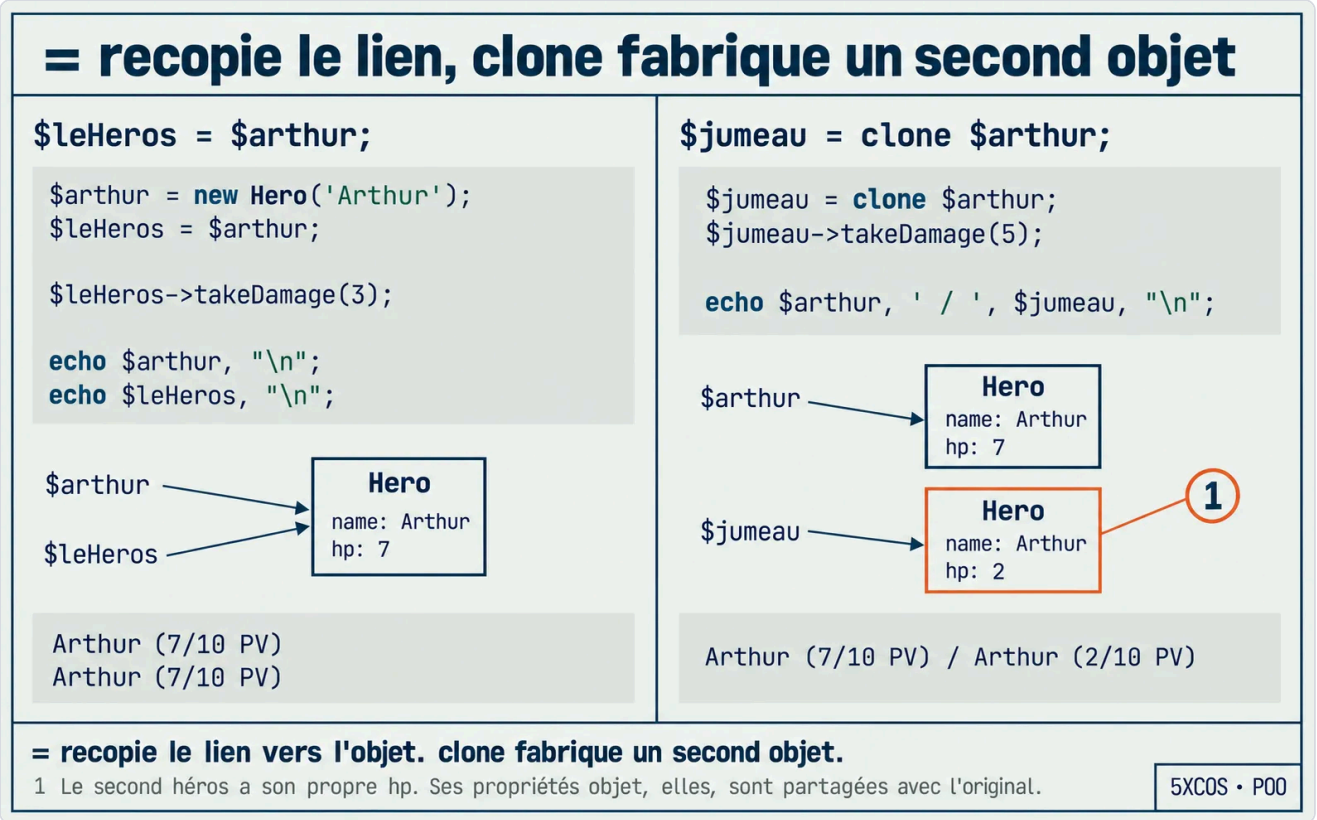
```
Arthur (7/10 PV)
Arthur (7/10 PV)
```

Vous avez blessé Arthur en passant par `$leHeros` : il n'y a jamais eu qu'un seul héros. C'est l'explication du code d'ouverture, et c'est aussi ce qui rend l'agrégation possible. L'épée de la salle **est** celle de votre variable, pas une copie.

Pour obtenir un second objet, il faut le demander avec `clone` :

```php
jumeau=clonejumeau = clone jumeau=clonearthur;
jumeau->takeDamage(5);echojumeau→takeDamage(5);

echo jumeau->takeDamage(5);echoarthur, ' / ', $jumeau, "\n";
```

```
Arthur (7/10 PV) / Arthur (2/10 PV)
```



`clone $arthur` recopie les propriétés une à une. Or `inventory` contient une référence : la copie reçoit la même référence, donc le même sac. Vérifiez :

```php
arthur=newHero('Arthur');arthur = new Hero('Arthur');
arthur=newHero('Arthur');arthur→inventory→add(new Item('Épée courte', 2.0));

jumeau=clonejumeau = clone jumeau=clonearthur;
jumeau->inventory->add(newItem('Potion',0.5));echo'me^mesac?',varexport(jumeau→inventory
→add(new Item('Potion', 0.5));

echo 'même sac ? ', var_export(jumeau->inventory->add(newItem('Potion',0.5));echo'me^mesac?
',varexport(jumeau→inventory === arthur->inventory,true),"\n";echo'Arthur:',arthur
→inventory, true), "\n";
echo 'Arthur : ', arthur->inventory,true),"\n";echo'Arthur:',arthur→inventory→count(), ' /
jumeau : ', $jumeau→inventory→count(), "\n";
```

```
même sac ? true
Arthur : 2 / jumeau : 2
```

La potion du jumeau a atterri chez Arthur. Pour que la copie ait son propre sac, la classe écrit la méthode `__clone()`. PHP l'appelle sur la copie, juste après l'avoir fabriquée :

```php
class Hero
{
    // ... tout le reste ne change pas.

    /** Appelée sur la copie : le sac est recopié, pas partagé. */
    public function __clone(): void
    {
        this->inventory=clonethis→inventory = clone this->inventory=clonethis→inventory;
    }
}
```

Le même script affiche maintenant :

```
même sac ? false
Arthur : 1 / jumeau : 2
```

Cette copie s'appelle une **copie profonde** : chaque héros a son sac. Écrire dans la propriété `readonly` est permis ici, et seulement ici : `__clone()` termine la fabrication d'un objet neuf.

Et les objets dans le sac ? `clone $this→inventory` recopie le tableau `$items`, mais chaque case contient une référence : les deux sacs pointent vers la même épée. Ici, rien ne gêne. Un `Item` est entièrement `readonly`, personne ne peut le modifier, donc le partager ne change rien. Si vos objets changeaient avec le temps, une arme qui s'use, une potion à moitié bue, le partage deviendrait un bug. `Inventory` écrirait alors son propre `__clone()`, qui clone chaque objet du tableau. La copie profonde s'arrête là où les objets ne bougent plus.

`==` ou `===` sur des objets

- `==` compare le **contenu** : même classe, et des propriétés qui se valent une à une.
- `===` compare l'**identité** : est-ce le même objet ?

```php
arthur=newHero('Arthur');arthur = new Hero('Arthur');
arthur=newHero('Arthur');leHeros = arthur;arthur;
arthur;leHeros->takeDamage(3);

sosie=newHero('Arthur');sosie = new Hero('Arthur');
sosie=newHero('Arthur');sosie->takeDamage(3);

echo 'arthur === leHeros : ', var_export(arthur===arthur === arthur===leHeros, true), "\n";
echo 'arthur === sosie   : ', var_export(arthur===arthur === arthur===sosie, true), "\n";
echo 'arthur == sosie    : ', var_export(arthur==arthur == arthur==sosie, true), "\n";
```

```
arthur === leHeros : true
arthur === sosie   : false
arthur == sosie    : true
```

Le sosie porte le même nom et les mêmes points de vie, donc `==` répond `true`. C'est pourtant un autre objet, donc `===` répond `false`. Utilisez `===` pour vérifier que deux variables désignent le même objet : retrouver l'exemplaire emprunté par un membre, retirer cette épée-là du sol.

> **ATTENTION**
>
> Sur deux objets, `===` ne compare pas les valeurs, contrairement à ce qu'il fait sur deux chaînes ou deux entiers. Deux héros identiques mais distincts donnent `false`. Pour comparer du contenu, comparez ce qui identifie vos objets, par exemple $a->name === b->name$.

5. Quand la relation est facultative

Une multiplicité `0..1` en UML devient un type **nullable** en PHP, c'est-à-dire un type qui accepte aussi la valeur `null`. Vous avez écrit `?Item $loot` au chapitre 3. La salle du Donjon porte `public private(set) ?Item $loot = null`, et sa méthode `take()` renvoie un `?Item`.

Lire une propriété nullable demande une précaution :

```php
room=newRoom('Tre´sor');vardump(room = new Room('Trésor');
var_dump(room=newRoom('Tre´sor');vardump(room->loot->name);
```

```
Warning: Attempt to read property "name" on null
NULL
```

`$room->loot->name` déclenche un avertissement dès que le sol est vide, et ne rend rien. L'opérateur `?->`, appelé *nullsafe*, évite ce test :

```php
return this->name.':'.(this->name.' : '.(this->name.':'.(this->loot?->name ?? 'rien au sol');
```

`$this->loot?->name` se lit ainsi : si `loot` vaut `null`, l'expression entière vaut `null` et `name` n'est jamais lue. Il fonctionne aussi devant une méthode : `$this->loot?->describe()`. L'opérateur `??` prend alors le relais et fournit la valeur de remplacement. En trois caractères, vous évitez un `if` de trois lignes.

> **NOTE**
>
> **Histoire de PHP.** L'opérateur `?->` est arrivé avec PHP 8.0, en 2020. Avant, il fallait écrire le test à la main : `if ($x !== null) { $x->m(); }`, ou sa version courte $x?x->m() : null. Vous croiserez encore ces deux écritures dans les projets existants. Elles font le même travail, en plus long.

À l'usage :

```php
tresor=newRoom('Tre´sor');echotresor = new Room('Trésor');
echo tresor=newRoom('Tre´sor');echotresor->loot?->name ?? 'rien au sol', "\n";

tresor->drop(newItem('Potion',0.5));echotresor->drop(new Item('Potion', 0.5));
echo tresor->drop(newItem('Potion',0.5));echotresor->loot?->name ?? 'rien au sol', "\n";
```

```
rien au sol
Potion
```

> **ASTUCE**
>
> `?->` ne dispense pas de vérifier. Il convient pour un affichage ou une valeur de remplacement. Quand l'absence est une erreur du métier, un prêt sans exemplaire, une commande sans client, levez plutôt une exception : c'est le chapitre 6. Sinon l'erreur n'apparaît que beaucoup plus loin dans le programme, là où elle devient difficile à comprendre.

6. Le pont : ce que vous verrez dans Laravel

Dans la partie Framework du cours, vous écrirez ceci :

```php
post=Post::find(1);foreach(post = Post::find(1);

foreach (post=Post::find(1);foreach(post→comments as $comment) {
    echo $comment→body;
}
```

`Post::find(1)` va chercher l'article numéro 1 dans la base de données et le rend sous forme d'objet. `$post→comments` donne la liste de ses objets `Comment`. C'est la structure de ce chapitre : une liste d'objets détenue par un objet, comme `$items` dans `Inventory`. Une différence : Laravel remplit cette liste depuis la base de données, vous n'écrivez aucun `new`.

Ce code a le défaut vu à la section 5. `find()` renvoie `null` quand aucun article ne porte ce numéro, et `$post→comments` lève alors l'avertissement `Attempt to read property "comments" on null`. Deux corrections possibles :

```php
// Rien à afficher si l'article n'existe pas.
post=Post::find(1);foreach(post = Post::find(1);

foreach (post=Post::find(1);foreach(post?→comments ?? [] as comment) {
    echo $comment→body;
}

// Ou : l'article doit exister, sinon Laravel renvoie une page 404.
post = Post::findOrFail(1);
```

Dans la documentation de Laravel, ces relations portent des noms : `hasMany`, `belongsTo`, `hasOne`. « Un `Post` *has many* `Comment` » est la relation `Inventory` / `Item` de ce chapitre : un trait avec `0..*` dans le diagramme, une propriété qui contient une liste dans la classe.

À retenir

- Une propriété typée par une classe **est** une relation. `?Item $loot` écrit la multiplicité `0..1`, et `/** @var Item[] */` renseigne l'éditeur sur un tableau d'objets.
- **Fabriqué dans le constructeur, c'est une composition** : losange plein, exclusif. **Reçu en paramètre, c'est une agrégation** : losange vide, partageable.
- Une variable ne contient pas l'objet, mais une référence vers lui. `b =a` ne fabrique pas de second objet. `clone` le fait, et `__clone()` décide ce que la copie partage.
- `==` compare le contenu, `===` compare l'identité. Sur des objets, prenez `===`.
- `?→` remplace le test `if ($x === null)` pour un affichage ou une valeur de remplacement, jamais pour cacher une erreur du métier.

Exercices

Sur papier

Trois questions à traiter sans ordinateur, avant d'ouvrir le kata.

1. Un `Dungeon` reçoit ses salles en paramètre au lieu de les fabriquer. Quel losange dessinez-vous alors, et que se passe-t-il si deux donjons reçoivent la même salle ?
2. Reprenez le diagramme de la section 3. Écrivez la propriété PHP qui traduit chacune de ses trois relations à losange : la salle et son butin ◇———, le héros et son sac ◆———, le sac et ses objets ◇———. Donnez à chaque fois la visibilité, le type et la valeur de départ.
3. Le héros expose son sac par la propriété `inventory`. N'importe quelle partie du programme peut donc écrire `$arthur->inventory->add(...)` sans passer par le héros. Est-ce compatible avec la composition ? Que changeriez-vous pour que seul le héros remplisse son sac ?

Kata : niveau 2

Dépôt : https://github.com/opmvpc/poo-katas-26. Lancez seulement ce niveau :

```bash
composer test -- --group=niveau-2
```

Ce que les tests attendent :

Classe	Signature	Comportement vérifié
`Item`	`__construct(string` $name$, $float$ `weight, Rarity $rarity = Rarity::Common)`	`$item->name` et `$item->weight` rendent ce qui a été passé au constructeur. Un poids négatif lève une `InvalidArgumentException`
`Inventory`	`__construct(float $maxWeight = 20.0)`	un sac neuf : `count()` vaut `0`, `totalWeight()` vaut `0.0`, `$bag->maxWeight` vaut `20.0`
`Inventory`	`add(Item $item): bool`	ajoute l'objet. Si le poids total dépassait `maxWeight`, n'ajoute rien et renvoie `false`
`Inventory`	`has(string $name): bool`	`true` pour un objet présent, `false` pour un objet absent
`Inventory`	`remove(string $name): void`	retire le premier objet portant ce nom. Ne fait rien si ce nom est absent, et `count()` reste juste
`Inventory`	`totalWeight(): float`	la somme des poids : `2.0` et `0.5` donnent `2.5`
`Hero`	`public readonly Inventory $inventory`	`$hero->inventory` existe dès la construction du héros, et deux héros n'ont jamais le même

Le test du sac plein accepte aussi une exception, le comportement du niveau 3. Rien ne s'écrit de l'extérieur, comme au chapitre 3 : `name`, `weight` et `maxWeight` sont `public readonly`, `inventory` aussi. Vous les lisez, `$bag->maxWeight`, vous ne les écrivez jamais. Aucune méthode `getName()`, aucune méthode `setName()`.

Le squelette d'`Item` commence ainsi :

```php
abstract class Item implements \Stringable
{
    public function __construct(
        public readonly string name,publicreadonlyfloatname,
        public readonly float name,publicreadonlyfloatweight,
        public readonly Rarity $rarity = Rarity::Common,
    ) {
        throw new \LogicException('À implémenter');
    }
```

`Item` y est déjà `abstract` et son constructeur prend un troisième paramètre, `Rarity $rarity` : c'est sa forme des niveaux suivants. Au niveau 2, vous complétez ce constructeur avec la validation du poids, puis les cinq méthodes d'`Inventory`. Comme une classe abstraite ne s'instancie pas, les tests passent par `Tests\Support\SimpleItem`, une sous-classe fournie dans `tests/Support/`. Vous n'y touchez pas.

Le seul piège du niveau tient en un test :

```php
arthur=newHero(’Arthur’);arthur = new Hero('Arthur');
arthur=newHero(’Arthur’);morgane = new Hero('Morgane');

arthur->inventory->add(newSimpleItem(’E´pe´ecourte’,2.0));expect(arthur→inventory→add(new
SimpleItem('Épée courte', 2.0));

expect(arthur->inventory->add(newSimpleItem(’E´pe´ecourte’,2.0));expect(arthur→inventory
→count())→toBe(1);
expect($morgane→inventory→count())→toBe(0);
```

Si le constructeur reçoit l'inventaire en paramètre, ou si les deux héros le partagent d'une autre façon, le test passe au rouge.

Le niveau 3 reprendra les mêmes relations avec l'arme du héros, `equip(Weapon $weapon): void` et une propriété `public private(set) ?Weapon $weapon`. C'est une relation `0..1`, comme l'épée au sol. Il demande aussi l'héritage : il attend donc le chapitre 6.

2.6 Héritage et polymorphisme

Séance 3 · ~2 périodes

Slides du chapitre (PDF) PDF

Vos classes se ressemblent. Une épée et une potion, c'est deux fois « un nom, un poids ». Un gobelin et un dragon, c'est deux fois « un nom, des points de vie, une attaque ». Ce chapitre donne deux mécanismes. Le premier met le code commun à un seul endroit. Le second parcourt trente monstres différents sans un seul `if`. Dans Laravel, `class UserController extends Controller` utilise le premier.

À la fin de ce chapitre, vous serez capables de :

1. **écrire** une hiérarchie de classes avec `extends` et `parent::__construct()`, et dire à quoi sert `protected` ;
2. **déclarer** une classe et une méthode **abstraites**, et dire ce que l'abstrait interdit ;
3. **utiliser** le polymorphisme : une liste d'objets de types différents, un seul appel de méthode ;
4. **choisir** entre hériter et posséder grâce au test du « est-un » ;
5. **créer** une exception personnalisée et l'attraper avec un `catch` ciblé.

Le code qui sent mauvais

Cinq minutes, en binôme. Ce code sent mauvais : un détail d'écriture y trahit un défaut de conception. Lisez-le et répondez à une seule question : **que faudra-t-il modifier pour ajouter un squelette ?**

```php
class Monster
{
    public function __construct(private string type,privateinttype, private int
type,privateinthp) {}

    public function attack(): int
    {
        if (this->type===’goblin’)return2;elseif(this→type === 'goblin') {
            return 2;
        } elseif (this->type===’goblin’)return2;elseif(this→type === 'dragon') {
            return 8;
        }

        return 0;
    }
}
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/heritage-et-polymorphisme

1. Deux classes qui se ressemblent trop

Au chapitre 5, `Item` avait un nom et un poids. Le jeu a maintenant besoin d'armes, qui infligent des dégâts, et de potions, qui soignent. La tentation est d'écrire deux classes complètes :

```php
final class Weapon
{
    public function __construct(
        public readonly string name,publicreadonlyfloatname,
        public readonly float name,publicreadonlyfloatweight,
        public readonly int damage,
    ) {
        if ($weight < 0) {
            throw new \InvalidArgumentException("Un poids n'est pas négatif, $weight
reçu.");
        }
    }
}

final class Potion
{
    public function __construct(
        public readonly string name,
        public readonly float weight,publicreadonlyintweight,
        public readonly int weight,publicreadonlyinthealing,
    ) {
        if ($weight < 0) {
            throw new \InvalidArgumentException("Un poids n'est pas négatif, $weight
reçu.");
        }
    }
}
```

Deux problèmes. Le premier est la duplication : le nom, le poids et sa validation sont écrits deux fois. Le jour où un objet gagne une durabilité, vous modifiez deux fichiers, puis trois. Le second est plus gênant. Sans classe commune, `Inventory::add()` doit énumérer les classes acceptées, et cette liste s'allonge à chaque nouvel objet du jeu.

L'**héritage** répond aux deux. `Item` porte le nom et le poids. `Weapon` ajoute les dégâts, `Potion` ajoute les soins. Chaque classe fille ne décrit que sa différence.

2. `extends`, `protected`, `parent::`

Voici `Item`, puis `Weapon` qui en hérite :

```php
abstract class Item
{
    public function __construct(
        public readonly string name,publicreadonlyfloatname,
        public readonly float name,publicreadonlyfloatweight,
    ) {
        if (weight < 0) {
            throw new \InvalidArgumentException("Un poids n'est pas négatif, $weight
reçu.");
        }
    }

    /** Chaque objet se décrit à sa façon : le parent ne peut pas répondre. */
    abstract public function describe(): string;
}

final class Weapon extends Item
{
    public function __construct(string name, float weight,publicreadonlyintweight, public
readonly int weight,publicreadonlyintdamage)
    {
        parent::__construct(name,name, name,weight);
    }

    public function describe(): string
    {
        return sprintf('%s : arme (%s kg, %d dégâts)', this->name,this→name,
this->name,this→weight, $this→damage);
    }
}
```

`extends` déclare que `Weapon` est un cas particulier d'`Item`. Elle en hérite toutes les propriétés et méthodes non privées : `name`, `weight` et la validation du poids, écrites une seule fois. Le mot `abstract`, sur la classe et sur `describe()`, est expliqué à la section 3.

`parent::__construct()` appelle le constructeur du parent. Une classe fille qui définit son propre constructeur doit faire cet appel elle-même. Sans lui, les propriétés du parent ne reçoivent aucune valeur, et la première lecture arrête le programme :

```
Fatal error: Uncaught Error: Typed property Item::$name must not be accessed before
initialization
```

C'est l'erreur la plus fréquente du chapitre. Une classe fille sans constructeur utilise directement celui du parent.

`protected` est la troisième visibilité, annoncée au chapitre 1. Une propriété `protected` est accessible dans sa classe et dans ses classes filles.

Visibilité	Accessible depuis…	À utiliser quand…
`public`	n'importe où	c'est le mode d'emploi de la classe

Visibilité	Accessible depuis…	À utiliser quand…
`protected`	la classe **et ses filles**	les filles doivent écrire dans la donnée
`private`	la classe seule	même les filles n'ont pas à y toucher

Ici, `Weapon::describe()` ne fait que lire `name` et `weight` : les deux restent `public readonly`, comme au chapitre 5. Aucune classe fille n'a besoin d'y écrire, donc `protected` ne sert pas. Choisissez toujours la visibilité la plus fermée qui suffit. `protected` attend le jour où une classe fille doit écrire dans une donnée du parent.

`Potion` est bâtie sur le même modèle, avec `healing` à la place de `damage` :

```php
final class Potion extends Item
{
    public function __construct(string name,floatname, float name,floatweight, public
readonly int $healing)
    {
        parent::__construct($name, $weight);
    }

    public function describe(): string
    {
        return sprintf('%s : potion (%s kg, +%d PV)', $this→name, $this→weight, $this
→healing);
    }
}
```

```php
echo (new Weapon('Épée courte', 2.0, 5))→describe(), PHP_EOL;
echo (new Potion('Potion de soin', 0.5, 5))→describe(), PHP_EOL;
```

```
Épée courte : arme (2 kg, 5 dégâts)
Potion de soin : potion (0.5 kg, +5 PV)
```

ASTUCE

`final` devant une classe interdit d'écrire une classe qui en hérite. Devant une méthode, il interdit de la redéfinir. Mettez-le sur les classes du bas de la hiérarchie, comme `Weapon` et `Potion`.

3. Abstrait : une classe qui ne se fabrique pas

Dans le Donjon, un objet ramassable est toujours une arme ou une potion. Aucun corps d'`Item::describe()` n'aurait de sens : la description dépend du type d'objet. Le mot-clé `abstract` a deux emplois.

- `abstract class Item` : cette classe **ne s'instancie pas**. Elle n'existe que pour être héritée.
- `abstract public function describe(): string;` : une **signature sans corps**. Toute classe fille concrète doit la fournir, sinon PHP refuse de charger le fichier.

```php
$item = new Item('Caillou', 1.0);
```

```
Fatal error: Uncaught Error: Cannot instantiate abstract class Item
```

PHP signale l'erreur dès la ligne fautive, pas plus loin sur un `describe()` vide. Une classe fille qui oublie `describe()` est refusée avant même de tourner :

```php
final class Rock extends Item
{
}
```

```
Fatal error: Class Rock contains 1 abstract method and must therefore be declared abstract
or implement the remaining method (Item::describe)
```

> **NOTE**
>
> **Histoire de PHP.** Les classes abstraites, les interfaces, les visibilités et le mot-clé `final` arrivent tous avec **PHP 5, en 2004**. Avant, la POO de PHP tenait en `class` et `extends`, tout était public et se déclarait avec `var $hp;`. Vous croiserez encore `var` dans du très vieux code : c'est un `public`.

Une classe abstraite peut contenir du code normal : des propriétés, un constructeur, des méthodes complètes. C'est ce qui la distingue d'une interface, au chapitre 7. `Monster` porte les points de vie et les dégâts encaissés, et laisse `attack()` abstraite.

PHP

```php
abstract class Monster
{
    public private(set) int maxHp;publicprivate(set)intmaxHp;

    public private(set) int maxHp;publicprivate(set)inthp = 0 {
        set ⇒ max(0, min(this->maxHp,this→maxHp, this->maxHp,value));
    }

    public function __construct(
        public readonly string name,intname,
        int name,intmaxHp,
    ) {
        this->maxHp=this→maxHp = this->maxHp=maxHp;
        this->hp=this→hp = this->hp=maxHp;
    }

    public function takeDamage(int $amount): void
    {
        $this→hp -= $amount;
    }

    public function isAlive(): bool
    {
        return $this→hp > 0;
    }

    /** Chaque monstre frappe à sa façon. Le parent ne peut pas décider. */
    abstract public function attack(): int;

    public function __toString(): string
    {
        return sprintf('%s (%d/%d PV)', $this→name, $this→hp, $this→maxHp);
    }
}

final class Goblin extends Monster
{
    public function __construct()
    {
        parent::__construct('Gobelin', 5);
    }

    public function attack(): int
    {
        return 2;
    }
}

final class Dragon extends Monster
{
    public function __construct()
    {
        parent::__construct('Dragon', 30);
    }
```

```
    public function attack(): int
    {
        return 8;
    }
}
```

Les attaques valent 2 et 8, sans hasard. Un monstre qui lance un dé est amusant, mais un test ne peut plus prévoir le résultat. Le chapitre 7 rendra le hasard maîtrisable.

`name` est `public readonly`, donc vous écrivez `$goblin->name`. `hp` et `maxHp` sont `public private(set)`, comme chez le héros. Pas de `protected` : `Goblin` et `Dragon` n'écrivent jamais dans les points de vie, le parent s'en charge. C'est la première règle du chapitre 3 : écrivez `private`, tout autre choix demande une raison. Le diagramme du chapitre 4 notait `#hp` sur `Monster`. Le code retenu ici garde `private(set)`, donc `-hp`.

Redéfinir une méthode

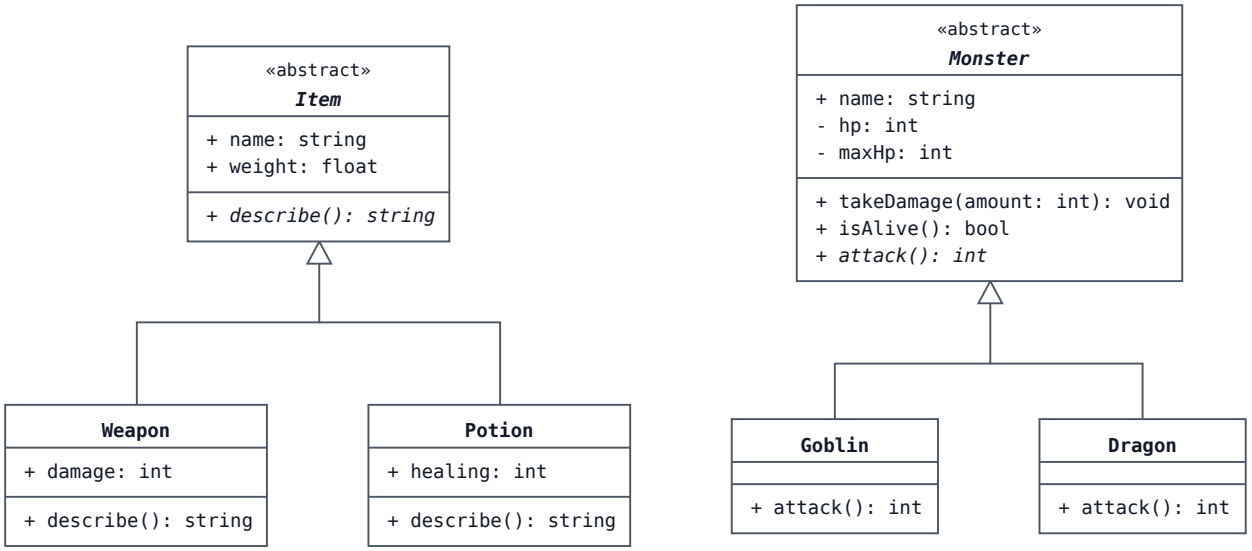
Une classe fille peut **redéfinir** une méthode du parent : elle écrit une méthode du même nom, et ce code remplace celui du parent pour les objets de la classe fille. La **signature** d'une méthode est sa ligne de déclaration : le nom, les paramètres avec leurs types, le type de retour. `public function __toString(): string` est la signature de `__toString()`. La méthode redéfinie garde la signature du parent : mêmes paramètres, même type de retour. PHP refuse le fichier sinon.

Redéfinir n'oblige pas à tout réécrire. `parent::` désigne la classe parente, et `parent::__toString()` exécute la version de `Monster`, sur le même objet : `$this` y désigne toujours le dragon. Vous connaissez déjà ce mécanisme avec `parent::__construct()`, qui appelle le constructeur du parent. La classe fille peut donc compléter la réponse du parent au lieu de la remplacer :

```php
final class Dragon extends Monster
{
    // … constructeur et attack() inchangés

    public function __toString(): string
    {
        return 'Un dragon ! ' . parent::__toString();
    }
}

echo new Goblin(), PHP_EOL;
echo new Dragon(), PHP_EOL;
```

```
Gobelin (5/5 PV)
Un dragon ! Dragon (30/30 PV)
```

Sans `parent::`, `Dragon::__toString()` aurait recopié le `sprintf()` de `Monster`. Le jour où le format change, il change à un seul endroit. Dans le kata, `Dragon` garde le `__toString()` de `Monster`.



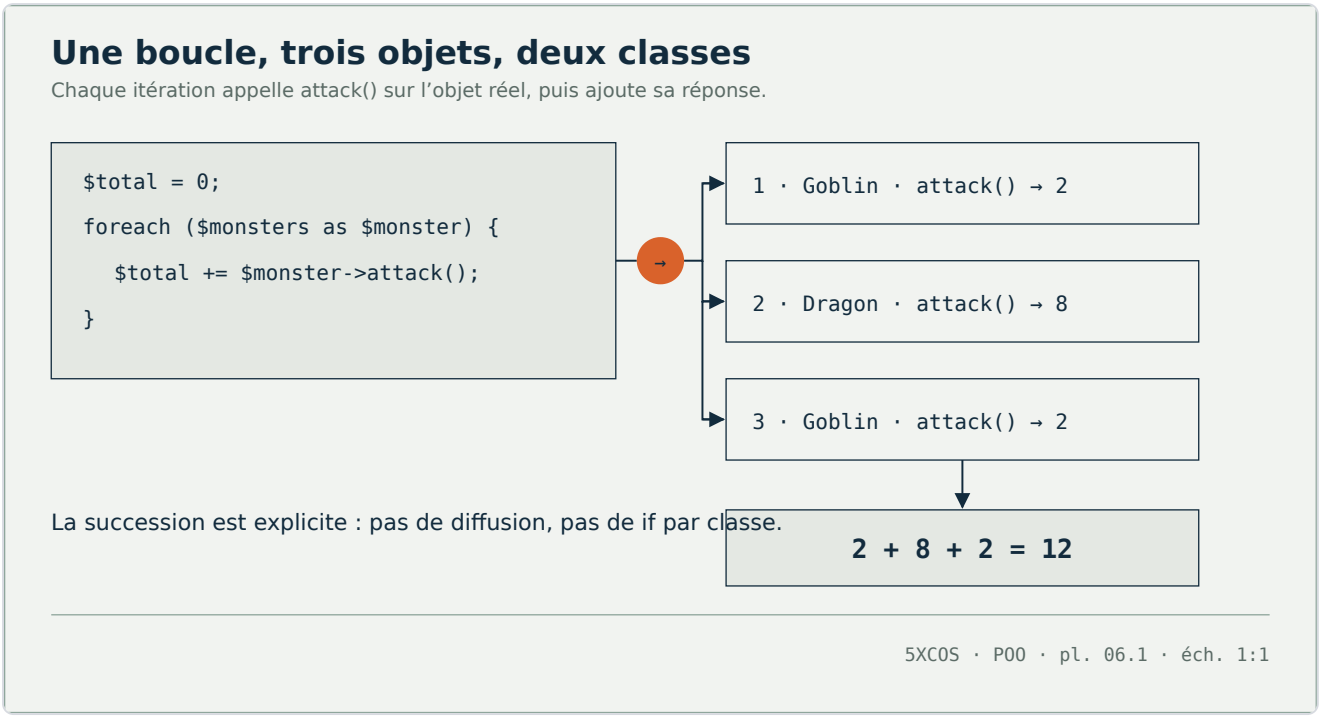
En UML, la **généralisation** ◁—— est un trait plein terminé par un triangle vide, collé au parent. Il se lit du bas vers le haut : « `Weapon` est un `Item` ». Le cours écrit le stéréotype `«abstract»` au-dessus du nom, comme au chapitre 4. Le site met en plus le nom et la méthode abstraite en italique.

4. Le polymorphisme : une boucle sans `if`

Une liste de monstres de types différents, et un seul appel de méthode.

```php
/** @var Monster[] monsters*/monsters = [new Goblin(), new Dragon(), new Goblin()];

total=0;foreach(total = 0;
foreach (total=0;foreach(monsters as monster) {
    $damage = $monster->attack();
    $total += $damage;
    printf('%s frappe pour %d.%s', $monster->name, $damage, PHP_EOL);
}
printf('Total encaissé : %d PV.%s', total, PHP_EOL);
```

```
Gobelin frappe pour 2.
Dragon frappe pour 8.
Gobelin frappe pour 2.
Total encaissé : 12 PV.
```



Comparez avec le code du début de chapitre. La cascade de `elseif` n'a pas été déplacée ailleurs : elle a disparu. Chaque monstre porte sa propre réponse à `attack()`, et la boucle ne connaît que le type `Monster`.

Le vrai gain se voit à l'ajout d'un monstre. Vous créez `Skeleton extends Monster`, vous écrivez son `attack()`, et aucune boucle existante ne change.

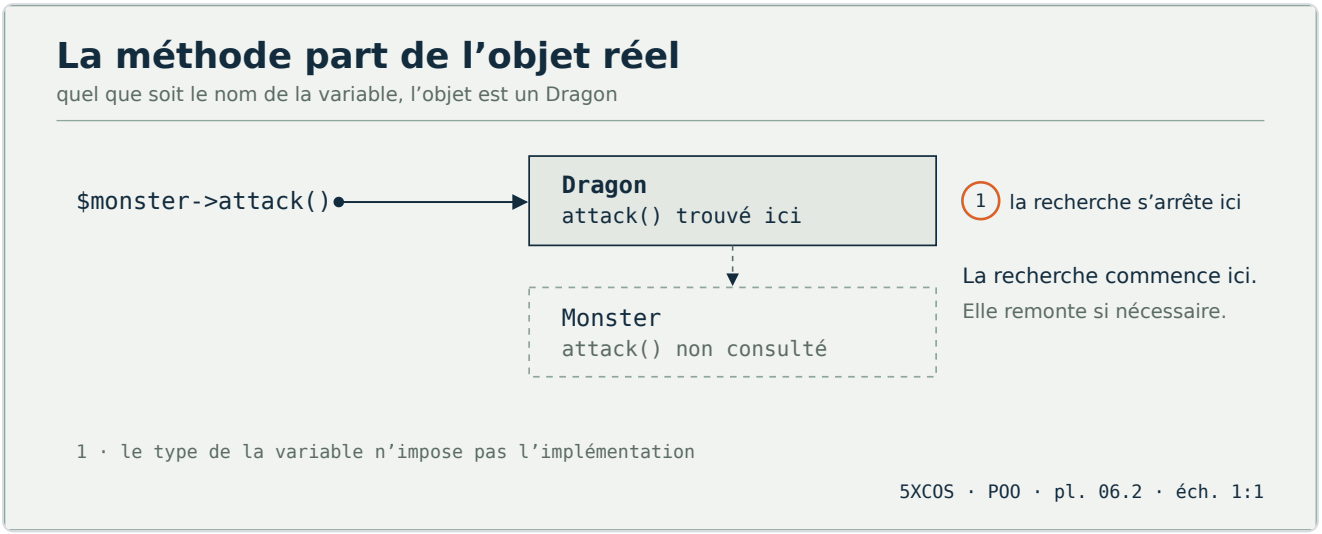
Qui décide de la méthode appelée ?

C'est l'objet, pas la variable.

```php
monster=newDragon();echogetclass(monster = new Dragon();
echo get_class(monster=newDragon();echogetclass(monster), ' attaque pour ',
monster->attack(),PHPEOL;vardump(monster→attack(), PHP_EOL;
var_dump(monster->attack(),PHPEOL;vardump(monster instanceof Monster, $monster instanceof
Goblin);
```

```
Dragon attaque pour 8
bool(true)
bool(false)
```

`get_class()` rend le nom de la classe réelle de l'objet.



À l'exécution, PHP part de la classe réelle de l'objet, ici `Dragon`, y cherche `attack()`, la trouve et l'appelle. S'il ne l'avait pas trouvée, il serait remonté à `Monster`. Si aucune classe ne la définit, le programme s'arrête : `Call to undefined method Dragon::attack()`. La recherche part toujours du bas.

`instanceof` répond à la question « cet objet est-il de ce type, ou d'un type qui en descend ? ». Il est utile pour une vérification ponctuelle. Mais une cascade de `instanceof` qui choisit l'attaque, c'est le `elseif` du début de chapitre qui revient sous un autre nom. La réponse reste la même : une méthode commune aux monstres.

5. Hériter ou posséder

Avant d'écrire `extends`, dites la phrase à voix haute : **« un X est un Y »**. Si elle est vraie sans explication, l'héritage est une piste sérieuse. Vérifiez ensuite que les propriétés et les méthodes du parent conviennent vraiment à la classe fille.

- « Une épée **est un** objet ramassable. » Juste. `Weapon extends Item`.
- « Un dragon **est un** monstre. » Juste. `Dragon extends Monster`.
- « Un héros **est une** arme. » Faux. C'est pourtant l'erreur qui arrive dès qu'une classe hérite pour récupérer du code.

Un exemple hors du Donjon :

```php
class Car
{
    public function __construct(public readonly int seats) {}

    public function openTrunk(): string
    {
        return "Le coffre s'ouvre.";
    }
}

// « Un camion, c'est une voiture en plus gros. » Vraiment ?
class Truck extends Car
{
    public function load(float tons): string
    {
        return "Chargement de {tons} t.";
    }
}

truck = new Truck(seats: 5);
echo truck->openTrunk(),PHPEOL;echotruck→openTrunk(), PHP_EOL;
echo truck->openTrunk(),PHPEOL;echotruck→seats, ' places', PHP_EOL;
```

```
Le coffre s'ouvre.
5 places
```

ATTENTION

PHP ne proteste pas. Le résultat est pourtant absurde : un camion n'a ni coffre ni cinq places. Ces membres n'existent sur `Truck` que parce que `Truck` a hérité pour gagner du temps, et n'importe qui pourra les appeler.

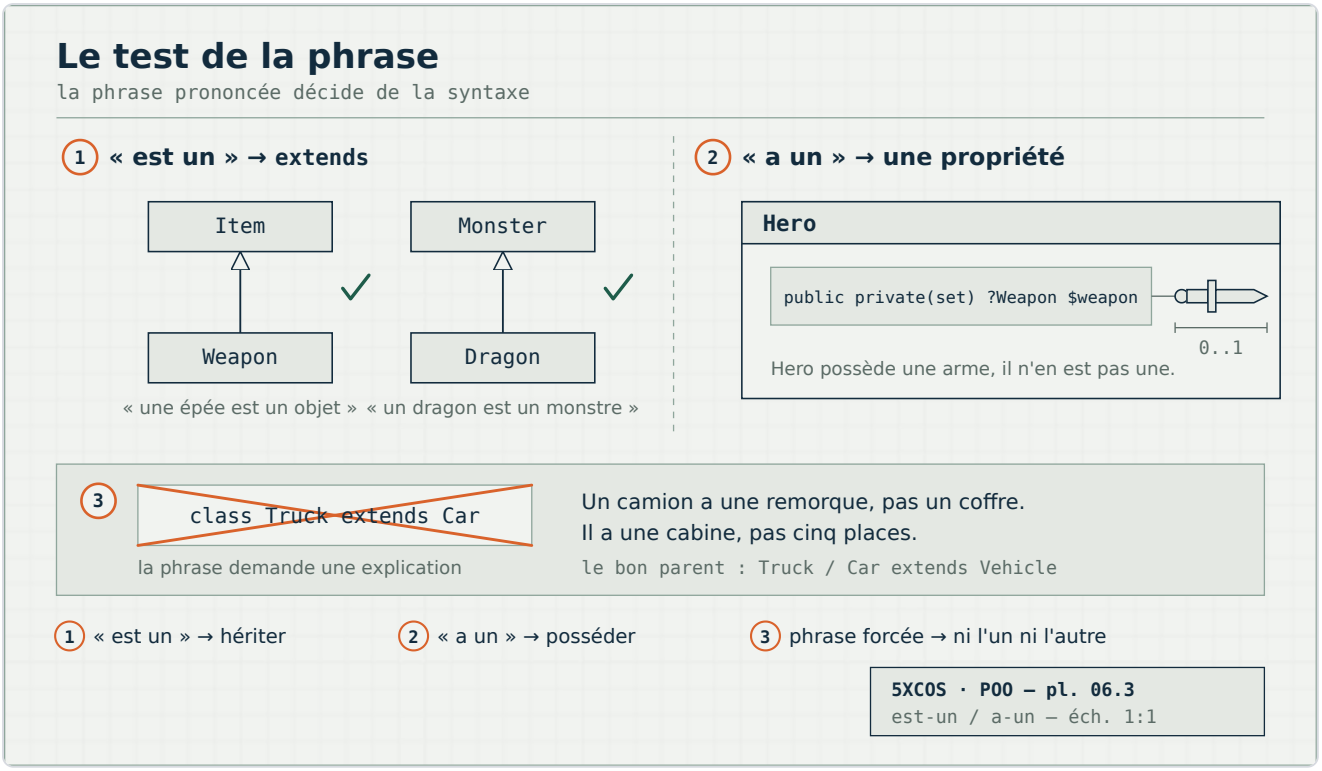
La phrase juste est « un camion **est un** véhicule », et « une voiture **est un** véhicule ». Le parent commun est `Vehicle`, pas `Car`.

Ne choisissez donc pas l'héritage seulement pour réutiliser du code. La classe fille doit aussi représenter un cas particulier de la classe parente. Le partage de code entre classes sans parent commun se fera autrement, au chapitre 7.

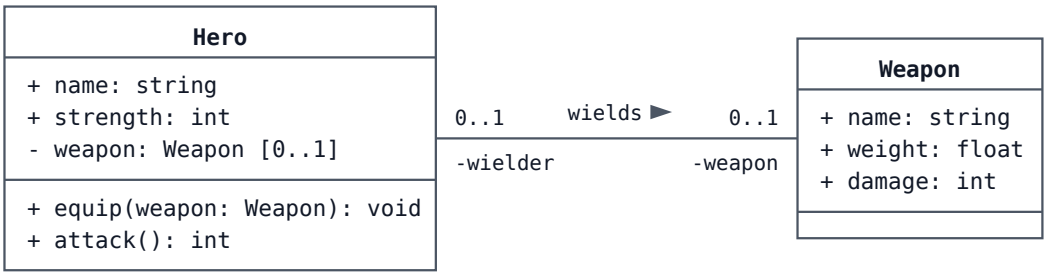
Quand la phrase « est un » sonne faux, la réponse est presque toujours « **a un** ». Le héros équipe une arme, il n'en est pas une. En code, c'est une propriété.

```php
final class Hero
{
    /** hp, maxHp, isFullHealth, inventory, takeDamage(), heal(), isAlive() et __toString()
:
     *  inchangés depuis les chapitres 3 et 5, omis ici pour ne montrer que l'arme. */
    public private(set) ?Weapon weapon = null;

    public function __construct(
        public readonly string name,
        int maxHp=10,publicreadonlyintmaxHp = 10,
        public readonly int maxHp=10,publicreadonlyintstrength = 2,
    ) {
        this->maxHp=this->maxHp = this->maxHp=maxHp;
        this->hp=this->hp = this->hp=maxHp;
        this->inventory = new Inventory();
    }

    public function equip(Weapon weapon): void
    {
        this->weapon=this->weapon = this->weapon=weapon;
    }

    public function attack(): int
    {
        return this->strength+(this->strength + (this->strength+(this->weapon?->damage ??
0);
    }
}

arthur=newHero('Arthur');printf('A`mainsnues:arthur = new Hero('Arthur');
printf('À mains nues : %d dégâts%s', arthur=newHero('Arthur');printf('A`mainsnues:arthur
->attack(), PHP_EOL);

arthur->equip(newWeapon('E´pe´ecourte',2.0,5));printf('E´pe´eenmain:arthur->equip(new
Weapon('Épée courte', 2.0, 5));
printf('Épée en main : %d dégâts%s',
arthur->equip(newWeapon('E´pe´ecourte',2.0,5));printf('E´pe´eenmain:arthur->attack(),
PHP_EOL);
```

```
À mains nues : 2 dégâts
Épée en main : 7 dégâts
```



`Hero` devient `final` : personne n'en hérite. Le trait entre `Hero` et `Weapon` est l'association ——— du chapitre 4, avec ses deux rôles :



L'arme arrive de l'extérieur, le héros peut en changer, et elle lui survit. Une classe fille, elle, ne change jamais de parent.

La phrase juste	Ce que vous écrivez
« un X **est un** Y »	`class X extends Y`
« un X **a un** Y »	une propriété `public private(set) ?Y $y` dans `X`

Une classe, un rôle. Une classe qui fait deux choses se coupe en deux avant de chercher son parent.

6. Les exceptions : une hiérarchie que vous prolongez

Au chapitre 5, `Inventory::add()` renvoyait `false` quand le sac était plein. Ça fonctionne, mais c'est facile à rater. `$inventory->add(new Weapon('Épée courte', 2.0, 5));` sans lire la valeur de retour s'exécute très bien et perd l'épée sans rien dire.

Lancer, attraper

Au chapitre 3, le constructeur de `Dice` lançait une `InvalidArgumentException` avec `throw`, et le programme s'arrêtait sur `Fatal error: Uncaught`. Voici ce qui se passe quand quelqu'un l'attrape :

```php
try {
    dice = new Dice(0);
    echo 'Dé créé.', PHP_EOL;
} catch (InvalidArgumentException e) {
    echo 'Impossible : ', $e->getMessage(), PHP_EOL;
}

echo 'Le programme continue.', PHP_EOL;
```

```
Impossible : Un dé a au moins 2 faces, 0 reçu.
Le programme continue.
```

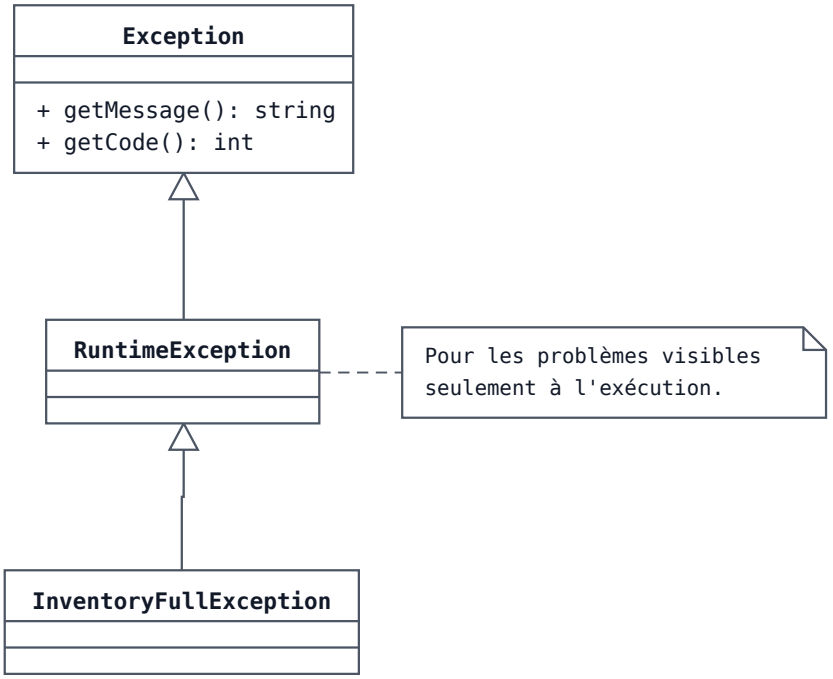
Le bloc `try` est le code surveillé. Dès que `throw` s'exécute, PHP quitte le `try` à cette ligne : `new` ne renvoie rien, `$dice` n'existe pas, le `echo 'Dé créé.'` est sauté. PHP regarde ensuite les `catch` qui suivent. Le premier dont le type correspond à l'exception lancée, la classe elle-même ou une de ses parentes, reçoit l'objet dans `$e` et s'exécute. Puis le programme reprend après le bloc, comme si rien ne s'était passé.

Si aucun `catch` ne correspond, l'exception remonte à la fonction qui a appelé, puis à celle du dessus, jusqu'à trouver un `try` qui l'attrape. Si elle arrive tout en haut sans être attrapée, PHP affiche `Fatal error: Uncaught …` avec la liste des appels traversés, et le programme s'arrête. C'est ce qui arrivait au chapitre 3.

Une exception est donc un objet, avec une classe, un message et une trace d'appels. Les exceptions de PHP forment une hiérarchie de classes, et vous y ajoutez la vôtre avec `extends` :

```php
/** Corps vide : tout vient de RuntimeException. Seul le nom compte. */
final class InventoryFullException extends RuntimeException
{
}
```

Les classes fournies par PHP, comme `RuntimeException` ou `InvalidArgumentException`, vivent dans le **namespace global**, celui qui n'a pas de nom. Une barre oblique seule en tête d'un nom désigne ce namespace. Dans le dépôt du kata, vos fichiers commencent par `namespace Dungeon;`, donc un nom sans barre est cherché dans `Dungeon` : `RuntimeException` devient `Dungeon\RuntimeException`, qui n'existe pas. Écrivez `extends \RuntimeException`. Le chapitre 3 écrivait `\InvalidArgumentException` pour la même raison. Où se place votre classe dans la famille :



`Exception` est la branche des problèmes applicatifs, celle dont vous héritez. `RuntimeException` regroupe ce qui ne se détecte qu'à l'exécution. Un sac trop plein en fait partie : tout dépend de ce que le joueur a ramassé. À côté, `Error` regroupe les erreurs du moteur, comme une classe abstraite instanciée. Dans le code d'une application, vous ne l'attrapez pas, vous corrigez le bug. Un test l'attrape pour vérifier qu'elle est bien levée, comme au niveau 1 du kata.

> **NOTE**
>
> **Histoire de PHP.** Les exceptions arrivent avec **PHP 5, en 2004**. Avant, une fonction signalait un problème en renvoyant `false` ou `-1`, et l'appelant devait penser à tester la valeur. Les cas graves se terminaient par un `die('Erreur')` qui coupait la page au milieu. **PHP 7, en 2015**, a ajouté la branche `Error` pour les erreurs du moteur, jusque-là impossibles à attraper.

Trois lignes, un corps vide, et vous avez votre propre type d'erreur, avec `getMessage()`, `getCode()` et la trace d'appels. C'est de l'héritage, rien d'autre. Ce qui compte, c'est le nom de la classe, parce que c'est lui qui rendra le `catch` précis. Le message, lui, reste celui que vous passez au constructeur.

```php
public function add(Item $item): void
{
    if ($this->totalWeight() + $item->weight > $this->maxWeight) {
        throw new InventoryFullException(sprintf(
            '"%s" ne rentre pas : le sac ne porte que %s kg.',
            $item->name,
            $this->maxWeight,
        ));
    }

    $this->items[] = $item;
}
```

`add()` ne renvoie plus rien. Aux chapitres 4 et 5, `add()` était déclarée `add(Item $item): bool` : elle renvoyait `false` quand le sac débordait. Elle est maintenant déclarée `add(Item $item): void` et elle lève une `InventoryFullException`. Deux raisons. La première : avec l'exception, le retour ne vaudrait plus jamais `false`, et une signature qui annonce un résultat impossible ment à celui qui la lit. La seconde : un `false` que personne ne teste disparaît en silence, alors qu'une exception non attrapée arrête le programme. À partir d'ici, `add()` renvoie `void` et lève.

```php
inventory = new Inventory(maxWeight: 3.0);

try {
    $inventory->add(new Weapon('Épée courte', 2.0, 5));
    $inventory->add(new Weapon('Marteau de guerre', 12.0, 9));
    echo 'Les deux armes sont dans le sac.', PHP_EOL;
} catch (InventoryFullException e) {
    echo 'Sac plein : ', e->getMessage(), PHP_EOL;
}

echo 'Poids transporté : ', inventory->totalWeight(), ' kg', PHP_EOL;
```

```
Sac plein : "Marteau de guerre" ne rentre pas : le sac ne porte que 3 kg.
Poids transporté : 2 kg
```

Le `try` s'arrête à la ligne fautive, donc le `echo` de succès n'est jamais exécuté. L'épée déjà ajoutée, elle, reste dans le sac : une exception n'annule pas ce qui a été fait avant. Et le `catch` est **ciblé**. Il n'attrape que les sacs pleins. Une faute de frappe sur un nom de méthode dans le même bloc continue de provoquer une erreur.

> **ATTENTION**
>
> Attrapez ce que vous attendez, là où vous savez quoi faire : afficher un message au joueur, lui proposer de jeter un objet, réessayer. Le programme continue. Laissez passer ce que vous n'attendez pas : une exception qui n'aurait jamais dû arriver signale un bug, et le programme qui s'arrête avec sa trace vous dit où chercher. `catch (Exception $e) { }` autour de tout le programme cache les deux à la fois. Le joueur ne sait plus pourquoi son épée a disparu, et vous ne savez plus que le code est cassé.

À retenir

- `extends` dit ce qu'une chose **est**. Le test : prononcer « un X est un Y », puis vérifier que le contenu du parent convient à la fille.
- `abstract` interdit l'instanciation et impose aux filles les méthodes sans corps. `protected` ouvre aux filles ce que `private` ferme, le Donjon n'en a pas encore besoin. Une classe fille qui a son propre constructeur doit appeler `parent::__construct()`.
- Le polymorphisme, c'est une boucle sur une liste de `Monster` qui appelle `attack()` sans savoir qui répond. Ajouter un monstre ne modifie aucune boucle existante.
- La méthode appelée dépend de **l'objet**, pas du type de la variable. PHP part de la classe réelle et remonte.
- Un `try` surveille un bloc. Le premier `catch` dont le type correspond reprend la main, et le programme continue après. Sans `catch`, l'exception remonte les appels jusqu'au `Fatal error: Uncaught`.
- `InventoryFullException extends \RuntimeException` : trois lignes suffisent, le nom de la classe rend le `catch` précis. Attrapez ce que vous attendez, laissez passer le reste.

Exercices

Sur papier

Deux questions à traiter sans ordinateur, avant d'ouvrir le kata.

1. Écrivez la phrase « est un » ou « a un » pour ces quatre paires, et dites ce que vous écririez en PHP : `Dungeon` / `Room`, `Room` / `Monster`, `Potion` / `Item`, `Hero` / `Inventory`.
2. `Inventory::remove()` ne fait rien quand l'objet n'est pas dans le sac. Faut-il lever une exception, ou ne rien faire est-il correct ici ? Justifiez en deux phrases, puis écrivez `ItemNotFoundException` si vous avez répondu oui. Les deux réponses se défendent, c'est le raisonnement qui compte. Restez sur papier : les tests du niveau 2 attendent que `remove()` ne fasse rien.

Kata niveau 3

Dans https://github.com/opmvpc/poo-katas-26, faites passer le groupe `niveau-3` :

```bash
composer test -- --group=niveau-3
```

Ce que les tests attendent :

Classe	Signature	Comportement vérifié
`Item`	`abstract`, `describe(): string` abstraite	le test demande à PHP si la classe est abstraite, et attend `true`
`Weapon`	`__construct(string` $name$, $float$`weight, int` $damage$, $Rarity$`rarity = Rarity::Common)`	`$sword->damage` est `public readonly`. `describe()` renvoie `"Épée courte : arme (2 kg, 5 dégâts)"`
`Potion`	`__construct(string` $name$, $float$`weight, int` $healing$, $Rarity$`rarity = Rarity::Common)`	`$potion->healing` est `public readonly`. `describe()` renvoie `"Potion de soin : potion (0.5 kg, +5 PV)"`
`Monster`	`abstract`, `__construct(string` $name$, int`maxHp)`	`name` **public readonly**, `hp` et `maxHp` en **`public private(set)`**. `hp` reste entre `0` et `maxHp` : après `takeDamage(3)` puis `takeDamage(99)`, `hp` vaut `2` puis `0`. `__toString()` renvoie `"Gobelin (5/5 PV)"`
`Goblin`	`__construct()`	`parent::__construct('Gobelin', 5)`. `attack()` renvoie **2**
`Dragon`	`__construct()`	`parent::__construct('Dragon', 30)`. `attack()` renvoie **8**
`InventoryFullException`	`extends \RuntimeException`	levée par `add()` quand ça déborde, et le sac reste inchangé
`Hero::equip()` et `Hero::$weapon`	`equip(Weapon): void`, `public private(set) ?Weapon $weapon`	`$hero->weapon` vaut `null` au départ. `attack()` passe de `2` à `7` avec une épée à 5 dégâts
`Hero::drink()`	`drink(Potion): void`	soigne du montant de la potion, puis la retire de l'inventaire

Le test de `drink()` en entier :

```php
hero=newHero('Arthur');hero = new Hero('Arthur');
hero=newHero('Arthur');potion = new Potion('Potion de soin', 0.5, 5);
hero->inventory->add(hero->inventory->add(hero->inventory->add(potion);

hero->takeDamage(6);hero->takeDamage(6);
hero->takeDamage(6);hero->drink(potion);expect(potion);

expect(potion);expect(hero->hp)->toBe(9);
expect($hero->inventory->has('Potion de soin'))->toBeFalse();
```

Cinq points de vigilance :

1. Les deux `describe()` sont comparés au caractère près. Notez le `%s` pour le poids, qui affiche `2` et non `2.00`, et le `+` devant les points de vie de la potion.
2. Le gobelin a 5 points de vie, le dragon 30. Le test vérifie aussi `(string) $goblin`.
3. `Item` a un troisième paramètre `Rarity $rarity = Rarity::Common` dans les fichiers fournis. Il ne sert qu'au niveau 4. Passez-le au `parent::__construct()` depuis `Weapon` et `Potion`, où il arrive en quatrième position.
4. Les fichiers fournis déclarent encore `add(Item $item): bool`, la signature du niveau 2. Passez-la à `void` en même temps que vous écrivez le `throw`. Les tests ne lisent pas la valeur de retour : ils vérifient que l'exception part et que le sac reste inchangé.
5. `Monster` déclare aussi `heal()` et la propriété `isFullHealth`, comme `Hero`. Écrivez-les de la même façon qu'au chapitre 3.

Lisez les tests avant de coder : ils sont l'énoncé complet.

Pour aller plus loin

Dans le dépôt du kata, ajoutez `src/Skeleton.php` : un squelette a 4 points de vie et `attack()` renvoie 3. Relancez `composer test -- --group=niveau-3`. Tout doit rester vert sans qu'aucun autre fichier ne bouge. Si vous avez dû ouvrir une boucle ou une autre classe, cherchez pourquoi.

2.7 Contrats : interfaces, traits et enums

Séance 3 · ~3 périodes

Slides du chapitre (PDF) PDF

Le chapitre 6 a donné l'héritage : dire ce qu'une chose **est**. Il a des limites. Une classe n'a qu'un seul parent. Or il faut parfois dire « ceci sait se battre » sans imposer une famille. Il faut parfois partager du code entre deux classes sans parent commun. Il faut parfois n'autoriser que trois valeurs, pas une de plus. Trois besoins, trois outils : **interface**, **trait**, **enum**. Vous les rencontrerez dans presque tous les fichiers de Laravel.

À la fin de ce chapitre, vous serez capables de :

1. **définir** une interface, l'implémenter, et typer un paramètre par un contrat plutôt que par une classe.
2. **utiliser** les interfaces natives de PHP pour rendre vos objets compatibles avec `count()` et `foreach`.
3. **choisir** entre une interface et une classe abstraite.
4. **extraire** du code dupliqué dans un trait, et le dessiner en UML sans le confondre avec un héritage.
5. **écrire** une enum avec des méthodes, et l'utiliser à la place d'une chaîne de caractères.

Le code de ce chapitre vit dans le dépôt des katas, dans le namespace `Dungeon` du chapitre 2. Les sorties de PHP affichent donc des noms complets, comme `Dungeon\Rarity`.

Le code qui sent mauvais

Cinq minutes, en binôme. Ce code sent mauvais : un détail d'écriture y trahit un défaut de conception. Le second affichage vaut `1`, pas `3`. Pourquoi ?

```php
final class Treasure
{
    public function __construct(private string $rarity) {}

    public function multiplier(): float
    {
        if ($this->rarity === 'rare') {
            return 1.5;
        }
        if ($this->rarity === 'Legendary') {
            return 3.0;
        }

        return 1.0;
    }
}

echo (new Treasure('rare'))->multiplier(), PHP_EOL;
echo (new Treasure('legendary'))->multiplier(), PHP_EOL;
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/contrats-interfaces-traits-enums

1. L'interface : un contrat sans code

Il faut une fonction qui permette à un héros ou à un monstre d'en attaquer un autre. Problème : `Hero` et `Monster` n'ont aucun parent commun, et il n'y a aucune raison d'en inventer un. Un héros n'est pas un monstre.

```php
interface Fighter
{
    public function attack(): int;

    public function takeDamage(int $amount): void;

    public function isAlive(): bool;
}
```

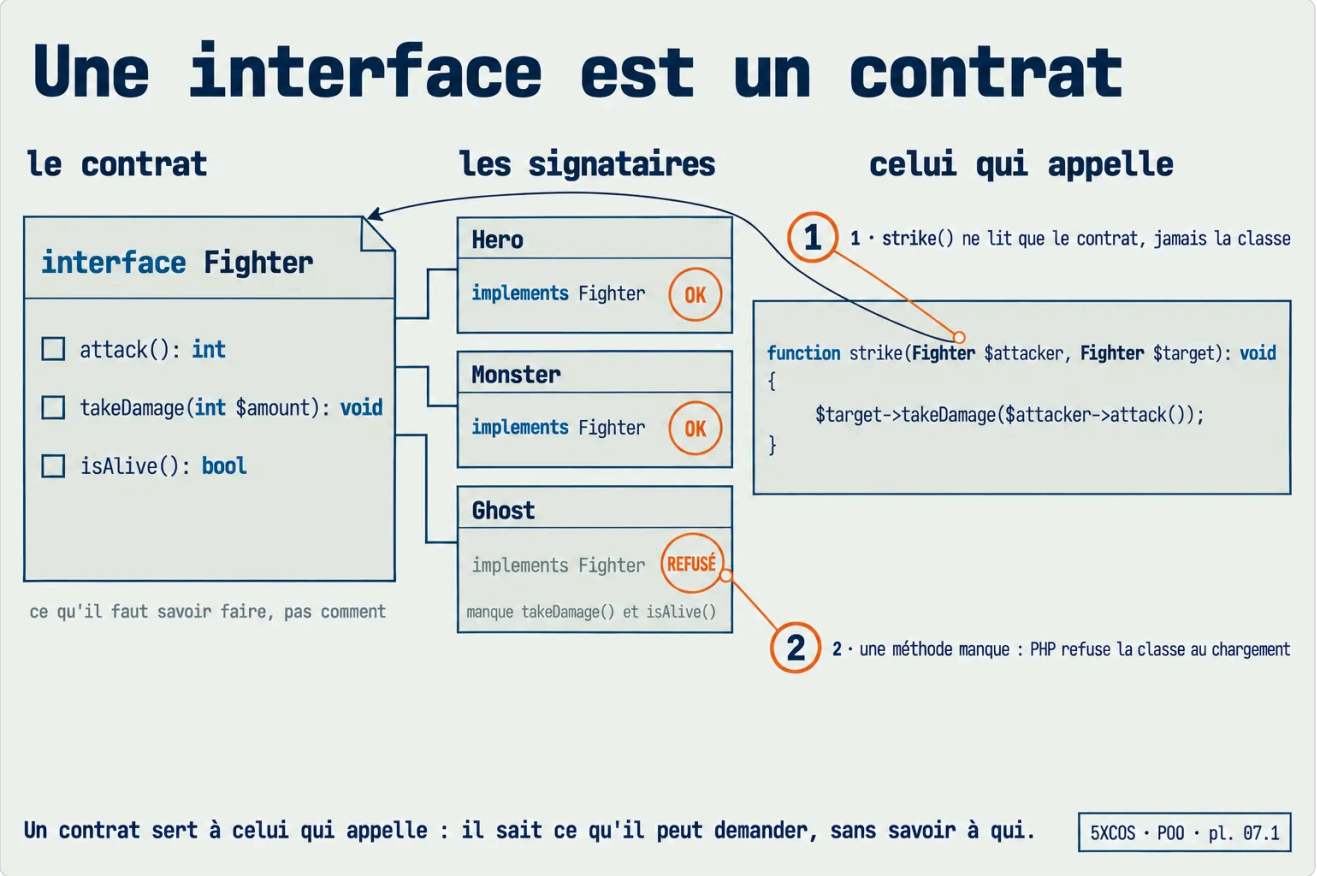
Une **interface** est un contrat : la liste de ce qu'une classe doit savoir faire, sans dire comment. Elle ne contient que des signatures de méthodes publiques : le nom de chaque méthode, ses paramètres, son type de retour, et pas une ligne de code. Une classe déclare qu'elle remplit ce contrat avec `implements`, et PHP vérifie au chargement qu'elle a bien écrit chaque méthode.

```php
final class Hero implements Fighter { /* les méthodes du chapitre 6 */ }

abstract class Monster implements Fighter { /* les méthodes du chapitre 6 */ }
```

Une classe concrète doit fournir les trois méthodes, directement ou par héritage. Une classe qui en oublie une est refusée au chargement :

```php
final class Ghost implements Fighter
{
    public function attack(): int
    {
        return 1;
    }
}
```

```
Fatal error: Class Dungeon\Ghost contains 2 abstract methods and must therefore be declared
abstract or implement the remaining methods (Dungeon\Fighter::takeDamage,
Dungeon\Fighter::isAlive)
```



Un contrat sert d'abord à celui qui appelle. `strike()`, plus bas, sait ce qu'elle peut demander à ses deux paramètres sans savoir de quelle classe ils sont. Ce sera utile chaque fois que des classes sans parent commun doivent jouer le même rôle, chaque fois qu'un objet doit se brancher sur une syntaxe de PHP,

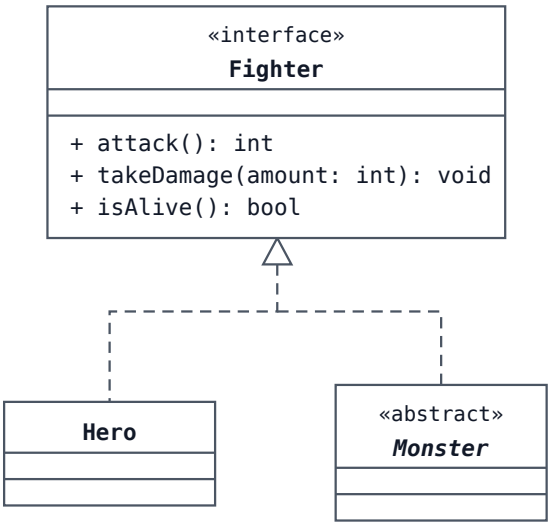
comme `count()` à la section 3, et chaque fois qu'une pièce doit pouvoir être remplacée par une autre, comme le dé truqué des tests du niveau 5.

Une classe implémente autant d'interfaces qu'elle veut, séparées par des virgules. Elle n'hérite que d'un seul parent. Et `Monster` peut rester abstraite tout en implémentant `Fighter` : elle laisse `attack()` à ses classes filles, qui remplissent le contrat pour elle.

Un paramètre se type par le contrat, pas par la classe :

```php
/** Ne demande ni un Hero ni un Goblin : deux objets qui savent se battre, c'est tout. */
function strike(Fighter attacker,Fighterattacker, Fighter attacker,Fightertarget): void
{
    target->takeDamage(target→takeDamage(target->takeDamage(attacker→attack());
}

arthur=newHero(’Arthur’);arthur = new Hero('Arthur');
arthur=newHero(’Arthur’);goblin = new Goblin();

strike(arthur,arthur, arthur,goblin);
strike(goblin,goblin, goblin,arthur);

echo arthur,PHPEOL;echoarthur, PHP_EOL;
echo arthur,PHPEOL;echogoblin, PHP_EOL;
var_dump(arthurinstanceofFighter,arthur instanceof Fighter, arthurinstanceofFighter,goblin
instanceof Fighter);
```

```
Arthur (8/10 PV)
Gobelin (3/5 PV)
bool(true)
bool(true)
```



Le trait pointillé au triangle vide ◁- - - - se lit « réalise » : `Hero` et `Monster` remplissent le contrat `Fighter`, sans lien de parenté entre elles.

`strike()` ne connaît ni `Hero`, ni `Goblin`, ni `Dragon`. Tout nouvel objet dont la classe implémente `Fighter` pourra lui être passé sans modifier la fonction. C'est encore du polymorphisme, comme au chapitre 6, mais sans obliger les deux classes à descendre du même parent.

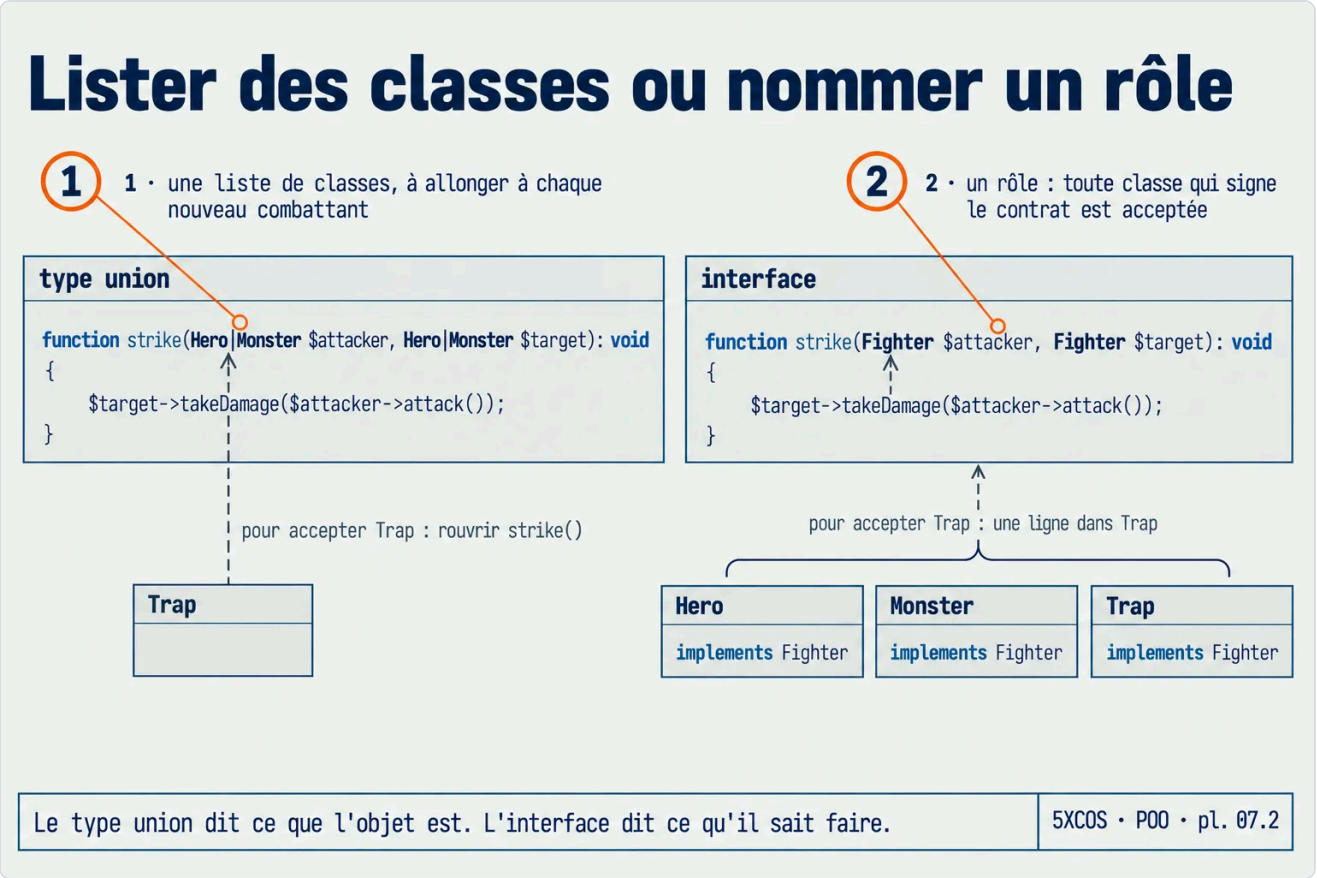
PHP offrait une autre écriture, sans interface. Un **type union** accepte l'un ou l'autre de plusieurs types, séparés par une barre verticale :

```php
function strike(Hero|Monster attacker,Hero|Monsterattacker, Hero|Monster
attacker,Hero|Monstertarget): void
{
    target->takeDamage(target→takeDamage(target->takeDamage(attacker→attack());
}
```

Vous croiserez souvent des types union dans du code réel : `int|float` pour un nombre, `string|null` pour une valeur qui peut manquer, `Countable|array` dans le message d'erreur de la section 3. Ici, l'écriture marche, mais elle liste des classes. Le jour où un `Trap` doit blesser le héros, il faut rouvrir `strike()` et allonger la liste. Et rien ne garantit que `Hero` et `Monster` aient bien `attack()` et `takeDamage()` : PHP ne le vérifie qu'au moment de l'appel.

L'interface renverse la question. `strike()` ne demande plus « es-tu un `Hero` ou un `Monster` ? », mais « sais-tu te battre ? ». La liste des classes n'existe plus, et la vérification des trois méthodes se fait au chargement de chaque classe qui écrit `implements Fighter`.

	`Hero|Monster`	`Fighter`
Ce que le paramètre dit	l'une de ces deux classes	tout objet qui sait se battre
Ajouter un `Trap` qui blesse	rouvrir `strike()`	écrire `implements Fighter` dans `Trap`
Qui vérifie que `attack()` existe	personne avant l'appel	PHP, au chargement de la classe
Quand l'utiliser	deux types de nature différente, sans contrat commun : `int|float`	plusieurs classes qui jouent le même rôle



NOTE

Histoire de PHP. Les interfaces arrivent avec **PHP 5, en 2004**, en même temps que les classes abstraites. Depuis **PHP 8.4**, une interface peut aussi imposer une propriété accessible en lecture. Ce cours n'utilise pas cette possibilité : une interface y contient seulement des signatures de méthodes.

2. Interface ou classe abstraite ?

Les deux imposent des méthodes. La question qui tranche : les classes concernées appartiennent-elles à la même famille ?

	Classe abstraite	Interface
Peut contenir du code	oui : méthodes complètes, constructeur	non : que des signatures
Peut avoir des propriétés	oui	non, dans ce cours
Combien par classe	**une seule** (`extends`)	autant qu'on veut (`implements`)
Ce qu'elle dit	« ceci **est un** … »	« ceci **sait faire** … »

	Classe abstraite	Interface
Dans le Donjon	`Monster` rassemble le code commun aux monstres	`Fighter` permet une même fonction de combat pour le héros et les monstres

ASTUCE

Une règle de nommage utile pour lire le code des autres. Une interface porte souvent un rôle, comme `Fighter`, ou un nom en `-able`, comme `Countable` et `Stringable`. Une classe abstraite porte le nom d'une famille : `Item`, `Monster`, `Model`.

Les deux vont souvent ensemble. `Monster` est abstraite **et** implémente `Fighter`.

3. Les interfaces natives de PHP

L'`Inventory` du chapitre 6 a une méthode `count()` depuis le chapitre 5. La fonction `count()` de PHP ne la connaît pas :

```php
inventory=newInventory();echocount(inventory = new Inventory();
echo count(inventory=newInventory();echocount(inventory);
```

```
Fatal error: Uncaught TypeError: count(): Argument #1 ($value) must be of type
Countable|array, Dungeon\Inventory given
```

Et `foreach (`$inventory as item)` ne parcourt rien : PHP parcourt les propriétés publiques d'un objet, il n'y en a aucune, la boucle ne fait pas un tour et ne dit rien. PHP fournit une série d'interfaces toutes faites pour brancher vos objets sur sa syntaxe.

Interface	Méthode à écrire	Ce qu'elle permet
`Stringable`	`__toString(): string`	`echo $item;` et l'interpolation `"{$item}"`
`Countable`	`count(): int`	`count($inventory)`
`IteratorAggregate`	`getIterator(): Traversable`	`foreach (`$inventory as item)`
`JsonSerializable`	`jsonSerialize(): mixed`	choisir ce que `json_encode()` produit
`ArrayAccess`	`offsetGet()`, `offsetSet()`…	`$inventory['épée']`

Le Donjon utilise les trois premières. `JsonSerializable` et `ArrayAccess` sont dans le tableau pour que vous les reconnaissiez quand vous les croiserez dans Laravel ; l'exercice 2 de « Pour aller plus loin » fait implémenter `JsonSerializable` à l'inventaire.

```php
final class Inventory implements \Countable, \IteratorAggregate
{
    /** @var Item[] */
    private array items=[];/**ContratdeCountable:cequecount(items = [];

    /** Contrat de Countable : ce que count(items=[];/
**ContratdeCountable:cequecount(inventory) doit renvoyer. */
    public function count(): int
    {
        return count(this→items);
    }

    /** Contrat de IteratorAggregate : ce que foreach doit parcourir. */
    public function getIterator(): \Traversable
    {
        return new \ArrayIterator(this→items);
    }

    // … le constructeur, add() et totalWeight() des chapitres 5 et 6, has() et remove() du
kata
}
```

La méthode `count()` existait déjà. Ce qui est neuf, c'est `implements \Countable`, qui la branche sur la fonction `count()` du langage. Les antislashs disent que `Countable`, `IteratorAggregate`, `Traversable` et `ArrayIterator` viennent du namespace global de PHP, pas du dossier `Dungeon`, comme `\RuntimeException` au chapitre 6. Sans eux, PHP chercherait `Dungeon\Countable` et ne le trouverait pas.

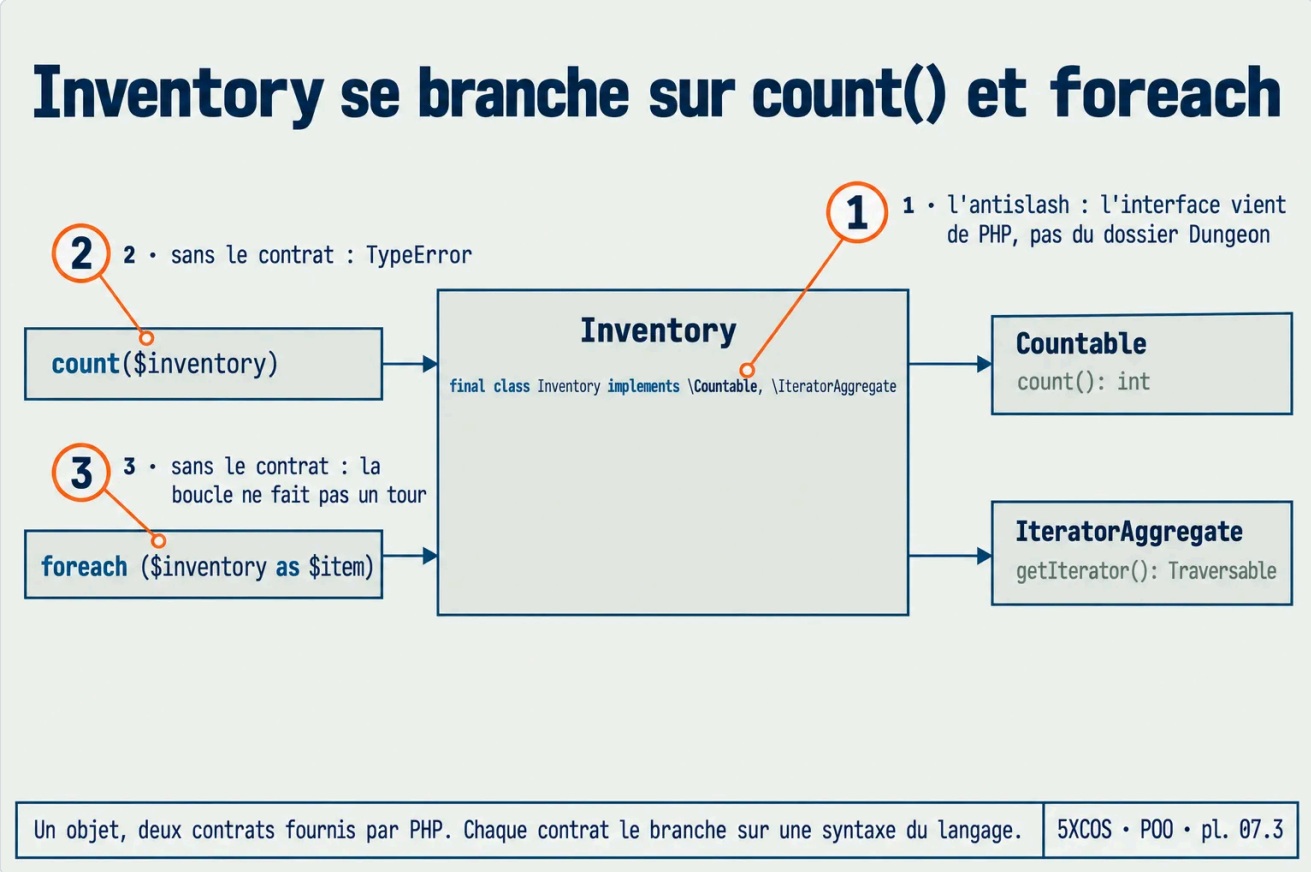
`ArrayIterator` est une classe de PHP : donnez-lui un tableau, elle sait le parcourir. Avec `IteratorAggregate`, vous lui déléguez le parcours. Avec `Iterator`, vous écririez vous-même les cinq méthodes du parcours, `current()`, `key()`, `next()`, `rewind()` et `valid()`. Le type de retour annoncé est `Traversable`, l'interface commune à tous les parcours.

`Stringable` suit la même idée. Une classe qui déclare `__toString()` implémente `Stringable` sans qu'on l'écrive, depuis PHP 8.0. L'écrire à la main documente le choix et permet de typer un paramètre par cette interface :

```php
abstract class Item implements \Stringable
{
    // … name, weight et describe() du chapitre 6

    /** Contrat de Stringable : afficher un objet, c'est le décrire. */
    public function __toString(): string
    {
        return $this→describe();
    }
}
```

```php
inventory=newInventory();inventory = new Inventory();
inventory=newInventory();inventory→add(new Weapon('Épée courte', 2.0, 5));
inventory->add(newPotion('Potiondesoin',0.5,5));echocount(inventory→add(new Potion('Potion
de soin', 0.5, 5));

echo count(inventory->add(newPotion('Potiondesoin',0.5,5));echocount(inventory), ' objets',
PHP_EOL;

foreach (inventoryasinventory as inventoryasitem) {
    echo '- ', $item, PHP_EOL;
}
```

```
2 objets
- Épée courte : arme (2 kg, 5 dégâts)
- Potion de soin : potion (0.5 kg, +5 PV)
```



`$inventory` n'est pas un tableau. Il se comporte comme un tableau parce qu'il implémente deux interfaces. Le `-` de la deuxième ligne affiche `$item` directement, grâce à `Item::__toString()`.

4. Le trait `HasHealth` : sortir le code dupliqué

Rouvrez le `src/Hero.php` et le `src/Monster.php` que vous avez écrits aux niveaux 1 à 3. Les deux classes portent exactement les mêmes propriétés et les mêmes méthodes de santé : `hp` avec son hook, `maxHp`, `isFullHealth`, `takeDamage()`, `heal()` et `isAlive()`. Elles ne peuvent pas hériter l'une de

l'autre : un héros n'est pas un monstre.

```php
final class Hero implements Fighter
{
    public private(set) int maxHp;publicprivate(set)intmaxHp;

    public private(set) int maxHp;publicprivate(set)inthp {
        set => max(0, min(this->maxHp,this→maxHp, this->maxHp,value));
    }

    public function takeDamage(int $amount): void
    {
        // … la vérification du chapitre 3
        $this→hp -= $amount;
    }

    // … et isFullHealth, heal(), isAlive()
}
```

`Monster` contient les mêmes déclarations, mot pour mot. Chaque règle de santé demande de modifier deux classes, et un jour l'une des deux sera oubliée.

Un **trait** regroupe des propriétés et des méthodes réutilisables dans plusieurs classes. Il ne crée aucun lien d'héritage : c'est du code que PHP recopie dans la classe qui l'utilise, au chargement.

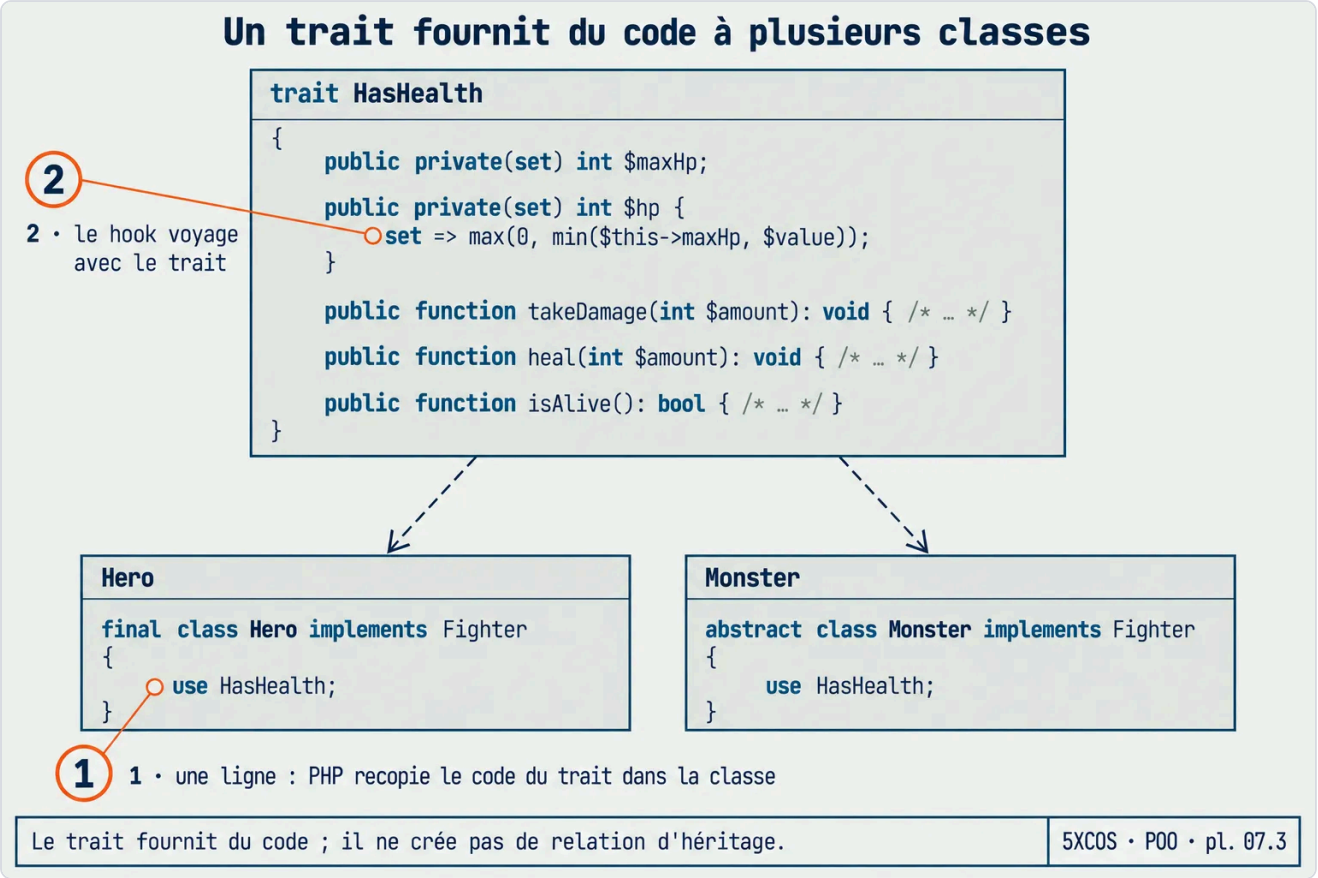
> **NOTE**
>
> **Histoire de PHP.** Les traits arrivent avec **PHP 5.4, en 2012**. Avant, deux classes sans parent commun qui voulaient le même code n'avaient que deux solutions : le recopier, ou passer par un troisième objet. Laravel en est rempli, et vous en verrez deux dans la section 7.

Déplacez d'abord les propriétés, le hook et les méthodes de santé dans `src/HasHealth.php`, qui ne contient que la déclaration du trait dans le squelette.

```php
trait HasHealth
{
    /** Le maximum de points de vie. */
    public private(set) int maxHp;/**Lespointsdeviecourants:lehooklesgardeentre0etmaxHp;

    /** Les points de vie courants : le hook les garde entre 0 et maxHp;/
**Lespointsdeviecourants:lehooklesgardeentre0etmaxHp. */
    public private(set) int hp {
        set => max(0, min($this->maxHp, $value));
    }

    public bool isFullHealth {
        get => this->hp===this->hp === this->hp===this->maxHp;
    }

    public function takeDamage(int amount): void
    {
        if ($amount < 0) {
            throw new \InvalidArgumentException("Un dégât ne peut pas être négatif, $amount
reçu.");
        }

        $this->hp -= $amount;
    }

    public function heal(int amount): void
    {
        if (amount < 0) {
            throw new \InvalidArgumentException("Un soin ne peut pas être négatif, $amount
reçu.");
        }

        this->hp += amount;
    }

    public function isAlive(): bool
    {
        return this->hp > 0;
    }
}
```

Écrivez ensuite une seule ligne dans chacune des deux classes : `use HasHealth;`.

```php
final class Hero implements Fighter
{
    use HasHealth;

    public readonly Inventory inventory;publicprivate(set)?Weaponinventory;
    public private(set) ?Weapon inventory;publicprivate(set)?Weaponweapon = null;

    public function __construct(
        public readonly string name,intname,
        int name,intmaxHp = 10,
        public readonly int strength = 2,
    ) {
        // … les vérifications du nom et de maxHp, comme dans votre kata
        $this->maxHp = $maxHp;
        $this->hp = $maxHp;
        $this->inventory = new Inventory();
    }

    public function attack(): int
    {
        return $this->strength + ($this->weapon?->damage ?? 0);
    }

    // … equip(), drink(), __toString()
}

abstract class Monster implements Fighter
{
    use HasHealth;

    public function __construct(
        public readonly string name,
        int $maxHp,
    ) {
        $this->maxHp = $maxHp;
        $this->hp = $maxHp;
    }

    abstract public function attack(): int;

    // … __toString() du chapitre 6
}
```



Les propriétés et les méthodes de santé ont disparu des deux fichiers. Le constructeur, lui, n'a pas changé : il affecte toujours `$this->maxHp` et `$this->hp`, qui sont maintenant déclarées dans le trait. Le contrat `Fighter` reste rempli, puisque `takeDamage()` et `isAlive()` viennent de `HasHealth`.

> **NOTE**
>
> Le mot-clé `use` a deux sens selon l'endroit où il se trouve. **En haut du fichier**, après le `namespace`, il importe une classe : `use Dungeon\Dice;` du chapitre 2. **À l'intérieur d'une classe**, il colle un trait : `use HasHealth;`. Même mot, deux mécanismes. Quand vous lirez un fichier de Laravel, regardez d'abord l'indentation.

PHP

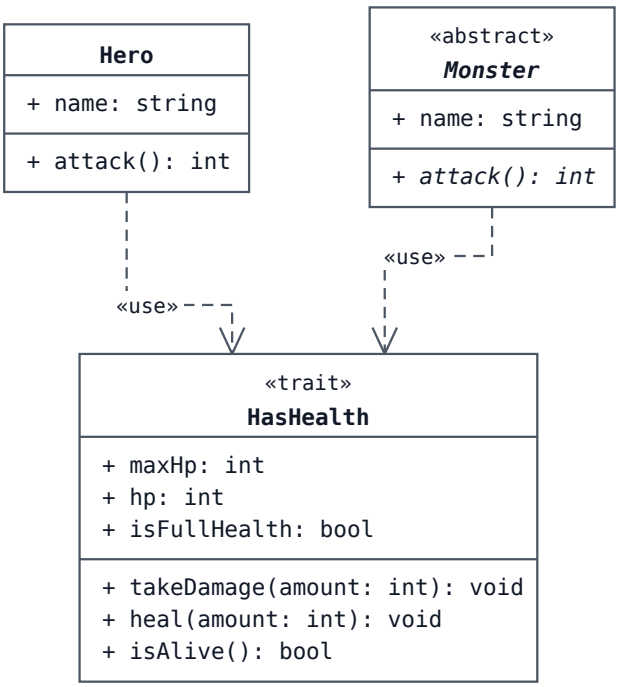
```php
var_dump(class_uses(Hero::class));
var_dump(class_uses(Goblin::class));
```

```
array(1) {
  ["Dungeon\HasHealth"]=>
  string(17) "Dungeon\HasHealth"
}
array(0) {
}
```

La fonction `class_uses()` liste les traits d'une classe. Elle regarde uniquement la classe qu'on lui donne, jamais ses parents. `Goblin` utilise pourtant bien les méthodes de `HasHealth`, mais le trait est collé dans `Monster`, pas dans `Goblin`.

> **ATTENTION**
>
> **Un trait n'est pas un parent.** N'utilisez pas un trait pour dire ce qu'une chose est. Un trait permet de réutiliser du code. Une interface indique les méthodes attendues. Une classe abstraite peut faire les deux pour une même famille.

UML n'a pas de trait : la notation date d'avant PHP 5.4. Le cours le dessine avec le mécanisme prévu par UML pour ce genre de cas, un **stéréotype**. Le trait est une boîte de classe marquée `«trait»`, avec ses propriétés et ses méthodes, et chaque classe qui le colle lui est reliée par une **dépendance** -----≫, un trait pointillé à pointe ouverte, étiqueté `«use»`. Pas de triangle : ce n'est ni un héritage, ni un contrat.



Trois flèches, trois idées. Le triangle plein ◁——— dit « est un ». Le triangle pointillé ◁- - - - dit « sait faire ». La pointe ouverte pointillée -----≫ dit « se sert de ». La boîte du trait garde ses méthodes visibles : rien n'est perdu, et rien n'est faux.

5. L'enum : trois valeurs, pas une de plus

Retour au code du début de chapitre. La rareté d'un objet n'est pas un texte libre, c'est un choix parmi trois.

NOTE

Histoire de PHP. Les **enums** arrivent avec **PHP 8.1, en 2021**. Avant, les développeurs écrivaient des constantes de classe, et le type restait une chaîne non contrôlée :

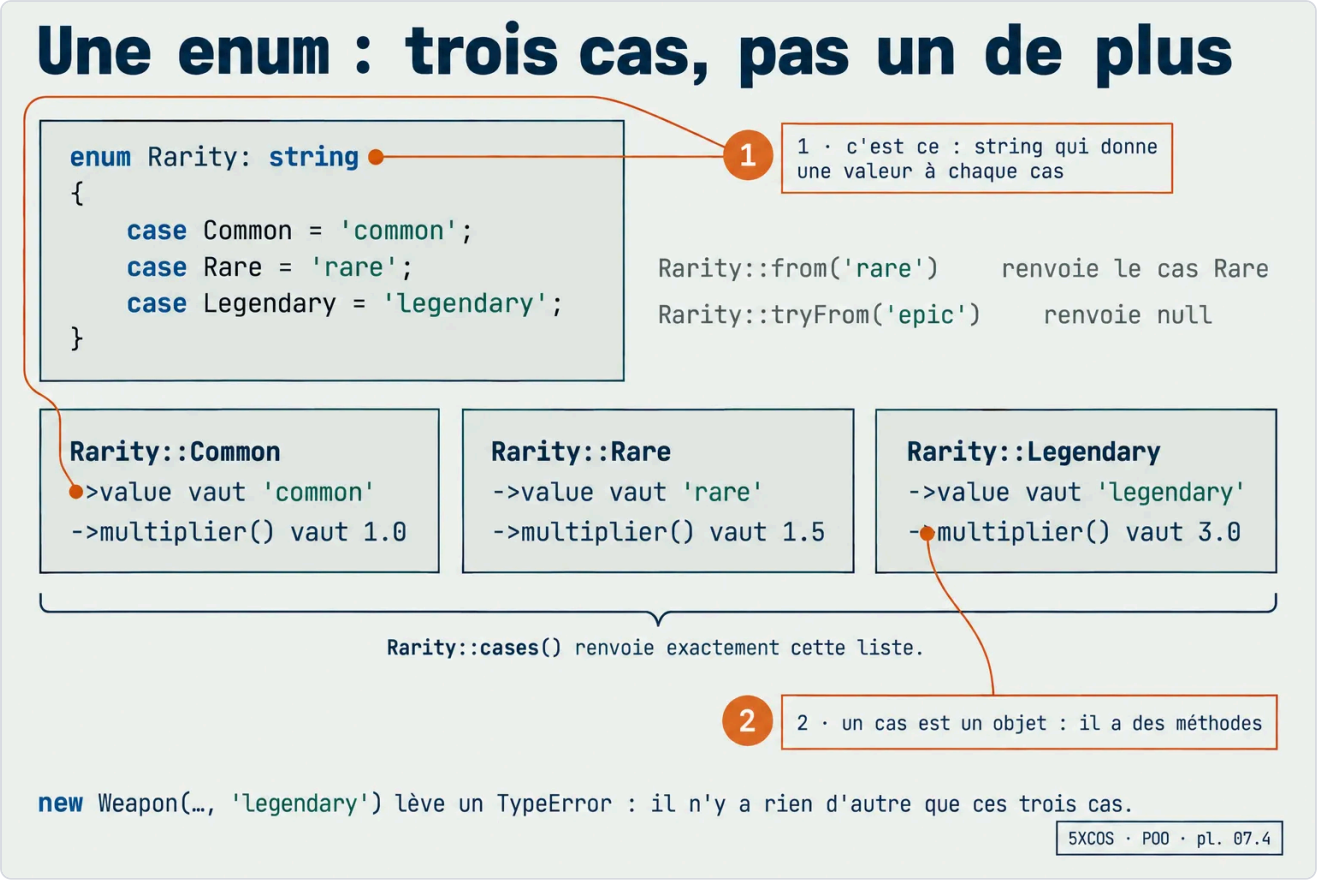
PHP

```php
class Rarity
{
    const COMMON = 'common';
    const RARE = 'rare';
}
```

`match` date de **PHP 8.0**. Avant lui, `switch` faisait le travail, avec une comparaison non stricte, des `break` à ne pas oublier et aucune valeur de retour.

PHP

```php
enum Rarity: string
{
    case Common = 'common';
    case Rare = 'rare';
    case Legendary = 'legendary';

    /** Combien vaut un objet de cette rareté, par rapport à un objet banal. */
    public function multiplier(): float
    {
        return match (this) {
            Rarity::Common ⇒ 1.0,
            Rarity::Rare ⇒ 1.5,
            Rarity::Legendary ⇒ 3.0,
        };
    }

    /** Le libellé affiché au joueur. */
    public function label(): string
    {
        return match (this) {
            Rarity::Common ⇒ 'Commun',
            Rarity::Rare ⇒ 'Rare',
            Rarity::Legendary ⇒ 'Légendaire',
        };
    }
}
```



La dernière ligne de la planche anticipe la fin de la section : `Weapon` attendra un `Rarity`, pas un texte. `'legendary'` est une chaîne, `Rarity::Legendary` est un cas de l'enum. PHP refuse la première avec un `TypeError`, comme il refuse un `string` là où un `int` est attendu.

Avec `: string` après le nom, chaque cas porte une valeur texte. Cette forme s'appelle une enum à valeurs associées. C'est cette valeur qui s'enregistre en base de données ou dans un fichier JSON. Sans `: string`, l'enum est simple, avec des cas nommés et rien d'autre.

Une enum est une classe un peu particulière : ses cas sont des objets, et elle peut avoir des méthodes. `match ($this)` va naturellement avec. Il compare strictement, il renvoie une valeur, et il refuse de deviner quand un cas manque. Ajoutez `Cursed` sans compléter les deux `match` :

```php
enum Rarity: string
{
    case Common = 'common';
    case Rare = 'rare';
    case Legendary = 'legendary';
    case Cursed = 'cursed';

    // … multiplier() et label() inchangés, sans ligne pour Cursed
}

echo Rarity::Cursed->multiplier();
```

```
Fatal error: Uncaught UnhandledMatchError: Unhandled match case Dungeon\Rarity::Cursed
```

Comparez avec le `if` du début de chapitre, qui renvoyait `1.0` en silence. Ici, PHP s'arrête et nomme le cas oublié.

```php
r=Rarity::Legendary;echor = Rarity::Legendary;
echo r=Rarity::Legendary;echor->name, ' / ', r->value,'/',r->value, ' / ', r->value,'/',r
->label(), ' / x', r->multiplier(),PHPEOL;foreach(Rarity::cases()asr->multiplier(), PHP_EOL;

foreach (Rarity::cases() as r->multiplier(),PHPEOL;foreach(Rarity::cases()ascase) {
    printf('%-10s %-10s x%.1f%s', case->value,case->value, case->value,case->label(), $case
->multiplier(), PHP_EOL);
}

var_dump(Rarity::from('rare'));
var_dump(Rarity::tryFrom('epic'));
```

```
Legendary / legendary / Légendaire / x3
common     Commun     x1.0
rare       Rare       x1.5
legendary  Légendaire x3.0
enum(Dungeon\Rarity::Rare)
NULL
```

- `->name` est le nom du cas, `->value` sa valeur associée. Ne les confondez pas.
- `cases()` donne la liste complète, dans l'ordre de déclaration. Plus besoin de maintenir un tableau à côté, qui finirait désynchronisé.
- `from('rare')` retrouve le cas correspondant. `from('epic')` s'arrête sur `ValueError: "epic" is not a valid backing value for enum Dungeon\Rarity`.
- `tryFrom('epic')` renvoie `null` au lieu de lever. Utilisez `tryFrom()` quand la donnée vient de l'extérieur, `from()` quand elle vient de votre base et qu'une valeur inconnue est un vrai bug.

`Item` reçoit sa rareté en quatrième paramètre, avec `Common` par défaut, et `Weapon` et `Potion` la passent au parent :

```php
abstract class Item implements \Stringable
{
    public function __construct(
        public readonly string name,publicreadonlyfloatname,
        public readonly float name,publicreadonlyfloatweight,
        public readonly Rarity rarity = Rarity::Common,
    ) {
        // … la validation du poids du chapitre 5
    }

    /** Règle du jeu, arbitraire : la valeur marchande dépend du poids et de la rareté. */
    public function value(): float
    {
        return $this->weight * $this->rarity->multiplier();
    }

    // … describe() et __toString()
}

final class Weapon extends Item
{
    public function __construct(string name, float weight,publicreadonlyintweight, public
readonly int weight,publicreadonlyintdamage, Rarity $rarity = Rarity::Common)
    {
        parent::__construct($name, $weight, $rarity);
    }

    // … describe()
}
```

Le problème du début disparaît. Le paramètre est typé `Rarity`, donc une chaîne est refusée :

```php
new Weapon('Épée courte', 2.0, 5, 'legendary');
```

```
Fatal error: Uncaught TypeError: Dungeon\Weapon::__construct(): Argument #4 ($rarity) must
be of type Dungeon\Rarity, string given
```

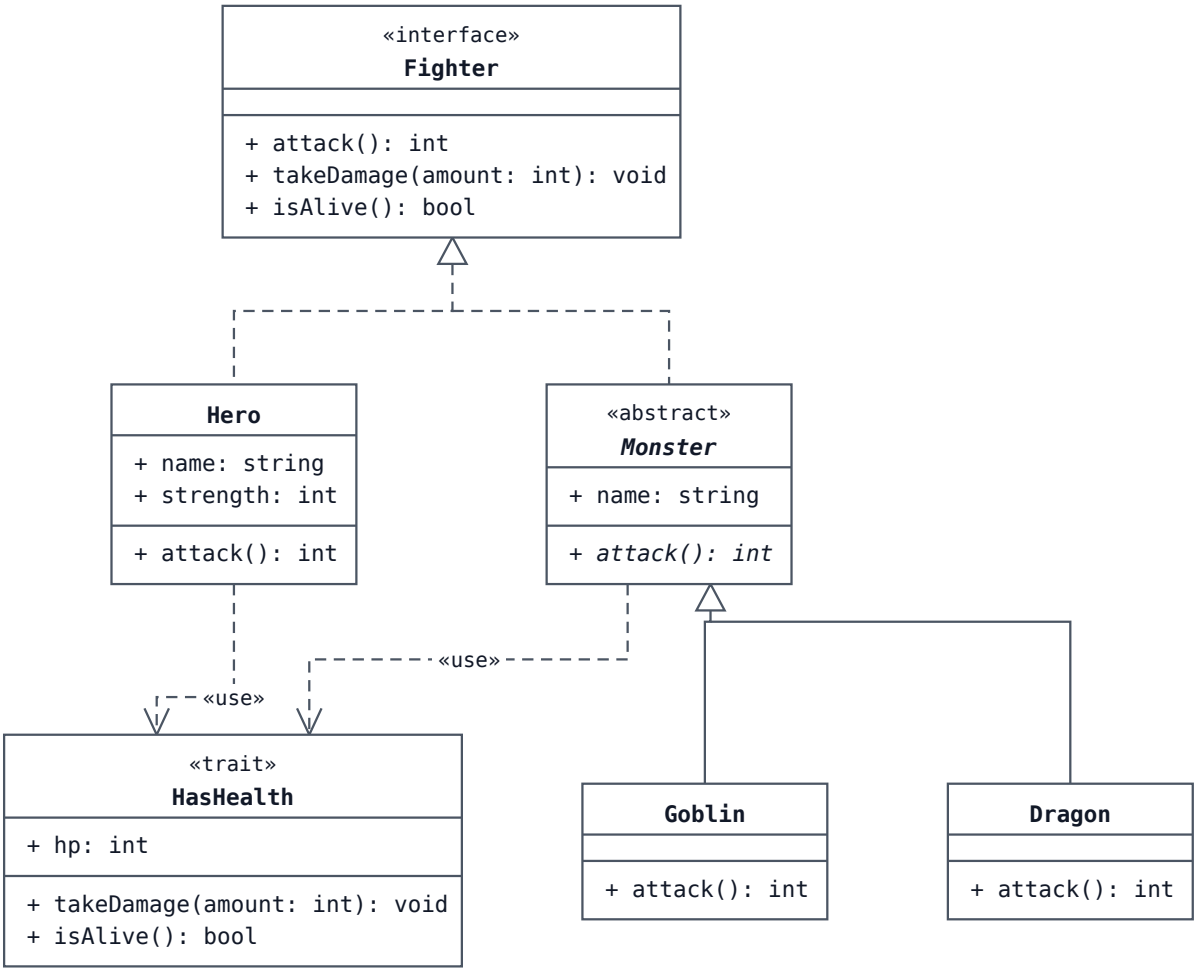
Il faut écrire `Rarity::Legendary`, et l'éditeur propose les trois cas.

6. En UML : réalisation, interface, enum

- La **réalisation** ◁- - - - dit « cette classe implémente cette interface ». C'est un trait **pointillé** terminé par un **triangle vide**, collé à l'interface. La généralisation ◁——— du chapitre 6 a le même triangle, mais un trait plein.
- Le mot-clé `interface` affiche `«interface»` au-dessus du nom.
- Le mot-clé `enum` affiche `«enumeration»`, avec les cas dans le compartiment des attributs.

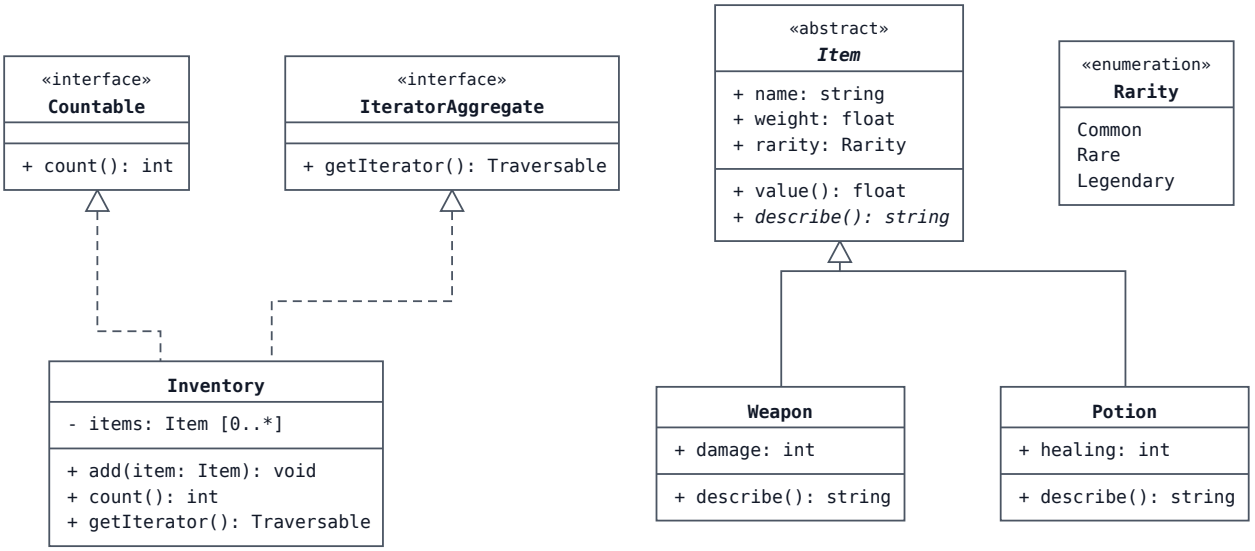
- Un **trait** est une boîte de classe stéréotypée `«trait»`. La **dépendance** `-----≫` `«use»`, pointillée à pointe ouverte, va de la classe vers le trait qu'elle colle.

Le premier diagramme montre le contrat de combat et le trait de santé. Le héros, son inventaire et son arme sont dessinés au chapitre 5.



Un héros et un monstre **savent se battre** et **se servent** du même trait de santé. Un gobelin et un dragon **sont des** monstres. `Monster` réalise `Fighter` sans écrire `attack()` : ses classes filles remplissent le contrat pour elle, et `takeDamage()` comme `isAlive()` lui viennent de `HasHealth`.

Le second diagramme montre les objets, leur rareté et les deux interfaces natives de l'inventaire.



Un inventaire **se compte** et **se parcourt**. Le losange vide ◇——— qui le relie aux objets est dessiné au chapitre 5 : ici, seule la propriété `-items: Item [0..*]` le rappelle. Chaque objet porte une rareté : elle se lit dans la propriété `+rarity: Rarity`, sans trait vers l'enum. Une enum est un type de valeur, comme `string` ou `int`, pas une classe à laquelle on s'associe. C'est la convention du chapitre 4 pour `Condition` dans `Copy`.

7. Ce que vous verrez dans Laravel

Ces trois mots-clés sont partout dans le framework.

```php
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use HasFactory, Notifiable;

    protected function casts(): array
    {
        return ['role' => UserRole::class];
    }
}
```

- `use HasFactory, Notifiable;` : deux traits, collés dans la classe. `HasFactory` apporte les méthodes qui fabriquent des `User` de test, `Notifiable` apporte `notify()`. Aucun des deux ne dit ce qu'un `User` est : ils lui ajoutent des méthodes. Et ils ne peuvent pas être des parents, parce que `User` en a déjà un.
- `extends Authenticatable` : l'héritage du chapitre 6. Les trois lignes `use` du haut du fichier importent des **noms** : deux traits et une classe. Ce qui colle les traits dans `User`, c'est la ligne `use HasFactory, Notifiable;` écrite à l'intérieur des accolades. `Authenticatable` désigne la classe

`Illuminate\Foundation\Auth\User`, renommée à l'import. Cette classe implémente plusieurs interfaces, et c'est en signant ces contrats que votre `User` devient utilisable par le système de connexion.

- `'role' ⇒ UserRole::class` : une conversion de valeur. `UserRole` est une enum de votre application. En base, la colonne contient `'admin'`. En PHP, `$user→role` est un cas de l'enum, avec ses méthodes. La couche base de données de Laravel fait le `from()` et le `→value` dans les deux sens. C'est exactement `Rarity`, avec la base de données autour.

Quand vous verrez que `Collection` implémente `Enumerable`, une interface qui prolonge `Countable` et `IteratorAggregate`, vous saurez pourquoi `count($users)` et `foreach ($users as $user)` fonctionnent sur un objet.

À retenir

- Une **interface** est un contrat : que des signatures de méthodes publiques, pas de code, et autant d'interfaces que nécessaire par classe. Typez les paramètres par l'interface, comme `Fighter $target`, pas par la classe concrète ni par une liste `Hero|Monster`.
- Les **interfaces natives** relient vos objets à la syntaxe du langage : `Countable` pour `count()`, `IteratorAggregate` et `ArrayIterator` pour `foreach`, `Stringable` pour `echo`.
- Classe abstraite quand il y a du **code et une famille** à partager. Interface quand il n'y a qu'un **rôle** à déclarer. Souvent les deux ensemble.
- Un **trait** est du code recopié dans la classe, pas un type et pas un parent. En UML, une boîte `«trait»` et une dépendance `«use»` `- - - - - ⟩`, jamais un triangle. `class_uses()` ne remonte pas aux parents.
- Une **enum** remplace une chaîne libre par une liste fermée de cas, avec des méthodes et un `match` complet. `from()` lève une erreur, `tryFrom()` renvoie `null`.

Trois outils qui se ressemblent, trois rôles :

	`interface Fighter`	`abstract class Monster`	`trait HasHealth`
Ce que c'est	un contrat : des signatures, pas de code	une famille, avec du code en commun	du code réutilisable, sans famille
Ce que la classe écrit	`implements Fighter`	`extends Monster`	`use HasHealth;`
Combien par classe	autant que nécessaire	un seul parent	autant que nécessaire
En UML	trait pointillé, triangle vide ◁- - - -	trait plein, triangle vide ◁———	boîte `«trait»`, pointillé à pointe ouverte `«use»` `- - - - - ⟩`
Type utilisable dans une signature	oui, `Fighter $target`	oui, `Monster $target`	non

Exercices

Sur papier

1. Interface, classe abstraite ou trait ? Un `Trap` et une `Potion` qui savent tous deux **blesser** un héros. Un `Skeleton` qui est un monstre. Un code de **journal des actions** que `Hero`, `Monster` et `Room` doivent tous porter. Justifiez chaque choix par la question de la section 2.
2. `class_uses(Goblin::class)` rend un tableau vide alors que `$goblin->takeDamage(2)` fonctionne. Expliquez en deux phrases.
3. Le chapitre 4 a laissé l'état d'un exemplaire de bibliothèque en attente : neuf, bon ou usé. Écrivez l'enum `Condition` avec une méthode `label()`, puis la propriété qui la porte dans `Copy`.

Kata niveau 4

Dans https://github.com/opmvpc/poo-katas-26 :

```bash
composer test -- --group=niveau-4
```

Les déclarations `implements`, l'interface `Fighter` et les trois cas de `Rarity` sont déjà dans les fichiers fournis. Vous écrivez les corps des méthodes, le trait, et la rareté dans `Item`.

Ce que les tests vérifient :

Ce qui est testé	Détail vérifié
`enum Rarity: string`	`cases()` a trois cas. `Rarity::Common->value` vaut `'common'`. `Rarity::from('rare')` rend `Rarity::Rare`. `Rarity::tryFrom('inconnu')` rend `null`.
`Rarity::multiplier(): float`	`1.0`, `1.5`, `3.0`
`Rarity::label(): string`	`'Commun'`, `'Rare'`, `'Légendaire'`
`Item::$rarity`	`public readonly`, vaut `Rarity::Common` par défaut
`Item::value(): float`	poids × multiplicateur : `2.0` pour une épée commune de 2 kg, `3.0` pour une lame rare de 2 kg, `3.0` pour un élixir légendaire de 1 kg
`Item::__toString(): string`	`(string) $sword` vaut exactement `$sword->describe()`
`interface Fighter`	`new Hero('Arthur')` et `new Goblin()` sont des `Fighter`. `is_subclass_of(Monster::class, Fighter::class)` vaut `true`.
`Fighter::attack(): int`	un héros de force 2 sans arme, un gobelin et un dragon : la somme des trois `attack()` vaut `12`
`trait HasHealth`	`class_uses(Hero::class)` **et** `class_uses(Monster::class)` le contiennent
`HasHealth::takeDamage()`	après `takeDamage(100)`, un héros de 10 PV et un gobelin ont `hp` à `0` et `isAlive()` à `false` : le hook `set` a suivi le déplacement

Ce qui est testé	Détail vérifié
`Inventory implements \Countable`	`count($bag)` vaut `2` après deux `add()`
`Inventory implements \IteratorAggregate`	`foreach ($bag as $item)` donne `['Épée courte', 'Potion de soin']`, dans l'ordre d'ajout

ATTENTION

`class_uses()` ne regarde que la classe qu'on lui donne. Écrivez `use HasHealth;` dans `Hero` et dans `Monster`, et nulle part ailleurs : `Goblin` et `Dragon` le reçoivent par héritage.

Bonus, niveau 5 : `Battle`

```bash
composer test -- --group=niveau-5
```

`Battle` fait s'affronter deux `Fighter` tour par tour. C'est la réponse à la question du chapitre 6 : un dé reçu de l'extérieur se remplace par un dé truqué dans les tests. Ce dé truqué, `FixedDice`, est fourni dans `tests/Support/`.

Ce qui est testé	Détail vérifié
`Battle::__construct(Fighter $a, Fighter $b, Dice $dice)`	le dé est reçu, jamais fabriqué dans `Battle`
`Battle::fight(): Fighter`	renvoie le vainqueur. `$a` frappe en premier, chaque coup vaut `attack()` plus un lancer, le combat s'arrête dès qu'un des deux tombe.
`FixedDice`	`new FixedDice([1, 3])` rend `1`, `3`, puis `1` à nouveau
trois combats	Arthur (20 PV, force 3) bat le gobelin avec un dé à `1` et finit à `17` PV. Arthur (10 PV, force 1) perd contre le dragon. Le même Arthur gagne avec une arme de 20 dégâts.

Pour aller plus loin

1. Dans un fichier à part, hors du dépôt, copiez l'enum `Rarity` et ajoutez-lui un cas `Cursed`, multiplicateur `0.5`, libellé « Maudit ». Appelez `multiplier()` et `label()` sur ce cas. Combien de `match` avez-vous dû corriger, et à quel moment l'erreur est-elle apparue ? Ne touchez pas au fichier du kata : un test compte les cas.
2. Faites implémenter `\JsonSerializable` à `Inventory`, dans le dépôt. `json_encode($bag, JSON_UNESCAPED_UNICODE)` sur une épée courte et une potion de soin doit sortir `[{"name":"Épée courte","weight":2,"value":2},{"name":"Potion de soin","weight":0.5,"value":0.5}]`.

3. Dans un fichier à part, reprenez la `Room` du chapitre 5 et donnez-lui une liste d'objets au sol au lieu d'un seul `loot`. Faites-lui implémenter `\Countable` et `\IteratorAggregate`, puis écrivez le `foreach` qui affiche son butin, un objet par ligne.

2.8 Atelier : dessiner avec des objets

Séance 4 · ~5 périodes

Slides du chapitre (PDF) PDF

Les chapitres 6 et 7 ont donné l'héritage, les classes abstraites et les interfaces sur le Donjon. Cet atelier les fait servir à autre chose : une petite bibliothèque de formes géométriques, puis un moteur de rendu qui transforme ces objets en image SVG. Les tests sont fournis. Ils décrivent exactement ce que votre code doit faire.

À la fin de ce chapitre, vous serez capables de :

1. **construire** une hiérarchie de classes à partir d'un contrat de tests.
2. **choisir** entre hériter d'une classe et en posséder une, et justifier ce choix.
3. **implémenter** une interface pour produire deux formats de sortie à partir des mêmes objets.
4. **produire** un fichier SVG à partir d'objets PHP et l'ouvrir dans un navigateur.

Le dépôt : https://github.com/opmvpc/formes-26. Forkez-le maintenant. Son code vit dans le namespace `Shapes`.

1. Le code qui sent mauvais

Cinq minutes, en binôme. Cet extrait vient d'un ancien projet de départ. Ce code sent mauvais : un détail d'écriture y trahit un défaut de conception. Lequel ?

PHP

```php
class SvgRenderer implements Renderer
{
    public function render(): string { /* fabrique le texte SVG */ }
    public function save(string path): void { /* écrit le .svg */ }
}

class JpgRenderer extends SvgRenderer implements Renderer
{
    public function save(string path): void { /* écrit le .jpg */ }
}
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/atelier-formes

2. Pourquoi ces objets-là

La question du chapitre 6 se pose devant tout `extends` : la phrase « un X est un Y » est-elle vraie en dehors du code ? Éviter de recopier trois méthodes ne suffit pas à justifier un héritage. Le dessin vectoriel se prête bien à cette question, et chaque notion des trois dernières séances y sert :

- une valeur qui ne change plus : un `Point`, deux coordonnées, `readonly`.
- un héritage vrai : un cercle **est** une forme, un rectangle **est** une forme.
- une classe abstraite utile : toutes les formes ont une aire, chacune la calcule autrement.
- une agrégation : un `Canvas` **contient** des formes, il n'en est pas une.
- une interface : `Renderer`, deux façons de dessiner les mêmes objets.

Et le résultat se regarde : à la fin, vous ouvrez votre dessin dans un navigateur.

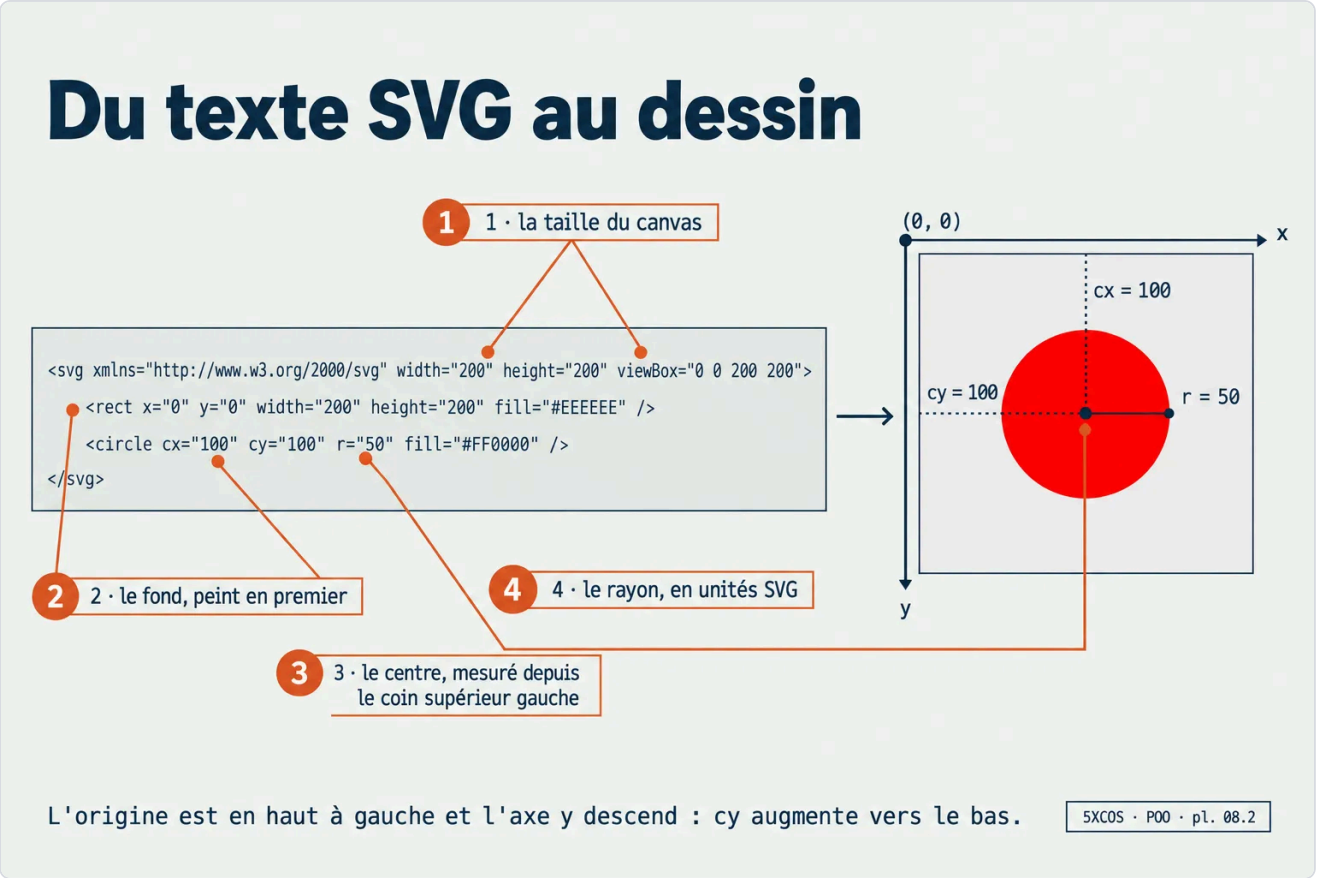
3. Le SVG en cinq balises

Un SVG est un fichier texte. Vous n'allez pas manipuler des pixels. Vous allez écrire des balises, comme en HTML.

```XML
<svg xmlns="http://www.w3.org/2000/svg" width="500" height="500" viewBox="0 0 500 500">
  <rect x="0" y="0" width="500" height="500" fill="#FFFFFF" />
  <line x1="0" y1="0" x2="500" y2="500" stroke="#0000FF" stroke-width="1" />
  <circle cx="250" cy="250" r="80" fill="#00FF00" />
  <rect x="50" y="50" width="250" height="400" fill="#FF0000" />
  <polygon points="250,0 300,200 500,200" fill="#FFFF00" />
</svg>
```

Ces cinq balises suffisent pour tout l'atelier.

> **ATTENTION**
>
> L'origine `(0, 0)` est en haut à gauche et l'axe Y descend. Une figure placée avec les coordonnées d'un repère mathématique apparaît donc retournée.



4. Le dépôt et les commandes

```bash
git clone https://github.com/<votre-compte>/formes-26.git
cd formes-26
composer install
composer test
```

Presque tout est rouge au départ : seules deux vérifications sur `Shape` passent déjà. Les fichiers de `src/` sont des squelettes : les classes existent, les méthodes ont leur signature, et chaque corps à écrire lève `LogicException('À implémenter')`. Trois constructeurs, ceux de `Point`, `SvgRenderer` et `JpgRenderer`, sont à écrire à partir des commentaires. Chaque fichier commence déjà par `declare(strict_types=1)`. Les tests de `tests/` sont complets et ne se modifient pas. Ils sont le cahier des charges.

Une étape à la fois :

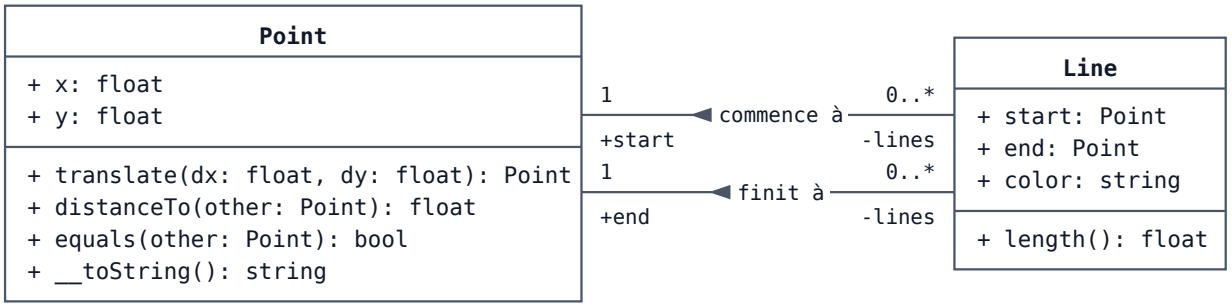
```bash
composer test:etape-1
composer test:etape-2
composer test:etape-3
composer test:etape-4
composer test:bonus
```

Envoyez votre travail sur GitHub à la fin de chaque étape. L'onglet **Actions** de votre fork lance une vérification par étape et affiche une coche par étape, verte ou rouge, plus une pour le bonus.

> **ASTUCE**
>
> Lisez la ligne `Failed asserting that…` avant de modifier votre code. Elle donne la valeur attendue et la valeur obtenue.

5. Étape 1 : `Point` et `Line`

Fourni : `src/Point.php` et `src/Line.php`, avec les signatures des méthodes et des indices en commentaire. **À écrire** : les deux constructeurs et les corps de méthode.



Un point est une valeur, pas un objet qui évolue. Un héros change d'état, il perd des points de vie. Un point, non. `(3, 4)` reste `(3, 4)`. Ce point est **immuable** : il ne change plus après sa construction. `readonly` fait respecter la règle par PHP, et `readonly` devant `class` rend toutes les propriétés `readonly` d'un coup, sans le répéter devant chacune. L'antislash de `\Stringable` dit que l'interface vient de PHP, pas du dossier `Shapes`, comme au chapitre 7.

```php
final readonly class Point implements \Stringable
{
    public function __construct(
        public float x,publicfloatx,
        public float x,publicfloaty,
    ) {}

    public function translate(float dx,floatdx, float dx,floatdy): self
    {
        return new self(this->x+this→x + this->x+dx, this->y+this→y + this->y+dy);
    }

    public function __toString(): string
    {
        return '(' . this->x.',’.this→x . ', ' . this->x.',’.this→y . ')';
    }

    // … distanceTo(), equals()
}
```

`translate()` ne déplace rien. Elle renvoie un nouveau point, et l'original garde ses coordonnées.

```php
depart=newPoint(1,1);depart = new Point(1, 1);
depart=newPoint(1,1);arrivee = depart->translate(4,-1);echodepart→translate(4, -1);

echo depart->translate(4,-1);echodepart, PHP_EOL;
echo $arrivee, PHP_EOL;
```

```
(1, 1)
(5, 0)
```

Toute écriture dans un point déjà construit est refusée :

```php
$depart→x = 10;
```

```
Fatal error: Uncaught Error: Cannot modify readonly property Shapes\Point::$x
```

Ce que les tests attendent

Classe	Comportement vérifié
`Point`	`__construct(float` $x, floaty$`)` : `new Point(3, 4)` donne `3.0` et `4.0`, entiers, décimaux et négatifs acceptés. Une écriture lève `Error`. `translate(4, -1)` sur `(1, 1)` rend `(5, 0)` sans toucher l'original. `distanceTo()` vaut `5.0` entre `(0, 0)` et `(3, 4)`, dans les deux sens, et `0.0` vers soi-même. `equals()` compare les coordonnées. `(string) new Point(10, -3)` vaut `(10, -3)`.

Classe	Comportement vérifié
`Line`	`__construct(Point` $start$, `Point` $Point$`end, string $color = '#000000')`. `$line->start` et `$line->end` rendent les objets reçus, en `public readonly`. `$line->color` vaut `#000000` par défaut, et `#ff0000` devient `#FF0000`. `length()` vaut `5.0` entre `(1, 1)` et `(4, 5)`, et `0.0` si les deux points sont confondus.

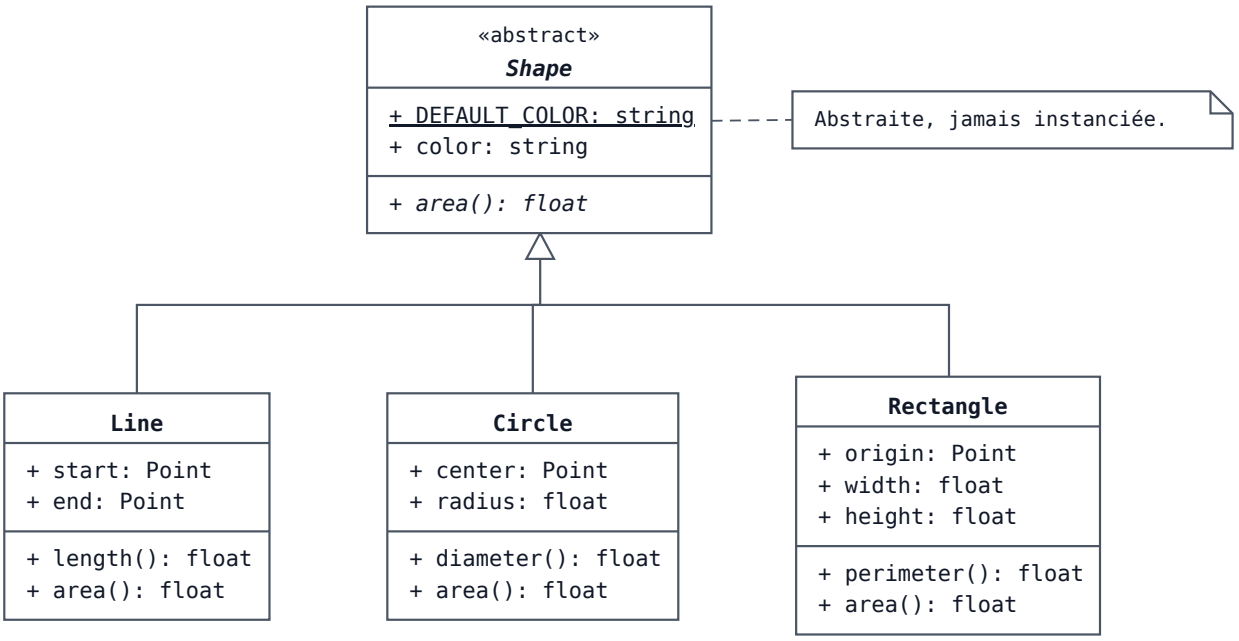
Indice : ne recopiez pas la formule de la distance dans `Line`. `Point::distanceTo()` la connaît déjà.

Vérification : `composer test:etape-1`, 13 tests verts.

Question : deux `Point(2, 7)` distincts sont égaux pour `equals()`, mais `===` répond `false`. Pourquoi ? Le chapitre 5 a la réponse.

6. Étape 2 : `Shape`, `Circle`, `Rectangle`

Fourni : `src/Shape.php` avec sa constante et sa méthode abstraite, `src/Circle.php` et `src/Rectangle.php` avec leurs signatures. **À écrire** : les trois constructeurs, avec leurs validations, et les calculs d'aire. Pas de méthode qui renvoie seulement une propriété : `radius`, `width`, `height` et `color` sont des propriétés `public readonly`, comme au chapitre 3.



Un cercle et un rectangle ont deux choses en commun : une couleur, et une aire. Les deux vont dans la classe parente `Shape`. La couleur a du code, puisque `Shape` la valide et la met en majuscules. L'aire n'en a pas, puisque rien ne se calcule sans savoir de quelle forme il s'agit.

```php
abstract class Shape
{
    public const string DEFAULT_COLOR = '#000000';

    /** La couleur, au format #RRGGBB, toujours en majuscules. Lisible partout, écrite ici une fois. */
    public readonly string color;

    public function __construct(string color = self::DEFAULT_COLOR)
    {
        if (preg_match('/^#[0-9A-Fa-f]{6}/',/', /',color) !== 1) {
            throw new \InvalidArgumentException("Couleur invalide : {color}");
        }

        this->color = strtoupper($color);
    }

    abstract public function area(): float;
}
```

Trois écritures nouvelles. `public const string DEFAULT_COLOR` est une **constante de classe** : une valeur fixée une fois pour toutes, lue avec `self::DEFAULT_COLOR` dans la classe et `Shape::DEFAULT_COLOR` ailleurs. `preg_match()` vérifie qu'un texte suit un motif : celui-ci veut un `#` puis six chiffres hexadécimaux. Et `$color` **n'est pas promu**, c'est voulu : une propriété `readonly` ne s'écrit qu'une fois. Le constructeur vérifie le texte reçu, puis écrit la version en majuscules dans `$this->color`. Une promotion aurait écrit la valeur brute avant toute vérification.

`abstract class` interdit d'instancier `Shape` :

```php
new Shape('#FF0000');
```

```
Fatal error: Uncaught Error: Cannot instantiate abstract class Shapes\Shape
```

`abstract public function area(): float;` n'a pas de corps. C'est une obligation faite aux classes filles, vérifiée par PHP au chargement du fichier, comme au chapitre 6.

Revenez ensuite sur `Line`. Elle aussi est une forme. Faites-la hériter de `Shape`, supprimez sa couleur en double et appelez `parent::__construct($color)` :

```php
final class Line extends Shape
{
    public function __construct(
        public readonly Point start,publicreadonlyPointstart,
        public readonly Point start,publicreadonlyPointend,
        string $color = self::DEFAULT_COLOR,
    ) {
        parent::__construct($color);
    }

    public function area(): float
    {
        return 0.0;
    }

    // … length()
}
```

Relancez les tests de l'étape 1 : ils doivent rester verts pendant que vous déplacez ce code.

> **ATTENTION**
>
> `area()` d'une ligne renvoie `0.0`. Un segment n'a pas de surface. Le contrat n'est donc pas parfait, et vous l'acceptez pour n'avoir qu'une seule classe parente.

Ce que les tests attendent

Classe	Comportement vérifié
`Shape`	La classe est abstraite, `area(): float` aussi. `Line`, `Circle` et `Rectangle` en héritent. Couleur par défaut `#000000`, `public readonly`. Les valeurs `rouge`, `#FFF`, `000000`, `#GGGGGG` et la chaîne vide lèvent `InvalidArgumentException`.
`Circle`	`__construct(Point` $center$, $float$`radius, string $color = '#000000')`. `$circle→center` et `$circle→radius` sont `public readonly`. `diameter()` vaut le double du rayon, `area()` vaut `M_PI`, la constante de PHP qui contient π, pour un rayon de 1. Un rayon nul ou négatif lève `InvalidArgumentException`.
`Rectangle`	`__construct(Point` $origin$, $float$`width, float` $height$, $string$`color = '#000000')`. `area()` vaut `20.0` et `perimeter()` vaut `18.0` pour un rectangle de 4 sur 5. Une dimension nulle ou négative lève `InvalidArgumentException`.

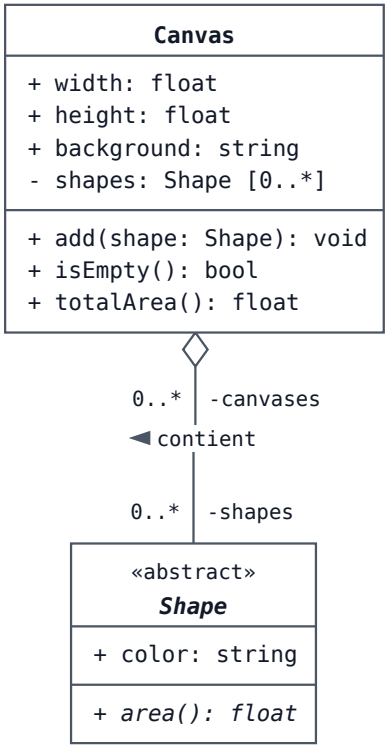
Indice : une classe fille qui définit son propre constructeur doit appeler `parent::__construct()` elle-même. PHP ne le fait pas à sa place.

Vérification : `composer test:etape-2`, 26 tests verts, et `composer test:etape-1` toujours vert.

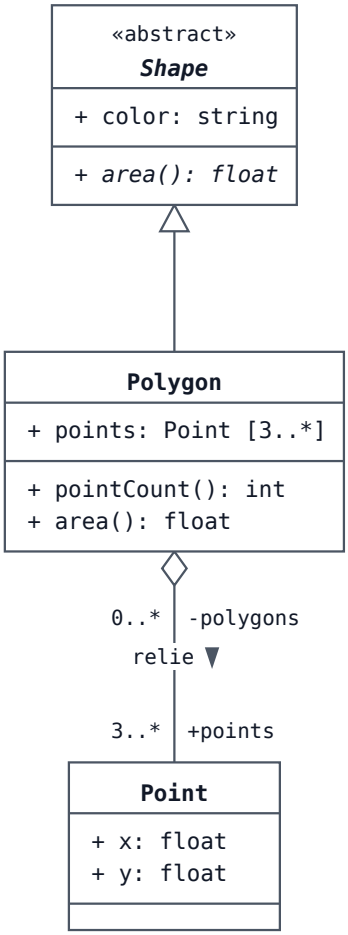
Question : la couleur est validée et stockée dans `Shape`, une fois pour toutes, alors que `area()` y reste abstraite. Sur quel critère une méthode reste-t-elle abstraite plutôt que d'être écrite dans le parent ?

7. Étape 3 : `Canvas` et `Polygon`

Fourni : `src/Canvas.php` avec la propriété `$shapes` déjà déclarée, `src/Polygon.php` avec la formule du lacet décrite en commentaire. **À écrire** : les deux constructeurs avec leurs validations, `add()`, `isEmpty()`, `totalArea()`, `pointCount()` et `Polygon::area()`.



Le losange vide ◇——— entre `Canvas` et `Shape` se lit comme au chapitre 5 : le canvas tient la liste des formes, mais les formes sont construites à l'extérieur et lui sont données par `add()`. Une même forme peut aller dans deux canvas, et elle survit à la disparition de l'un. Entre `Canvas` et `Shape`, il n'y a pas de triangle : un canvas n'est pas une forme, il en contient. `class Canvas extends Shape` s'écrit sans erreur, relisez la phrase à voix haute, elle ne tient pas.



Même losange vide entre `Polygon` et `Point`. Les sommets sont construits à l'extérieur, puis donnés au polygone. Comme un `Point` est immuable, le partager ne présente aucun risque.

`add()` ne contient aucun `if` pour refuser un `Point`. Le type du paramètre s'en charge :

```php
public function add(Shape $shape): void
{
    $this->shapes[] = $shape;
}
```

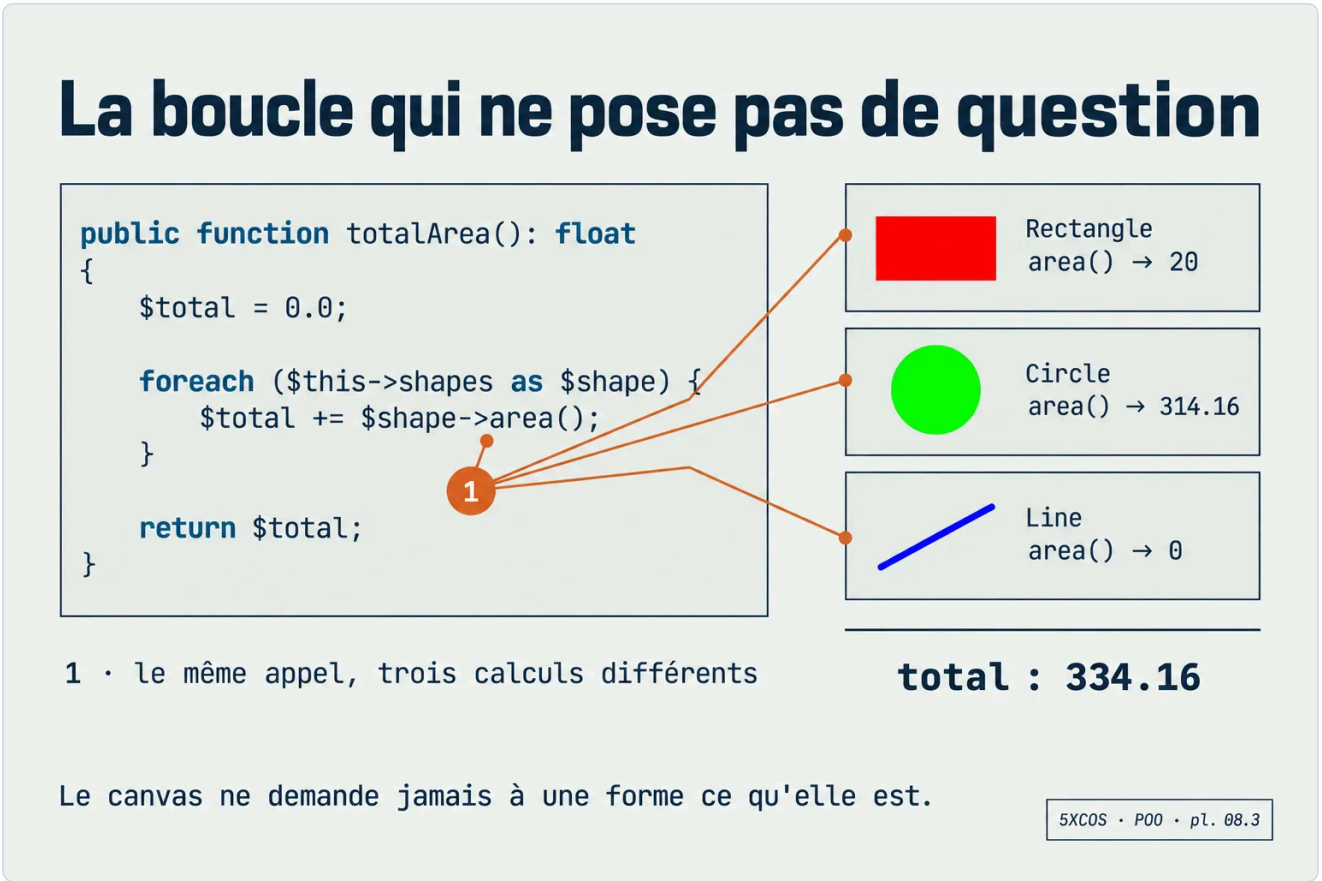
```php
$canvas->add(new Point(0, 0));
```

```
Fatal error: Uncaught TypeError: Shapes\Canvas::add(): Argument #1 ($shape) must be of type
Shapes\Shape, Shapes\Point given
```

`totalArea()` est la méthode qui justifie la hiérarchie de l'étape 2 :

```php
public function totalArea(): float
{
    total=0.0;foreach(total = 0.0;

    foreach (total=0.0;foreach(this->shapes as shape) {
        $total += $shape->area();
    }

    return total;
}
```

Le canvas ne demande jamais à une forme ce qu'elle est. À chaque tour de boucle, PHP appelle la méthode `area()` de l'objet parcouru. C'est le polymorphisme du chapitre 6, et il évite ici d'écrire un cas par forme.



Ce que les tests attendent

Classe	Comportement vérifié
`Canvas`	`__construct(float` *width*, *float*`height, string $background = '#FFFFFF')`. Le fond ressort en majuscules. Un canvas neuf a `$canvas->shapes` vide et `isEmpty()` à `true`. `$canvas->shapes` se lit partout et ne se remplace pas : `$canvas->shapes = []` lève `Error`. `add()` conserve l'ordre. `add(new Point(0, 0))` lève `TypeError`. `totalArea()` vaut `40.0` pour deux rectangles de 20 et une ligne. Une taille nulle ou négative lève `InvalidArgumentException`.

Classe	Comportement vérifié
`Polygon`	`__construct(array` $points$, $string$`color = '#000000')`, et `Polygon` hérite de `Shape`. `$polygon->points` garde l'ordre, `pointCount()` les compte. `area()` vaut `6.0` pour le triangle `(0,0) (4,0) (0,3)` et `100.0` pour un carré de côté 10. Moins de trois sommets, ou un élément qui n'est pas un `Point`, lève `InvalidArgumentException`.

Indice : le type `array` de PHP ne dit pas ce qu'il contient. C'est à vous de parcourir le tableau reçu par `Polygon` et de vérifier chaque élément avec `instanceof Point`.

Vérification : `composer test:etape-3`, 18 tests verts.

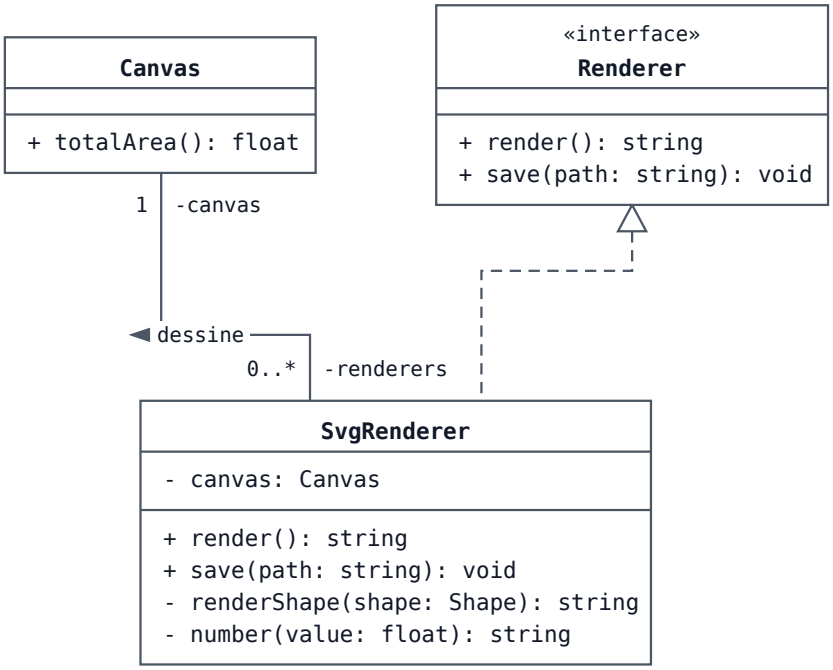
Question : une même forme peut-elle être ajoutée à deux canvas différents ? Que devient-elle si l'un des deux disparaît ? Comparez votre réponse avec la multiplicité `0..*` côté `Canvas`.

8. Étape 4 : `Renderer` et `SvgRenderer`

Fourni : l'interface `src/Renderers/Renderer.php`, complète et à ne pas modifier, et le squelette `src/Renderers/SvgRenderer.php`. **À écrire** : le constructeur, `render()`, `save()`, `renderShape()` et `number()`.

```php
interface Renderer
{
    public function render(): string;

    public function save(string $path): void;
}
```

Deux méthodes, aucune ligne de code. Comme `Fighter` au chapitre 7, l'interface dit ce qu'un moteur de rendu doit savoir faire, pas comment il s'y prend.



Le pointillé ◁----- dit que `SvgRenderer` réalise le contrat. Le moteur de rendu reçoit le canvas dans son constructeur. Il ne le fabrique pas. Il l'utilise pour produire du texte.

`renderShape()` traduit une forme en balise. Elle utilise `match (true)` : PHP compare `true` à chaque condition, dans l'ordre, et s'arrête à la première vraie. C'est une suite de `if/elseif` qui renvoie une valeur, dans la même écriture que le `match` du chapitre 7.

```php
private function renderShape(Shape $shape): string
{
    return match (true) {
        $shape instanceof Line => sprintf(
            '<line x1="%s" y1="%s" x2="%s" y2="%s" stroke="%s" stroke-width="1" />',
            $this->number($shape->start->x),
            $this->number($shape->start->y),
            $this->number($shape->end->x),
            $this->number($shape->end->y),
            $shape->color,
        ),
        // … Circle, Rectangle, Polygon …
        default => throw new \InvalidArgumentException(
            'Forme inconnue du renderer : '.$shape::class,
        ),
    };
}
```

Le `default` lève une exception, parce qu'un moteur de rendu qui rencontre une forme inconnue doit le signaler au lieu de dessiner du vide. `$shape::class` donne le nom complet de la classe de l'objet, comme `get_class()` au chapitre 6.

Le chapitre 6 déconseillait cette cascade de `instanceof`. Elle revient ici, et c'est un choix de conception : le format de sortie n'appartient pas aux formes. Connaître chaque type pour écrire sa balise est le métier du moteur de rendu. La question de fin de section y revient.

Ce que les tests attendent

Le document commence par `<?xml version="1.0" encoding="UTF-8"?>`, contient `xmlns="http://www.w3.org/2000/svg"`, la largeur, la hauteur et le `viewBox`, puis se termine par `</svg>`. Le fond est peint en premier. Les formes suivent dans l'ordre d'ajout. `save()` écrit dans le fichier exactement ce que `render()` renvoie.

Élément	Balise attendue
fond du canvas	`<rect x="0" y="0" width="500" height="500" fill="#00FFFF" />`
`Line`	`<line x1="1" y1="1" x2="1" y2="500" stroke="#0000FF" stroke-width="1" />`
`Circle`	`<circle cx="250" cy="250" r="12.5" fill="#000000" />`
`Rectangle`	`<rect x="50" y="50" width="250" height="400" fill="#FF0000" />`
`Polygon`	`<polygon points="1,1 1,500 500,500" fill="#FF0000" />`

> **ASTUCE**
>
> Les tests attendent `500`, pas `500.0`, et `12.5` tel quel. Une conversion simple, `(string) $value`, donne déjà ces deux résultats. `number()` sert aux calculs qui traînent des décimales, comme `1 / 3` : la solution arrondit à deux décimales avec `number_format()`, puis retire les zéros et le point inutiles avec `rtrim()`.

Vérification : `composer test:etape-4`, 10 tests verts. Lancez ensuite l'exemple fourni.

```bash
php examples/star.php
```

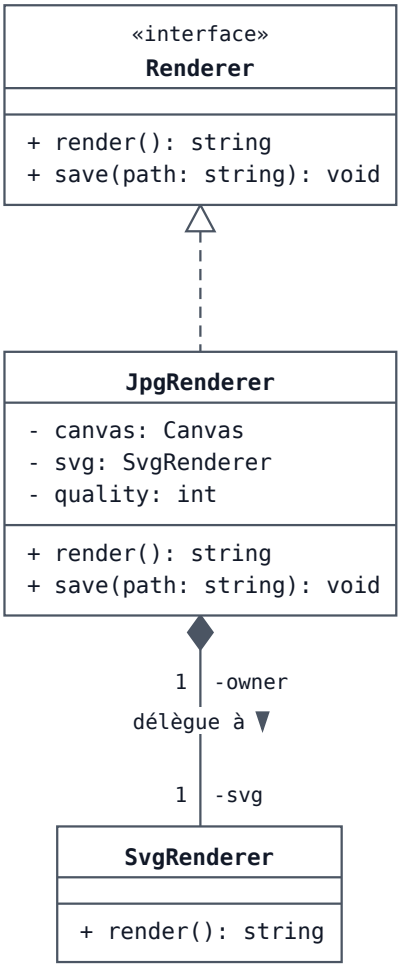
```
Étoile écrite dans examples/star.svg (2 formes, aire totale : 77527 px²).
```

Ouvrez `examples/star.svg` dans votre navigateur.

Question : le `match` de `renderShape()` grandit d'un cas à chaque nouvelle forme. Une autre conception donnerait une méthode `toSvg()` à chaque forme. Que coûterait-elle le jour où le projet ajoute un troisième format de sortie ?

9. Bonus : `JpgRenderer`

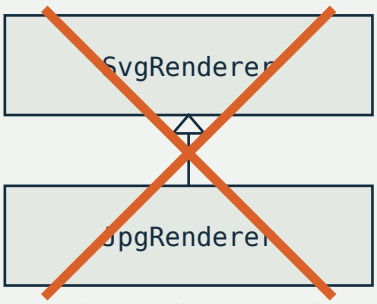
Le bonus reprend le problème posé en ouverture. Vous écrivez la deuxième classe qui remplit le contrat `Renderer`, cette fois avec l'extension **GD** et le paquet `meyfa/php-svg` déjà installé.



`JpgRenderer` **a un** `SvgRenderer` au lieu d'en hériter, le losange plein ◆——— : il fabrique son moteur SVG dans son constructeur et le garde pour lui. Il lui demande le texte SVG, le convertit en image, puis renvoie les octets du JPEG. Deux classes sœurs, pas une classe et sa fille.

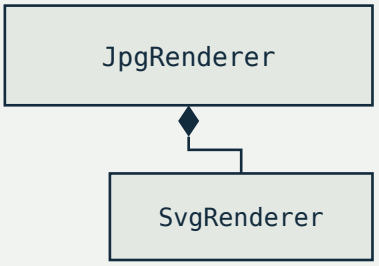
Hériter ou posséder ?

HÉRITAGE REFUSÉ



« JpgRenderer est-il un SvgRenderer ? »
Non : ce ne sont pas la même chose.

ASSEMBLAGE RETENU



JpgRenderer a un SvgRenderer.
`render()` → texte SVG

5XCOS · POO · pl. 08.1 · éch. 1:

```bash
php -m
```

Cherchez `gd` dans la liste. Sans cette extension, les tests du groupe `bonus` sont ignorés et non rouges. Les indices détaillés sont dans le squelette et dans le README du dépôt.

10. Le rendu

1. Les quatre étapes vertes, en local et dans l'onglet Actions de votre fork.
2. `php examples/star.php` écrit l'étoile.
3. Modifiez ensuite `examples/star.php` et dessinez ce que vous voulez. Une maison, un logo, une fractale. Une seule contrainte : au moins trois types de formes différents.
4. Postez votre SVG sur le canal du cours, avec le lien vers votre fork.

À retenir

- `readonly` interdit toute écriture après la construction. Une méthode qui « déplace » un tel objet renvoie une nouvelle instance.
- **Une classe abstraite** met en commun ce qui est partagé et impose le reste par des méthodes abstraites.
- **Le polymorphisme**, c'est un même appel qui exécute un code différent selon l'objet reçu. Dans `totalArea()`, il évite d'écrire un cas par forme.
- **Un héritage se justifie par la phrase « un X est un Y »**, pas par le code économisé. Si elle sonne faux, X a un Y.
- **Une interface est un contrat sans code.** Deux classes sans parenté commune peuvent le remplir.

Exercices

Sur papier

Les quatre questions de fin d'étape se traitent sur papier, avant de coder l'étape suivante : l'égalité de deux points (étape 1), le critère qui laisse `area()` abstraite (étape 2), la forme partagée entre deux canvas (étape 3), le coût d'un troisième format de sortie (étape 4).

Atelier

1. **Obligatoire.** Le rendu de la section 10, en entier.
2. **Au choix.** Ajoutez `src/Ellipse.php`, définie par un centre et deux rayons, et le cas correspondant dans `renderShape()`. Vérifiez avec un dessin dans `examples/star.php` et l'ouverture du SVG. Comptez les fichiers que vous avez dû ouvrir : ce nombre répond à la question de l'étape 4.
3. **Bonus.** `JpgRenderer` vert.

4. **Bonus difficile.** Ajoutez `Canvas::withBackground(string $color): Canvas`, qui renvoie un nouveau canvas avec les mêmes formes et un autre fond, sans modifier l'original. Quelle idée de l'étape 1 réutilisez-vous ?

2.9 Atelier : recodez la Collection de Laravel

Séance 5 · ~2 périodes

Slides du chapitre (PDF) PDF

À la fin de ce chapitre, vous serez capables de :

1. **écrire** une pile, une liste et une collection qui font passer les tests fournis.
2. **lire et écrire** une fonction passée en paramètre, avec la syntaxe courte `fn` comme avec `function`.
3. **justifier** pourquoi une transformation renvoie une nouvelle collection au lieu de modifier l'ancienne.
4. **estimer** le coût d'un bout de code en O(1), O(n) ou O(n²).
5. **lire** une chaîne d'appels `collect(...)→filter(...)→map(...)→sum()` dans la documentation de Laravel.

Le dépôt : https://github.com/opmvpc/collection-26. Forkez-le maintenant. Son code vit dans le namespace `Collections`.

1. Le code qui sent mauvais

Cinq minutes, en binôme. `$items` est le contenu du sac d'Arthur. Le but est d'afficher les objets légers et leur poids total. Ce code sent mauvais : plusieurs détails d'écriture y trahissent des défauts de conception. Lesquels ?

```php
items=[['name'⇒'E´pe´elongue','weight'⇒3.5],['name'⇒'Potiondesoin','weight'⇒0.5],
['name'⇒'Bouclier','weight'⇒6.0],['name'⇒'Torche','weight'⇒1.0],];items = [
    ['name' ⇒ 'Épée longue', 'weight' ⇒ 3.5],
    ['name' ⇒ 'Potion de soin', 'weight' ⇒ 0.5],
    ['name' ⇒ 'Bouclier', 'weight' ⇒ 6.0],
    ['name' ⇒ 'Torche', 'weight' ⇒ 1.0],
];

items=[['name'⇒'E´pe´elongue','weight'⇒3.5],['name'⇒'Potiondesoin','weight'⇒0.5],
['name'⇒'Bouclier','weight'⇒6.0],['name'⇒'Torche','weight'⇒1.0],];total = 0;
noms='';for(noms = '';
for (noms='';for(i = 0; i<count(i < count(i<count(items); i++) {
    if ($items[$i]['weight'] < 5) {
        $total = $total + $items[$i]['weight'];
        $noms = $noms . $items[$i]['name'] . ', ';
    }
}

echo 'Léger : ' . noms . '(' . $total . ' kg)', PHP_EOL;
```

```
Léger : Épée longue, Potion de soin, Torche, (5 kg)
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/atelier-collection

2. Une `Collection` dans Laravel

Dans Laravel, `User::all()` ne renvoie pas un tableau PHP. Il renvoie un objet : une `Illuminate\Database\Eloquent\Collection`, qui hérite de `Illuminate\Support\Collection`. C'est la même chose pour `$post->comments`, pour le résultat d'un `where()->get()`, et pour presque tout ce qui contient plusieurs éléments. La documentation de Laravel suppose que vous savez lire ceci :

```php
noms=noms = noms=users->filter(fn (user)=>user) => user)=>user->isActive())
    ->sortBy(fn (user)=>user) => user)=>user->name)
    ->pluck('name');
```

Pour lire cet exemple, il faut comprendre `filter()`, `sortBy()`, `pluck()` et la syntaxe `fn`. Tant que ces quatre choses restent floues, une grande partie de la documentation reste illisible. L'atelier construit cette `Collection` en trois étapes, puis compare son comportement à celui de Laravel.

3. Le dépôt et les commandes

```bash
git clone https://github.com/<votre-compte>/collection-26.git
cd collection-26
composer install
composer test
```

Tout est rouge au départ. Chaque corps de méthode à compléter dans `src/` lève `LogicException('À implémenter')`. Les constructeurs, `guardIndex()` et la classe `Node` sont déjà écrits. Complétez le reste pour qu'il respecte les tests fournis.

Une étape à la fois :

```bash
composer test -- --group=etape-1
composer test -- --group=etape-2
composer test -- --group=etape-3
composer test -- --group=bonus
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/atelier-collection

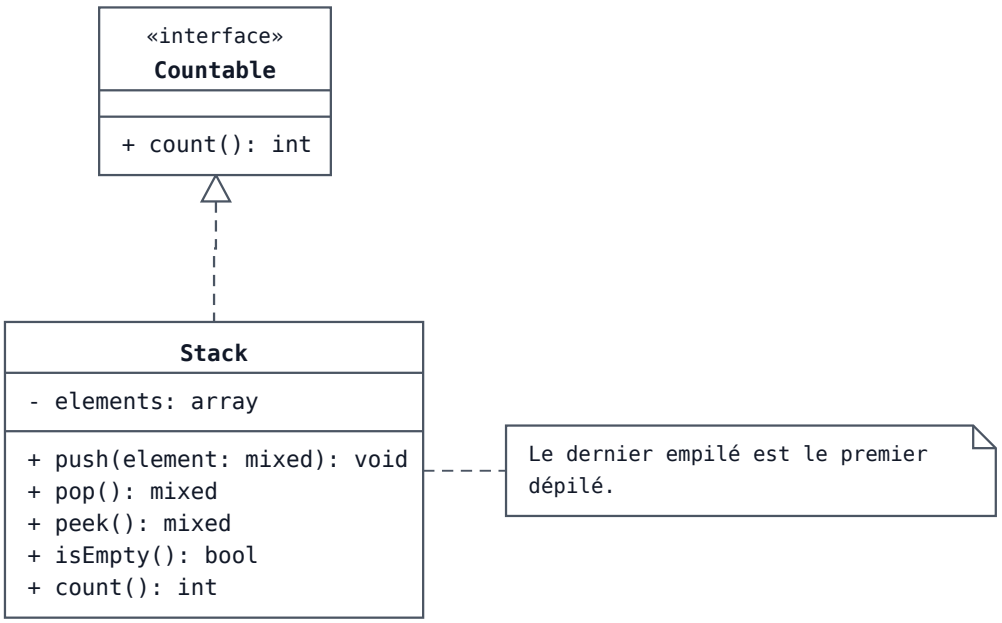
L'onglet **Actions** de votre fork lance une vérification par étape, indépendante des autres. Une étape 1 rouge n'empêche pas l'étape 3 de tourner.

Deux règles, comme au chapitre 8 : ne modifiez pas les tests fournis, et ne touchez pas à `ListInterface`. Les tests sont le cahier des charges, l'interface est le contrat.

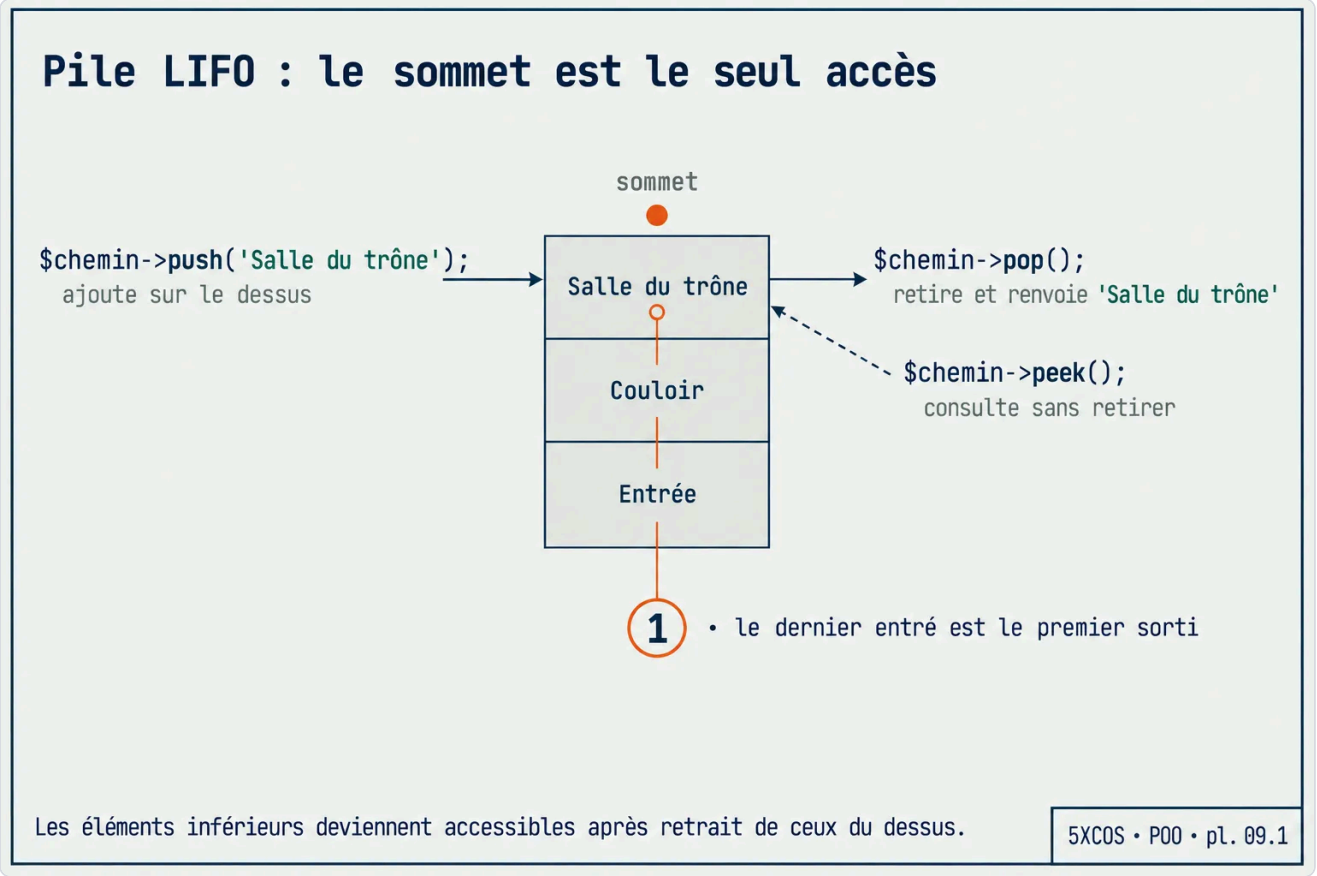
4. Étape 1 : la pile

Fourni : `src/Stack.php`, avec les cinq signatures et les indices en commentaire. **À écrire** : les cinq corps de méthode.

Une pile d'assiettes : vous posez sur le dessus, vous reprenez sur le dessus. Le dernier arrivé est le premier servi. C'est une pile **LIFO**, pour *last in, first out*. Dans le Donjon, c'est l'historique des salles visitées, et faire demi-tour revient à reprendre la dernière.



`mixed` est le type PHP qui accepte n'importe quelle valeur : une pile ne regarde pas ce qu'elle range. `push()` pose sur le dessus, `pop()` retire le dessus et le renvoie, `peek()` le consulte sans le retirer.



Une fois la classe écrite :

```php
visitees=newStack();visitees = new Stack();
visitees=newStack();visitees→push('Entrée');
visitees->push('Couloir');visitees→push('Couloir');
visitees->push('Couloir');visitees→push('Salle du trône');

echo visitees->peek(),PHPEOL;echovisitees→peek(), PHP_EOL;
echo visitees->peek(),PHPEOL;echovisitees→pop(), PHP_EOL;
echo visitees->pop(),PHPEOL;echocount(visitees→pop(), PHP_EOL;
echo count(visitees->pop(),PHPEOL;echocount(visitees), PHP_EOL;
```

```
Salle du trône
Salle du trône
Couloir
1
```

Un mot sur `count($visitees)`. Votre classe déclare `implements Countable` : PHP accepte donc de lui appliquer la fonction native `count()`, qui appelle votre méthode `count()`. Comme au chapitre 7, une interface native ouvre la syntaxe du langage à vos objets.

Ce que les tests attendent

Méthode	Comportement vérifié
`push()`, `count()`	Une pile neuve est vide. Après trois `push()`, `count()` vaut `3`.
`pop()`	Rend les éléments dans l'ordre inverse de l'empilement, puis la pile redevient vide.
`peek()`	Rend le sommet sans le retirer : deux appels de suite donnent la même valeur.
pile vide	`pop()` et `peek()` lèvent `UnderflowException`.
`Countable`	`count($stack)` fonctionne.
contenu	La pile accepte un entier comme un tableau.

`UnderflowException` est une exception de PHP, prête à l'emploi, comme `InvalidArgumentException` au chapitre 3 : elle dit qu'on retire d'un conteneur vide. Les fichiers fournis l'importent déjà en haut.

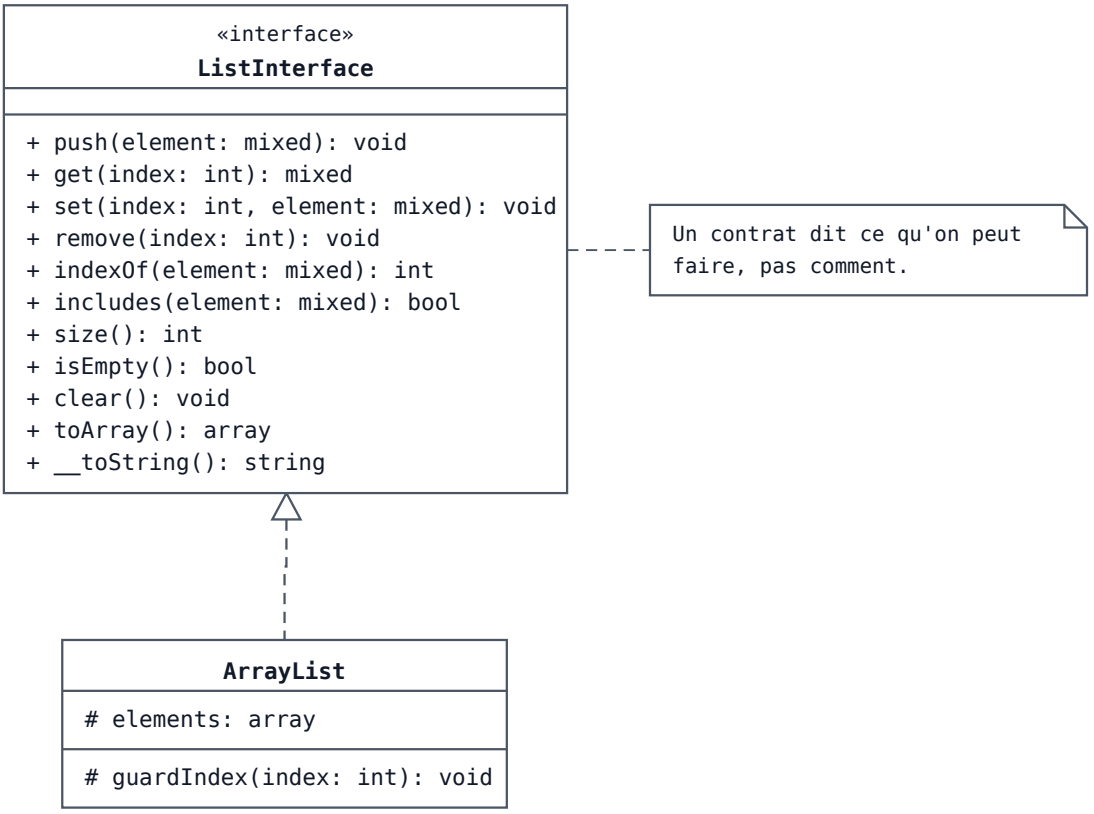
Indice : tout se joue à la fin du tableau. `array_pop()` retire le dernier élément et vous le rend. `array_key_last()` donne l'index du dernier sans y toucher.

Vérification : `composer test -- --group=etape-1`, 9 tests verts.

Question : pourquoi une pile n'a-t-elle pas de méthode `get(int $index)` ? Qu'est-ce que vous gagnez à interdire quelque chose ?

5. Étape 2 : la liste et son contrat

Fourni : l'interface `src/ListInterface.php`, à ne pas modifier, et `src/ArrayList.php` avec ses onze signatures et sa méthode `guardIndex()` déjà écrite. **À écrire** : les onze corps de méthode.



Une liste indexée de 0 à `size() - 1`, sans trou. C'est la contrainte importante : après `remove(1)`, l'ancien index 2 devient l'index 1. Tout ce qui suit se décale. Le `#` devant `elements` et `guardIndex` est la visibilité `protected` du chapitre 6 : la classe qui héritera d'`ArrayList` à l'étape 3 doit lire le tableau.

```php
compagnie=newArrayList();compagnie = new ArrayList();
compagnie=newArrayList();compagnie→push('Arthur');
compagnie->push('Perceval');compagnie→push('Perceval');
compagnie->push('Perceval');compagnie→push('Merlin');
compagnie->remove(1);echocompagnie→remove(1);

echo compagnie->remove(1);echocompagnie→get(1), PHP_EOL;
echo compagnie->size(),PHPEOL;echocompagnie→size(), PHP_EOL;
echo compagnie->size(),PHPEOL;echocompagnie→indexOf('Perceval'), PHP_EOL;
echo $compagnie, PHP_EOL;
```

```
Merlin
2
-1
[
    "Arthur",
    "Merlin"
]
```

Ce que les tests attendent

Méthode	Comportement vérifié
`push()`, `get()`, `set()`	Les éléments s'ajoutent à la fin et se relisent par index.
`remove()`	Retire l'élément et décale les suivants. Aucun trou dans les index.
`indexOf()`	Rend l'index de l'élément, et `-1` s'il est absent. La comparaison est stricte : `"2"` n'est pas `2`.
`includes()`, `size()`, `isEmpty()`, `clear()`, `toArray()`	Les cinq autres méthodes du contrat.
`__toString()`	Exactement la sortie de `json_encode()` avec les options `JSON_PRETTY_PRINT` et `JSON_UNESCAPED_UNICODE`, comme ci-dessus : une valeur par ligne, les accents tels quels.
index invalide	`get()`, `set()` et `remove()` lèvent `OutOfRangeException` hors des bornes, une autre exception prête à l'emploi de PHP.

ATTENTION

Le piège de `indexOf()`. `array_search(`$element$`,this→elements, true)` cherche bien avec `===`, mais renvoie `false` quand l'élément est absent, pas `-1`. Comme la méthode est déclarée `: int` et le fichier `declare(strict_types=1)`, PHP refuse ce retour :

```
Fatal error: Uncaught TypeError: Collections\ArrayList::indexOf(): Return value must be of type int, false returned
```

C'est à vous de traduire le `false` en `-1`. Sans `declare(strict_types=1)` en tête de fichier, PHP convertirait `false` en `0` sans rien dire, et votre liste répondrait que l'absent est en première position.

Indice : `array_splice()` retire un élément et renumérote les suivants. La méthode `guardIndex()` est fournie : appelez-la au début de `get()`, `set()` et `remove()`.

Vérification : `composer test -- --group=etape-2`, 18 tests verts.

Question : `ListInterface` ne contient aucune ligne de code, seulement des signatures. À quoi sert un fichier pareil ?

6. Une fonction passée en paramètre

Les exemples qui suivent travaillent sur le sac d'Arthur. La classe `Item` vient des tests du dépôt, dans `tests/Fixtures/Item.php`. C'est la version courte de l'`Item` du début du chapitre 5 : un nom, un poids, tous deux `public readonly`, sans la validation du poids. Vous écrivez donc `$item→weight` sans accesseur.

```php
final class Item
{
    public function __construct(
        public readonly string name,publicreadonlyfloatname,
        public readonly float name,publicreadonlyfloatweight,
    ) {}
}
```

L'étape 3 repose sur une idée : une fonction est une valeur comme une autre. Vous pouvez la ranger dans une variable ou la passer en paramètre. Une fonction passée à une autre pour être appelée par elle s'appelle un **callback**.

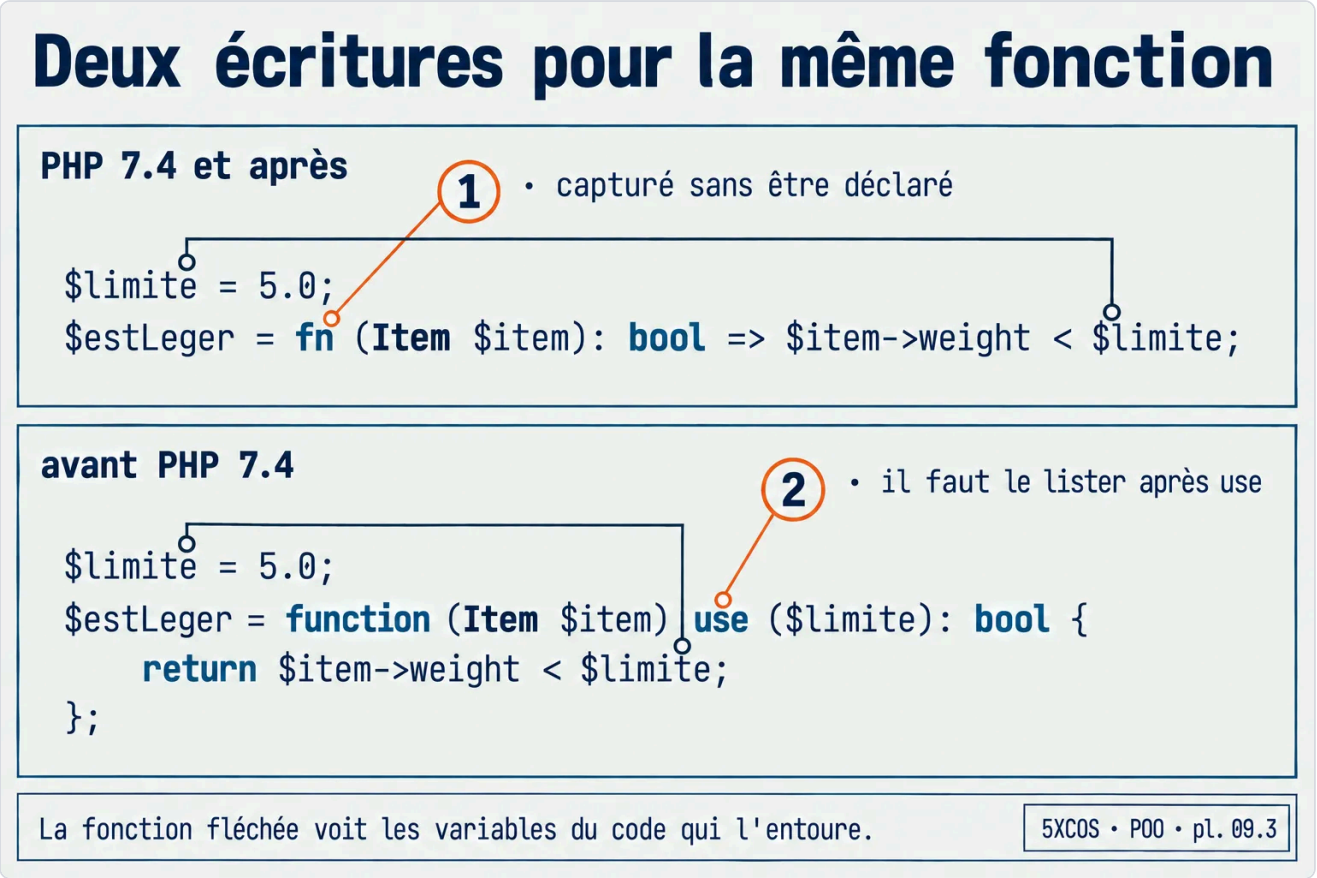
```php
crier=fn(stringcrier = fn (string crier=fn(stringnom): string ⇒ mb_strtoupper(nom).’!
’;echonom) . ' !';

echo nom).’!’;echocrier('Arthur'), PHP_EOL;
```

```
ARTHUR !
```

`fn (paramètres): type ⇒ expression` est la **fonction fléchée**. Une seule expression, dont le résultat est renvoyé sans `return`. C'est la forme que vous verrez partout dans Laravel. `mb_strtoupper()` met en majuscules, lettres accentuées comprises. La fonction fléchée utilise les variables du code qui l'entoure sans avoir à les déclarer :

```php
limite=5.0;limite = 5.0;
limite=5.0;estLeger = fn (Item item):bool⇒item): bool ⇒ item):bool⇒item→weight <
limite;echolimite;

echo limite;echoestLeger(new Item('Torche', 1.0)) ? 'oui' : 'non', PHP_EOL;
echo $estLeger(new Item('Bouclier', 6.0)) ? 'oui' : 'non', PHP_EOL;
```

```
oui
non
```



Dans une signature, le type d'un callback s'écrit `callable` : une valeur que PHP peut appeler comme une fonction. Vous le lirez dans les boîtes UML de l'étape 3.

L'écriture plus ancienne, `function () { ... }`, reste utile quand le corps fait plusieurs lignes. Mais elle ne capture pas les variables locales toute seule : il faut les lister après `use`.

ATTENTION

Le piège du `use` oublié. Ici, `$limite` n'existe pas à l'intérieur de la fonction :

```php
limite=5.0;limite = 5.0;
limite=5.0;estLeger = function (Item item): bool {
    return $item->weight < $limite;
};

echo estLeger(new Item('Torche', 1.0)) ? 'oui' : 'non', PHP_EOL;
```

```
Warning: Undefined variable $limite in leger.php on line 4
non
```

PHP se contente d'un avertissement, traite `$limite` comme `null`, et répond `non`. Votre torche de 1 kg n'est plus légère. La version correcte s'écrit `function (Item $item)use(limite): bool { ... }`.

NOTE

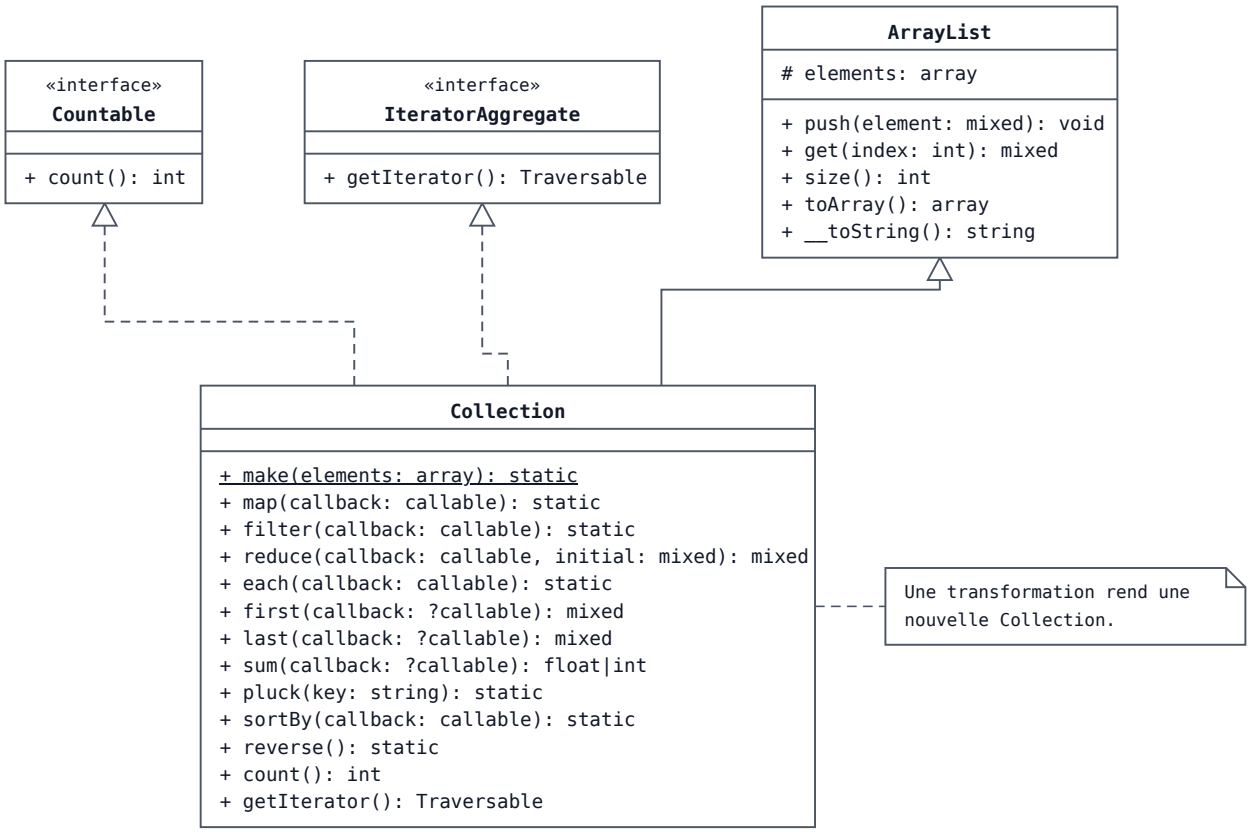
Histoire de PHP. Les fonctions fléchées `fn` sont arrivées en PHP 7.4, en 2019. Avant, il fallait écrire `function ($x) use ($y) { return …; }`, avec la liste explicite des variables emportées. Les fonctions qui reçoivent un callback sont bien plus anciennes : `array_map()` et `array_filter()` existent depuis PHP 4. La nouveauté n'est donc pas l'idée, c'est l'écriture courte.

7. Étape 3 : la Collection

Fourni : `src/Collection.php`, avec les treize signatures et un indice pour la plupart. **À écrire** : les treize corps de méthode, en commençant par `make()`.

PHP

```php
$butin = [
    new Item('Épée longue', 3.5),
    new Item('Potion de soin', 0.5),
    new Item('Bouclier', 6.0),
    new Item('Torche', 1.0),
];
```



Le trait plein au triangle vide ◁—— se lit `Collection extends ArrayList`. Vous ne réécrivez ni `push`, ni `get`, ni `size`, ni `toArray`, ni `__toString`. Tout est hérité. `Collection` n'ajoute que les transformations, plus `count()` et `getIterator()`, qui viennent des deux interfaces natives du chapitre 7.

Deux types nouveaux dans ces boîtes : `static` désigne la classe sur laquelle la méthode est appelée, ici `Collection`. `float|int` se lit « un flottant ou un entier » : la somme de quatre entiers reste un entier.

La règle de l'étape

Une transformation ne modifie jamais la collection de départ. Elle en renvoie une nouvelle. Cela vaut pour `map`, `filter`, `pluck`, `sortBy` et `reverse`. `each()` est la seule exception : elle exécute le callback pour son effet, puis rend la collection elle-même.

```php
sac=Collection::make(sac = Collection::make(sac=Collection::make(butin);
legers=legers = legers=sac→filter(fn (Item item):bool⇒item): bool ⇒ item):bool⇒item
→weight < 5.0);

echo sac->size(),PHPEOL;echosac→size(), PHP_EOL;
echo sac->size(),PHPEOL;echolegers→size(), PHP_EOL;
```

```
4
3
```

Le sac d'origine n'a pas bougé. C'est ce qui rend le chaînage sûr : chaque maillon reçoit une collection et en produit une autre, sans modifier la précédente.

`make()`, la fabrique

`Collection::make([...])` est une méthode statique : elle s'appelle sur la classe, pas sur un objet. Comme `Dice::d6()` au chapitre 1, et comme le `collect([...])` de Laravel. Écrivez-la en premier, les autres méthodes s'appuient dessus. Pour `map()` et `filter()`, remplissez un nouveau tableau, puis rendez `static::make($resultat)` : `static::` appelle `make()` sur la classe de l'objet courant, `self::` l'appellerait sur la classe où le code est écrit. Ici les deux donnent une `Collection`.

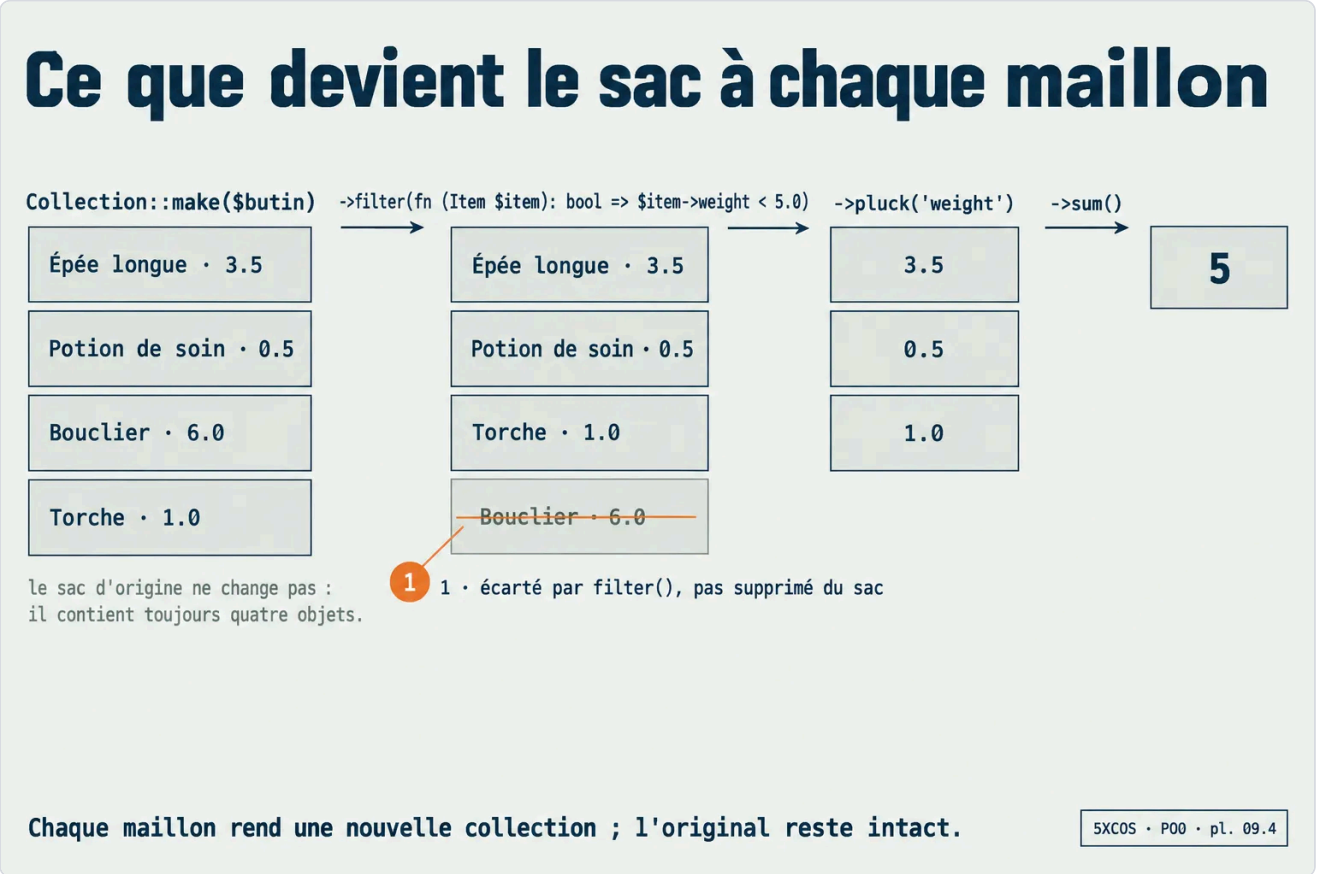
`map()`, `filter()`, `pluck()`

`map()` transforme chaque élément et rend la collection des résultats. `filter()` garde les éléments pour lesquels le callback répond `true`. `pluck('name')` lit la propriété `name` de chaque objet, ou la clé `name` de chaque tableau.

```php
noms=Collection::make(noms = Collection::make(noms=Collection::make(butin)
    →pluck('name')
    →map(fn (string nom):string⇒mbstrtoupper(nom): string ⇒
mb_strtoupper(nom):string⇒mbstrtoupper(nom));

echo noms->get(0),PHPEOL;noms→get(0), PHP_EOL;

noms->get(0),PHPEOL;poids = Collection::make(butin)->filter(fn(Itembutin)
    →filter(fn (Item butin)->filter(fn(Itemitem): bool ⇒
item->weight<5.0)->pluck('weight')->sum();echoitem→weight < 5.0)
    →pluck('weight')
    →sum();

echo item->weight<5.0)->pluck('weight')->sum();echopoids, PHP_EOL;
```

```
ÉPÉE LONGUE
5
```

Quatre lignes pour le poids des objets légers, et chacune fait une seule chose : sélectionner, extraire, additionner.



`Countable` et `IteratorAggregate`

Vous connaissez les deux depuis l'`Inventory` du chapitre 7 : `count()` et `getIterator()`, une ligne chacune, et `getIterator()` rend un `ArrayIterator` construit sur vos éléments.

```php
echo count(sac),PHPEOL;foreach(sac), PHP_EOL;

foreach (sac),PHPEOL;foreach(sac→sortBy(fn (Item item):float⇒item): float ⇒
item):float⇒item→weight) as $item) {
    echo $item→name, ' (', $item→weight, ' kg)', PHP_EOL;
}
```

```
4
Potion de soin (0.5 kg)
Torche (1 kg)
Épée longue (3.5 kg)
Bouclier (6 kg)
```

`reduce()` : une seule valeur à partir de tous les éléments

`reduce()` promène un **accumulateur**, une variable qui garde le résultat intermédiaire. Le callback le reçoit en premier argument, et l'élément en second.

```php
phrase=Collection::make(phrase = Collection::make(phrase=Collection::make(butin)
    →pluck('name')
    →reduce(fn (string acc,stringacc, string acc,stringnom): string ⇒ acc===''?acc === ''
? acc===''?nom : acc.','.acc . ', ' . acc.','.nom, '');

echo $phrase, PHP_EOL;
```

```
Épée longue, Potion de soin, Bouclier, Torche
```

Le second argument, ici la chaîne vide, est la valeur de départ de l'accumulateur. Cet exemple ajoute le séparateur avant chaque nom sauf le premier. La virgule en trop du début de chapitre ne peut plus se produire.

Ce que les tests attendent

Méthode	Comportement vérifié
`make()`	Remplit la collection depuis un tableau. Sans argument, elle est vide.
`map()`, `filter()`	Rendent une nouvelle collection, laissent l'originale intacte, et renumérotent les index à partir de `0`.
`reduce()`	Le callback reçoit l'accumulateur, puis l'élément.
`each()`	Exécute le callback sur chaque élément et rend la collection elle-même.
`first()`, `last()`	Avec ou sans callback, et `null` quand rien ne correspond.

Méthode	Comportement vérifié
`sum()`	Additionne des nombres : `10` sur `[1, 2, 3, 4]`, `0` sur une collection vide. Avec un callback, additionne ce qu'il renvoie : `11.0` sur le sac.
`pluck()`	Lit une propriété d'objet ou une clé de tableau.
`sortBy()`, `reverse()`	L'originale garde son ordre.
`Countable`, `IteratorAggregate`	`count($collection)` et `foreach ($collection as $item)` fonctionnent.
héritage	`push()`, `size()`, `indexOf()`, `includes()` et `__toString()` marchent sans être réécrits.

> **ASTUCE**
>
> Si `reduce()` résiste, écrivez d'abord `sum()`. C'est un `reduce()` dont l'accumulateur part de `0` et dont l'opération est une addition.

Vérification : `composer test -- --group=etape-3`, 24 tests verts.

8. Le coût d'un algorithme

Vous avez deux façons d'atteindre un élément : par index avec `get()`, ou en parcourant avec `indexOf()`. Elles ne coûtent pas la même chose, et la notation **Big O** le dit en un symbole. Elle compte des étapes, pas des secondes, et ne garde que ce qui domine quand `n` grandit.

```php
noms=['Arthur','Perceval','Merlin','Guenie`vre'];noms = ['Arthur', 'Perceval', 'Merlin',
'Guenièvre'];

noms=['Arthur','Perceval','Merlin','Guenie`vre'];premier =
noms[0];//O(1):unee´tape,quellequesoitlatailleforeach(noms[0];          // O(1)  : une
étape, quelle que soit la taille

foreach (noms[0];//O(1):unee´tape,quellequesoitlatailleforeach(noms as
nom)//O(n):4e´tapespour4e´le´mentsforeach(nom) {     // O(n)  : 4 étapes pour 4 éléments
}

foreach (nom)//O(n):4e´tapespour4e´le´mentsforeach(noms as $a) {       // O(n²) : 16 étapes
pour 4 éléments
    foreach ($noms as $b) {
    }
}
```

Notation	Nom	Dans votre code
O(1)	constant	`get()`, `push()`, `size()` sur `ArrayList`

Notation	Nom	Dans votre code
O(n)	linéaire	`indexOf()`, `map()`, `filter()`, `sum()`, et `remove()`, parce qu'`array_splice()` renumérote tout ce qui suit
O(n²)	quadratique	une boucle qui appelle `indexOf()` à chaque tour

Deux boucles imbriquées qui parcourent chacune les `n` éléments donnent O(n²), et le coût grimpe très vite.

9. Côte à côte avec la `Collection` de Laravel

Comparons. En haut votre classe, en bas `Illuminate\Support\Collection`.

```php
// Votre Collection
echo Collection::make(butin)->filter(fn(Itembutin)
    →filter(fn (Item butin)->filter(fn(Itemitem): bool ⇒
item->weight<5.0)->sum(fn(Itemitem→weight < 5.0)
    →sum(fn (Item item->weight<5.0)->sum(fn(Itemitem): float ⇒
item->weight),PHPEOL;//Illuminate\Support\Collectionechocollect(item→weight), PHP_EOL;

// Illuminate\Support\Collection
echo collect(item->weight),PHPEOL;//Illuminate\Support\Collectionechocollect(butin)
    →filter(fn (Item item):bool⇒item): bool ⇒ item):bool⇒item→weight < 5.0)
    →sum(fn (Item item):float⇒item): float ⇒ item):float⇒item→weight), PHP_EOL;
```

```
5
5
```

Mêmes noms de méthodes, mêmes callbacks, même règle de la nouvelle collection. `collect()` est un raccourci pour `new Collection(...)`, l'équivalent de votre `make()`. Pour lancer vraiment la seconde chaîne, l'exercice 5 installe le paquet de Laravel dans votre dépôt.

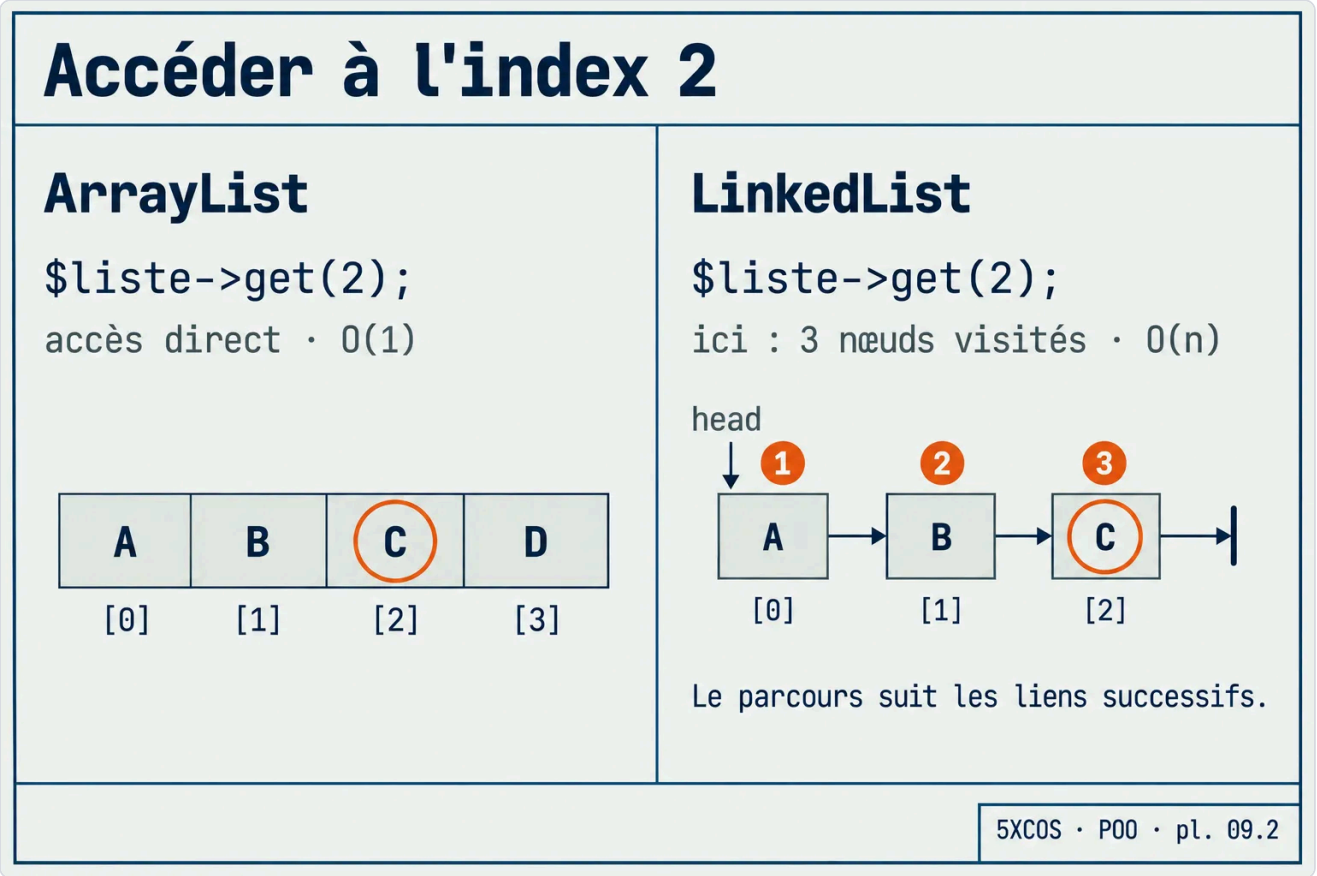
> **NOTE**
>
> Une vraie différence, qui vaut d'être connue. Le `filter()` de Laravel conserve les clés d'origine, alors que le vôtre renumérote à partir de 0. Sur le sac d'Arthur, Laravel rend les clés `0`, `1` et `3`, et votre classe rend `0`, `1` et `2`. D'où le `→values()` que vous croiserez souvent dans du code Laravel : il remet les index à plat. `sortBy()` et `reverse()` suivent la même règle.

La vraie `Collection` a plus de cent cinquante méthodes de plus, comme `groupBy` ou `chunk`. Inutile de les mémoriser : vous avez de quoi les comprendre en lisant leur description.

10. Bonus : `LinkedList`

Même contrat, autre mécanique. Les éléments ne sont plus rangés côte à côte : chaque maillon, un `Node`, garde un élément et le maillon suivant. La classe est fournie, elle tient en deux propriétés :

```php
final class Node
{
    public function __construct(
        public mixed element=null,public?Nodeelement = null,
        public ?Node element=null,public?Nodenext = null,
    ) {}
}
```



Lire le 50e élément demande donc de traverser les 49 précédents, soit O(n) au lieu de O(1). En revanche, retirer un maillon déjà atteint ne décale rien : le maillon précédent reçoit le suivant dans `next`. La méthode `remove(int $index)` reste O(n), parce qu'il faut d'abord atteindre le maillon. Les tests du bonus vérifient le même contrat que ceux de l'étape 2, en 14 tests analogues. Un paramètre typé `ListInterface` accepte une `ArrayList` comme une `LinkedList`, alors que tout diffère à l'intérieur.

À retenir

- Une **transformation renvoie une nouvelle collection**. `map`, `filter`, `sortBy` et `reverse` ne modifient jamais l'original, dans votre classe comme dans Laravel.

- Un **callback** est une fonction passée en paramètre. `fn (…) ⇒ …` utilise les variables autour d'elle. `function (…) use (…) { … }` demande de les lister.
- `Collection extends ArrayList` : vous héritez de ce qui existe déjà, vous n'ajoutez que ce qui est nouveau.
- Les **interfaces natives** ouvrent la syntaxe de PHP à vos objets. `Countable` donne `count()`, `IteratorAggregate` donne `foreach`.
- **Big O** compte les étapes, pas les secondes. Accès par index O(1), parcours O(n), boucles imbriquées $O(n^2)$.

Exercices

Sur papier

1. Sans lancer le code, écrivez la sortie de `Collection::make($butin)->reverse()->first()->name`, puis de `Collection::make(`$butin`)->sortBy(fn(`$Item`i): float ⇒ $i->weight)->last()->name`. Vérifiez ensuite avec `php`.
2. Classez en O(1), O(n) ou $O(n^2)$: `Stack::peek()`, `ArrayList::includes()`, `LinkedList::get()`, et une boucle sur une liste qui appelle `includes()` sur une autre liste à chaque tour.

Atelier

1. **Obligatoire.** Les groupes `etape-1` à `etape-3` verts, poussés, vérification verte dans l'onglet Actions.
2. **Obligatoire.** Créez `examples/ouverture.php` et réécrivez-y le code de la section 1 avec votre `Collection` : cinq lignes, une constante pour le `5`, une virgule au bon endroit. `php examples/ouverture.php` doit afficher exactement `Léger : Épée longue, Potion de soin, Torche (5 kg)`.
3. **Au choix.** Ajoutez `avg(?callable $callback = null): float`, qui rend la moyenne et `0.0` sur une collection vide. Réutilisez `sum()` et `count()`. Ajoutez deux tests dans un nouveau fichier `tests/Unit/AvgTest.php`, avec `->group('etape-3')` : le groupe affichera alors 26 tests. Laravel, lui, rend `null` pour la moyenne d'une collection vide : notez la différence.
4. **Au choix.** Ouvrez https://laravel.com/docs/collections et repérez trois méthodes dont le nom suffit à deviner le comportement. Notez ce que vous pensez qu'elles renvoient, puis vérifiez. Gardez vos notes pour la séance 6.
5. **Au choix.** Lancez `composer require --dev illuminate/collections` dans votre dépôt, puis exécutez les deux chaînes de la section 9 dans un même fichier. Comparez ensuite les clés rendues par les deux `filter()` avec `var_dump(...->toArray())`.
6. **Bonus.** `LinkedList`, groupe `bonus`.

2.10 Jouer au Donjon : la classe Game

À la maison · ~4 périodes

Depuis le chapitre 1, vous construisez un donjon pièce par pièce : un héros, des monstres, des objets, un inventaire. Tout cela vit dans votre dépôt de katas, et personne n'y a encore joué. Ce chapitre assemble les briques en un vrai jeu, dans le terminal : un nom de héros, des salles, des combats coup par coup, du butin à choisir, une victoire ou une mort.

Vous recevez le plan, les étapes et ce que l'écran doit afficher. Le code, c'est vous.

À la fin de ce chapitre, vous serez capables de :

1. **assembler** des classes existantes dans une classe qui orchestre une partie, sans y recopier leurs règles.
2. **lancer** un programme PHP depuis le terminal et lire ce que l'utilisateur tape.
3. **centraliser** la lecture des réponses du joueur dans deux méthodes, pour que le reste du jeu n'en dépende pas.
4. **concevoir** deux classes à partir d'un diagramme, `Room` et `Dungeon`, puis les relier au reste.

Le travail se fait dans votre fork de https://github.com/opmvpc/poo-katas-26, niveau 4 vert. Vous ajoutez des fichiers, vous ne touchez ni aux tests ni aux classes existantes : `composer test` reste vert du début à la fin.

1. Ce que vous allez construire

Voici une partie complète, jouée avec le code de la correction. Vos textes peuvent varier, le menu non : il ne propose que ce qui est possible, et chaque action garde son numéro.

```
═══ Le Donjon ═══
Comment s'appelle votre héros ? Arthur
Bienvenue, Arthur (30/30 PV). 3 salles vous séparent de la sortie.

--- Entrée ---
Un Gobelin (5/5 PV) vous barre le passage. Au sol : Potion de soin.
Arthur (30/30 PV) · en main : mains nues · sac : 0/20 kg, 0 objet
  1. Attaquer
  2. Ramasser un objet
  0. Quitter
Votre choix : 1
Arthur frappe le Gobelin : 4 dégâts. Gobelin (1/5 PV)
Le Gobelin frappe Arthur : 6 dégâts. Arthur (24/30 PV)
Arthur (24/30 PV) · en main : mains nues · sac : 0/20 kg, 0 objet
  1. Attaquer
  2. Ramasser un objet
  0. Quitter
Votre choix : 1
Arthur frappe le Gobelin : 5 dégâts. Gobelin (0/5 PV)
Le Gobelin tombe.
Arthur (24/30 PV) · en main : mains nues · sac : 0/20 kg, 0 objet
  2. Ramasser un objet
  6. Avancer
  0. Quitter
Votre choix : 2
Vous ramassez : Potion de soin : potion (0.5 kg, +5 PV)
Arthur (24/30 PV) · en main : mains nues · sac : 0.5/20 kg, 1 objet
  3. Boire une potion
  5. Jeter un objet
  6. Avancer
  0. Quitter
Votre choix : 6

--- Couloir ---
Personne. Au sol : Épée courte, Masse du gardien.
Arthur (24/30 PV) · en main : mains nues · sac : 0.5/20 kg, 1 objet
  2. Ramasser un objet
  3. Boire une potion
  5. Jeter un objet
  6. Avancer
  0. Quitter
Votre choix : 2
Ramasser quoi ?
  1. Épée courte : arme (2 kg, 5 dégâts)
  2. Masse du gardien : arme (18 kg, 7 dégâts)
  0. Rien
Votre choix : 1
Vous ramassez : Épée courte : arme (2 kg, 5 dégâts)
L'équiper ? (o/n) o
Vous tenez maintenant : Épée courte
Arthur (24/30 PV) · en main : Épée courte · sac : 2.5/20 kg, 2 objets
  2. Ramasser un objet
  3. Boire une potion
  5. Jeter un objet
  6. Avancer
```

```
  0. Quitter
Votre choix : 2
Le sac refuse : "Masse du gardien" ne rentre pas : le sac ne porte que 20 kg.
Arthur (24/30 PV) · en main : Épée courte · sac : 2.5/20 kg, 2 objets
  2. Ramasser un objet
  3. Boire une potion
  5. Jeter un objet
  6. Avancer
  0. Quitter
Votre choix : 5
Vous posez au sol : Potion de soin
Arthur (24/30 PV) · en main : Épée courte · sac : 2/20 kg, 1 objet
  2. Ramasser un objet
  6. Avancer
  0. Quitter
Votre choix : 2
Ramasser quoi ?
  1. Masse du gardien : arme (18 kg, 7 dégâts)
  2. Potion de soin : potion (0.5 kg, +5 PV)
  0. Rien
Votre choix : 1
Vous ramassez : Masse du gardien : arme (18 kg, 7 dégâts)
L'équiper ? (o/n) o
Vous tenez maintenant : Masse du gardien
Arthur (24/30 PV) · en main : Masse du gardien · sac : 20/20 kg, 2 objets
  2. Ramasser un objet
  4. Équiper une arme
  5. Jeter un objet
  6. Avancer
  0. Quitter
Votre choix : 5
Vous posez au sol : Épée courte
Arthur (24/30 PV) · en main : Masse du gardien · sac : 18/20 kg, 1 objet
  2. Ramasser un objet
  6. Avancer
  0. Quitter
Votre choix : 6

--- Salle du trône ---
Un Dragon (30/30 PV) vous barre le passage. Au sol : rien.
Arthur (24/30 PV) · en main : Masse du gardien · sac : 18/20 kg, 1 objet
  1. Attaquer
  0. Quitter
Votre choix : 1
Arthur frappe le Dragon : 14 dégâts. Dragon (16/30 PV)
Le Dragon frappe Arthur : 11 dégâts. Arthur (13/30 PV)
Arthur (13/30 PV) · en main : Masse du gardien · sac : 18/20 kg, 1 objet
  1. Attaquer
  0. Quitter
Votre choix : 1
Arthur frappe le Dragon : 15 dégâts. Dragon (1/30 PV)
Le Dragon frappe Arthur : 10 dégâts. Arthur (3/30 PV)
Arthur (3/30 PV) · en main : Masse du gardien · sac : 18/20 kg, 1 objet
  1. Attaquer
  0. Quitter
Votre choix : 1
```

```
Arthur frappe le Dragon : 10 dégâts. Dragon (0/30 PV)
Le Dragon tombe.
Arthur (3/30 PV) · en main : Masse du gardien · sac : 18/20 kg, 1 objet
  6. Avancer
  0. Quitter
Votre choix : 6

═══ Victoire ═══
Arthur (3/30 PV) sort du Donjon après 3 salles, avec 1 objet dans son sac :
- Masse du gardien : arme (18 kg, 7 dégâts)
```

Sept actions, toujours les mêmes numéros : 1 attaquer, 2 ramasser, 3 boire, 4 équiper, 5 jeter, 6 avancer, 0 quitter. Une action absente du menu est une action impossible dans l'état courant.

2. Ce que vous avez déjà

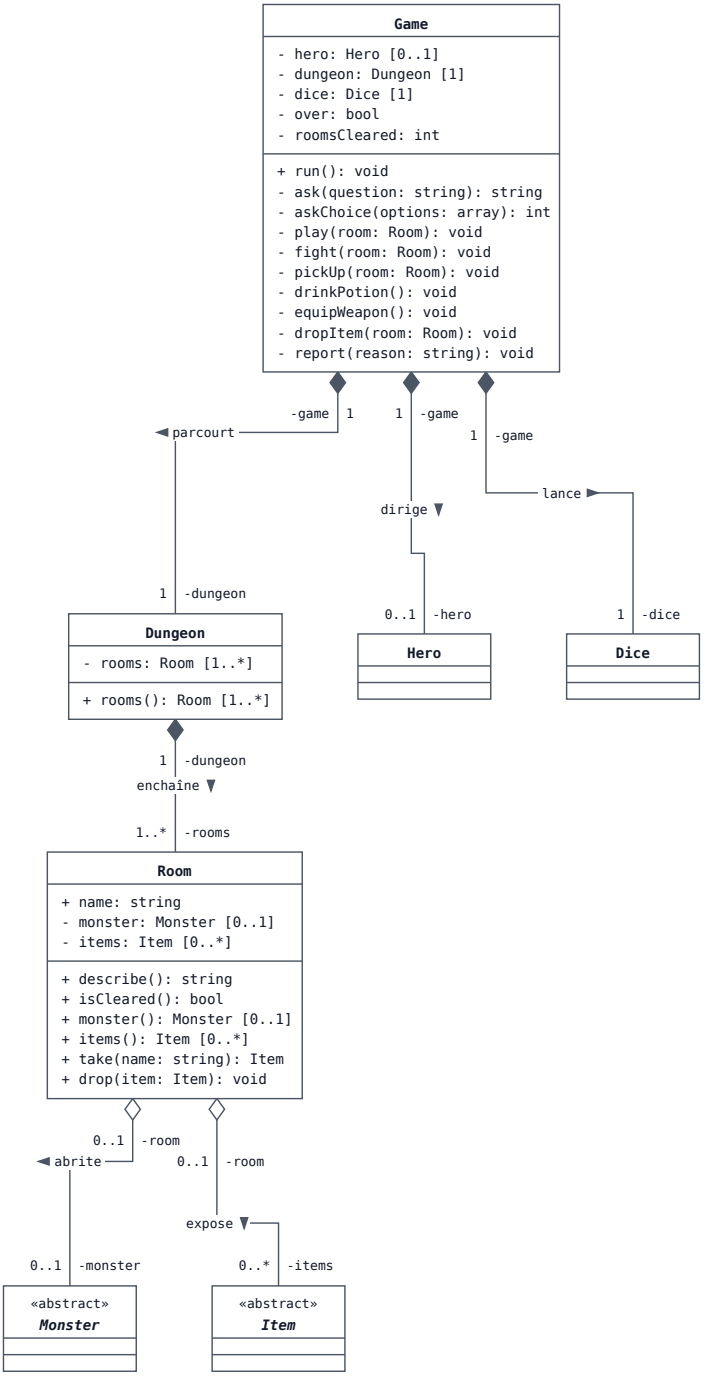
Presque tout. Relisez les signatures dans `src/` avant d'écrire une ligne : la moitié du travail consiste à appeler ce qui existe.

Brique	Ce qu'elle vous donne	Chapitre
`Hero`	`new Hero('Arthur', 30)`, `->inventory`, `->weapon`, `->equip(Weapon)`, `->drink(Potion)`, `->attack()`, `->takeDamage(int)`, `->isAlive()`, `->isFullHealth`, `echo $hero`	1, 3, 5, 6, 7
`Goblin`, `Dragon`	`new Goblin()` a 5 PV et frappe à 2, `new Dragon()` a 30 PV et frappe à 8. `echo $monster` donne `Gobelin (5/5 PV)`	6
`Weapon`, `Potion`	`new Weapon('Épée courte', 2.0, 5)`, `new Potion('Potion de soin', 0.5, 5)`, `->name`, `->weight`, `echo $item` donne `Épée courte : arme (2 kg, 5 dégâts)`	6, 7
`Inventory`	`->add(Item)`, qui lève `InventoryFullException` au-delà de `->maxWeight`, `->remove(string)`, `->totalWeight()`, `count($inventory)`, `foreach`	5, 6, 7
`Dice`	`Dice::d6()`, `->roll()`	1, 3
`Fighter`, `HasHealth`, `Rarity`	le contrat de combat, la santé, la rareté	7

Quatre pièces restent à écrire. Une **salle** avec ce qu'elle contient. Un **donjon** qui enchaîne les salles. Une classe `Game` qui mène la partie. Un **script** qui la lance. `Room` et `Dungeon` ont été dessinées au chapitre 5, jamais codées : c'est le moment.

3. Le plan

Un diagramme avant le code, comme au chapitre 4. Gardez les noms et les signatures publiques. Les méthodes privées de `Game` sont une proposition : une par action du menu, plus celles qui lisent le clavier.



Quatre décisions à lire dans ce diagramme.

- `Game` **fabrique le héros, le donjon et le dé**, donc les possède : losange plein ◆———. Le `0..1` du héros dit que la partie existe un instant sans lui, le temps de demander son nom.
- **Une salle garde un monstre et des objets sans contrôler leur durée de vie** : losange vide ◇———. Le monstre existe avant d'être placé, l'objet quitte la salle pour le sac, et peut y revenir. C'est le `drop()` et le `take()` du chapitre 5, avec plusieurs objets.
- `Room` **rend ce que** `Game` **doit lui demander, rien de plus.** `monster()` rend `null` dans une salle vide. `items()` liste le sol pour que le joueur choisisse. `take()` retire l'objet choisi, `drop()` en repose un.

- `Game` **a trois collaborateurs** : le héros, le donjon, le dé. Elle demande, elle affiche, elle enchaîne. Elle ne calcule ni points de vie, ni poids, ni dégâts d'arme : ces règles vivent dans `Hero`, `Inventory` et les monstres.

Dessinez votre version sur papier, vingt minutes, avant d'ouvrir l'éditeur.

4. Lancer PHP depuis le terminal

Jusqu'ici, vos classes tournaient dans des tests. Un jeu se lance à la main. Créez `bin/play.php` à la racine du dépôt :

```php
<?php

declare(strict_types=1);

require __DIR__ . '/../vendor/autoload.php';

(new Dungeon\Game())->run();
```

La ligne `require` charge l'autoload de Composer, comme au chapitre 2 : `Dungeon\Game` se trouve tout seul dans `src/Game.php`. La dernière ligne construit la partie et la lance.

```bash
php bin/play.php
```

```
Fatal error: Uncaught Error: Class "Dungeon\Game" not found in .../bin/play.php:7
```

C'est normal, `Game` n'existe pas encore : l'étape 1 la crée. Lancez toujours le jeu avec cette commande. Un script `composer play` ne transmet pas le clavier au programme sous Windows.

Deux outils de PHP suffisent pour dialoguer. `STDIN` est le flux dans lequel PHP reçoit ce que l'utilisateur tape. `fgets(STDIN)` attend une ligne et Entrée, puis renvoie la ligne avec son saut de ligne à la fin, d'où un `trim()` systématique. Quand l'entrée est fermée, `fgets()` renvoie `false` : traitez ce cas comme un abandon. `echo` avec `PHP_EOL` affiche. La fonction `readline()` fait presque la même chose avec une invite, mais elle n'est pas installée partout : gardez `fgets(STDIN)`.

> **ASTUCE**
>
> Écrivez deux méthodes privées dès le début. La première pose une question et rend la réponse nettoyée. La seconde affiche un menu numéroté et redemande tant que le numéro n'existe pas. Tout le jeu passe par elles : le jour où les réponses viendront d'ailleurs, il n'y aura que ces deux endroits à changer.

5. Les étapes

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/poo/jouer-au-donjon-la-classe-game

Étape 1 : le script et le nom

But : `php bin/play.php` demande un nom, construit un `Hero` et le salue.

```
=== Le Donjon ===
Comment s'appelle votre héros ? Arthur
Bienvenue, Arthur (30/30 PV).
```

Indices

- `Hero` refuse déjà un nom vide avec une `InvalidArgumentException`. Ne réécrivez pas la règle : attrapez l'exception dans `Game` et reposez la question.
- `new Hero($name, 30)` : un héros de 10 PV meurt au premier coup du dragon, qui frappe à 8 plus le dé. La difficulté, c'est `Game` qui la décide.
- `echo $hero` affiche `Arthur (30/30 PV)` grâce à `__toString()`. Servez-vous-en partout.
- `Game` garde le héros dans une propriété privée, que `run()` remplit après la lecture d'un nom valide.

Piège : écrire les six étapes avant de lancer le jeu une seule fois. Une partie complète prend une minute, lancez-la après chaque étape.

Étape 2 : les salles

But : `src/Room.php` et `src/Dungeon.php` existent, et le jeu décrit la première salle.

```
--- Entrée ---
Un Gobelin (5/5 PV) vous barre le passage. Au sol : Potion de soin.
```

Indices

- `__construct(string $name, ?Monster $monster = null, array $items = [])`. Relisez `Polygon`, au chapitre 8, pour le type `array` : à vous de vérifier que chaque élément est un `Item`, et de lever une `InvalidArgumentException` sinon.
- `describe()` rend la phrase complète, `Game` l'affiche. Une salle sans monstre dit « Personne. », une salle sans objet dit « Au sol : rien. » Les noms viennent de `$item->name`.
- `isCleared()` répond `true` quand la salle n'a pas de monstre ou que celui-ci n'est plus en vie. `Monster` sait déjà répondre à `isAlive()`.
- `Dungeon` construit ses trois salles dans son constructeur : l'Entrée avec un gobelin et la potion, le Couloir avec l'épée courte et une `Weapon('Masse du gardien', 18.0, 7)`, la Salle du trône avec le dragon. `rooms()` les rend dans l'ordre, et `Game` les parcourt avec `foreach`.

Piège : donner à `Room` une méthode qui rend les points de vie du monstre pour que `Game` les regarde. `Game` n'a besoin du monstre que pour le combat. `isCleared()` répond au reste.

Étape 3 : la boucle et le menu

But : le jeu enchaîne les salles, affiche une ligne d'état et un menu à chaque tour, et exécute le choix.

```
Arthur (30/30 PV) · en main : mains nues · sac : 0/20 kg, 0 objet
  1. Attaquer
  2. Ramasser un objet
  0. Quitter
Votre choix : 9
Ce choix n'existe pas.
Votre choix :
```

Indices

- Deux boucles emboîtées : `foreach` sur les salles, `while` sur les tours d'une salle. La boucle intérieure s'arrête quand le joueur avance ou quitte, ou quand le héros meurt.
- La ligne d'état vient entièrement d'objets qui existent : `echo $hero`, `$hero→weapon?→name`, `$hero →inventory→totalWeight()`, `count($hero→inventory)`. `Game` met les mots autour.
- Le menu se construit à partir de l'état, avant l'affichage. « Attaquer » si un monstre est vivant, « Ramasser » s'il reste un objet au sol, « Boire » s'il y a une potion dans le sac, « Équiper » s'il y a dans le sac une arme qu'on ne tient pas, « Jeter » s'il y a autre chose que l'arme en main, « Avancer » quand la salle est franchie. Chaque action garde son numéro.
- Un `match` sur le numéro choisi, comme le `match` de `Rarity::multiplier()`. Chaque cas appelle une méthode privée de `Game`. Le `match` reste court, les méthodes portent le détail.
- `fgets(STDIN)` rend une chaîne. `ctype_digit($answer)` vaut `true` si elle ne contient que des chiffres. Alors seulement, `(int) $answer` la transforme en nombre : `(int) 'quatre'` vaudrait `0`.

Piège : `exit()` ou `die()` pour quitter. Ces deux mots arrêtent le script sur-le-champ, sans rien exécuter ensuite : le bilan de l'étape 6 ne s'affichera jamais. La boucle doit se terminer proprement.

Étape 4 : le combat, un tour à la fois

But : « Attaquer » joue un tour. Le héros frappe, et si le monstre tient encore debout, il répond. Chaque coup dit ses dégâts et l'état de celui qui les reçoit.

```
Votre choix : 1
Arthur frappe le Gobelin : 4 dégâts. Gobelin (1/5 PV)
Le Gobelin frappe Arthur : 6 dégâts. Arthur (24/30 PV)
Votre choix : 1
Arthur frappe le Gobelin : 5 dégâts. Gobelin (0/5 PV)
Le Gobelin tombe.
```

Les dégâts dépendent du dé : vos chiffres seront différents. Le texte, lui, ne change pas.

Indices

- La règle d'un coup est celle de `Battle` : `attack()` plus un lancer de dé, encaissé par `takeDamage()`. Les trois méthodes du contrat `Fighter` suffisent, dans un sens puis dans l'autre. `Game` affiche le montant et l'objet touché, avec `echo $monster` et `echo $hero`.
- Pas d'`echo` dans `Hero::attack()` ni dans `takeDamage()`. Ces méthodes sont testées, et un test n'affiche rien. C'est `Game` qui raconte, avec les valeurs que les objets lui rendent.
- Le dé se crée une fois, dans le constructeur de `Game`, et sert à tous les coups. `Dice::d6()` rend le jeu jouable. `Dice::d20()` rend chaque coup imprévisible.
- Si le héros tombe, la partie est finie : sortez des deux boucles. La propriété `$over` du diagramme est faite pour cela. Une exception à vous, `HeroDiedException`, attrapée dans `run()`, est l'autre façon. Choisissez, et dites pourquoi dans le `README`.

Piège : afficher les points de vie du monstre en les recalculant dans `Game`. `echo $monster` le fait déjà, et il dit `Gobelin (0/5 PV)`.

Étape 5 : le butin

But : ramasser, équiper, jeter, boire.

```
Votre choix : 2
Ramasser quoi ?
  1. Épée courte : arme (2 kg, 5 dégâts)
  2. Masse du gardien : arme (18 kg, 7 dégâts)
  0. Rien
Votre choix : 1
Vous ramassez : Épée courte : arme (2 kg, 5 dégâts)
L'équiper ? (o/n) o
Vous tenez maintenant : Épée courte
Votre choix : 2
Le sac refuse : "Masse du gardien" ne rentre pas : le sac ne porte que 20 kg.
Votre choix : 5
Vous posez au sol : Potion de soin
```

Indices

- Quand plusieurs objets sont possibles, un sous-menu les liste avec `echo $item`, plus « 0. Rien ». Quand un seul est possible, pas de question. Écrivez ce sous-menu une fois, il sert aux quatre actions du butin.
- Ramasser, c'est retirer l'objet de la salle puis l'ajouter au sac : `Room::take()` d'un côté, `$this->hero->inventory->add()` de l'autre. Si le sac refuse, `InventoryFullException` remonte : attrapez-la dans `Game`, affichez son message, et rendez l'objet à la salle avec `drop()`.
- Un `Weapon` ramassé peut être tenu tout de suite avec `equip()`. « Équiper une arme » fait la même chose avec une arme déjà dans le sac. `Game` utilise `instanceof Weapon` pour décider ce qu'elle propose. Le calcul des dégâts reste dans `Hero`.
- Jeter, c'est le chemin inverse de ramasser : `remove()` du sac, `drop()` dans la salle. L'arme en main ne se jette pas : `Hero` n'a pas de méthode pour la lâcher, et c'est très bien ainsi.

- `Hero::drink()` soigne, puis retire du sac la première potion de ce nom. Si elle n'y est pas, le héros se soigne quand même : à `Game` de ne proposer que des potions du sac, trouvées par un `foreach` sur l'inventaire et un `instanceof Potion`.
- Boire à pleine santé ne rend rien et fait quand même disparaître la potion. Un message évite l'erreur : `isFullHealth` est déjà là.

Piège : un nom en dur, `'Potion de soin'`, dans `Game`. Le jour où le donjon contient une autre potion, le jeu ne la verra pas. Cherchez par type, pas par nom.

Étape 6 : la fin

But : la partie se termine toujours par un bilan.

```
=== Victoire ===
Arthur (3/30 PV) sort du Donjon après 3 salles, avec 1 objet dans son sac :
- Masse du gardien : arme (18 kg, 7 dégâts)
```

Indices

- Victoire quand la boucle des salles se termine avec un héros en vie. Défaite quand le héros tombe. Abandon quand le joueur choisit `0` ou quand `fgets()` rend `false`. Trois fins, une seule méthode de bilan, avec la raison en paramètre.
- Le sac s'affiche avec `foreach` et `echo '- ', $item`, comme l'inventaire du chapitre 7. `count()` donne le nombre d'objets.
- Comptez les salles terminées dans `Game`, au moment d'avancer : c'est la seule information que personne d'autre ne connaît. Le nombre de salles de l'accueil, lui, vient de `count($this->dungeon->rooms())`.

6. Les règles du jeu, pour vous

Le résultat compte, la façon aussi. La faute la plus fréquente, et sa correction :

```php
// Game recalcule ce que Hero sait déjà : à supprimer.
hp=hp = hp=this->hero->hp - damage;echo"Arthur(damage;
echo "Arthur (damage;echo"Arthur(hp/{this->hero->maxHp} PV)", PHP_EOL;

// Game demande, Hero répond.
this->hero->takeDamage(damage);echodamage);
echo damage);echothis->hero, PHP_EOL;
```

Relisez votre `Game` avec ces cinq questions.

1. `Game` **recalcule-t-elle des points de vie, un poids ou des dégâts d'arme ?** Supprimez le calcul et appelez la méthode de `Hero`, `Inventory` ou `Weapon`.
2. **Chaque exception est-elle attrapée là où vous pouvez y répondre ?** Un nom vide se redemande, un sac plein s'annonce. Aucune trace PHP ne doit atteindre le joueur.

3. **Le menu ment-il ?** Proposer « Attaquer » dans une salle vide ou « Boire » sans potion, c'est un bug d'affichage. Les options se construisent à partir de l'état.
4. **Le programme se termine-t-il toujours par le bilan ?** Pas d'`exit()`, pas de `die()`. Les boucles s'arrêtent, `run()` se termine, le script aussi.
5. `composer test` **est-il toujours vert ?** Vous avez ajouté trois classes et un script, rien cassé. Poussez, et regardez l'onglet Actions.

7. Pour aller plus loin

À faire seulement quand une partie complète se joue de bout en bout.

- **Le combat automatique.** Si vous avez fait le niveau 5, `Battle::fight()` joue tout un combat d'un coup et rend le vainqueur. Ajoutez une action « Combattre jusqu'au bout » qui l'appelle, à côté du tour par tour.
- **Un donjon différent à chaque partie.** Un `Dice` pour choisir le monstre de chaque salle, `null` compris, un autre pour le butin.
- **La rareté qui compte.** Une `Weapon` légendaire frappe plus fort : `$this->rarity->multiplier()` existe déjà. Placez le calcul dans `Weapon`, appelé par `Hero::attack()`. Le chapitre 3 dit pourquoi ce n'est pas le travail de `Game`.
- **Sauvegarder et reprendre.** `JsonSerializable` sur le héros et l'inventaire, un fichier `save.json`, un menu « Reprendre ou nouvelle partie ». L'exercice 2 du chapitre 7 est la moitié du chemin. Reconstruire les objets depuis le fichier est l'autre moitié, la plus longue.
- **Tester `Room` et `Dungeon`.** Ajoutez `tests/RoomTest.php` : une salle sans monstre est franchie d'office, `take()` retire bien l'objet et `drop()` le remet, un élément qui n'est pas un `Item` est refusé. Trois tests, dans le style de ceux du chapitre 2.

8. Ce que vous montrez

1. Votre fork poussé, CI verte, avec `src/Game.php`, `src/Room.php`, `src/Dungeon.php` et `bin/play.php`.
2. Votre diagramme, sur papier ou en photo dans le dépôt, cohérent avec les trois classes rendues.
3. Une section « Jouer » dans le `README` du fork : la commande, les règles, deux ou trois décisions de conception en une phrase chacune.
4. Une partie jouée devant un binôme au début de la séance suivante. Il tape, vous regardez.

Ce qui est regardé	Ce qui fait dire que c'est acquis
Le lancement	`php bin/play.php` demande un nom et affiche la première salle
Le modèle	`Room` et `Dungeon` existent, `Game` ne recalcule rien que le Donjon sait déjà faire
La partie	du nom du héros jusqu'au bilan, coup par coup, sans trace d'erreur, avec un menu qui ne ment jamais
Les tests existants	`composer test` vert, CI verte

À retenir

- **Une classe qui orchestre ne recalcule rien.** `Game` appelle `Hero`, `Inventory` et les monstres. Les règles restent chez elles, et `Game` raconte ce qu'elles lui rendent.
- **Deux méthodes privées lisent le terminal**, et le reste du jeu reçoit des réponses déjà nettoyées.
- **Un menu se construit à partir de l'état**, il ne se tape pas en dur.
- **Une exception se traite là où vous pouvez y répondre**, et jamais avec une trace PHP devant le joueur.
- **Dessinez le diagramme d'abord**, vingt minutes sur papier, puis codez dans l'ordre des étapes.

Exercices

Sur papier

1. `Room::monster()` rend `null` dans une salle vide. Comment `Game` s'en sort-elle sans `if ($monster !== null)` partout ?
2. `Game` possède le héros, une salle n'en possède pas le monstre. Retrouvez les deux losanges dans votre diagramme et dites, en une phrase chacun, pourquoi.
3. Votre `match` sur le choix du joueur tient-il en moins de dix lignes ? Sinon, qu'est-ce qui devrait partir dans une méthode ?
4. Si demain le jeu se joue dans une page web au lieu du terminal, quelles méthodes de `Game` changent ? Si la réponse est « toutes », relisez la section 4.

2.11 Lire du code Laravel

Séance 5 · ~1 période

Vous avez écrit des classes, des traits et des enums dans un donjon. Voici six extraits d'un projet Laravel 12. Aucune notion nouvelle : ce sont les mots des chapitres 1 à 9, dans du code que vous n'avez pas écrit.

À la fin de ce chapitre, vous serez capables de :

1. **nommer** le concept POO derrière chaque ligne d'un fichier Laravel ;
2. **repérer** l'héritage, les traits, les enums, les interfaces et les relations dans un fichier inconnu ;
3. **distinguer** ce que Laravel fabrique pour vous de ce que vous écrivez ;
4. **retrouver** le chapitre du cours qui explique une ligne que vous ne comprenez pas.

Les quatre premiers extraits sont des fichiers écrits dans un projet. Les deux derniers viennent du code du framework lui-même. Laravel n'écrit pas `declare(strict_types=1)` en tête de ses fichiers : c'est notre convention à nous, pas la sienne.

1. Un modèle Eloquent

Eloquent est la partie de Laravel qui relie une classe PHP à une table de la base de données. Voici un modèle et l'enum qui l'accompagne.

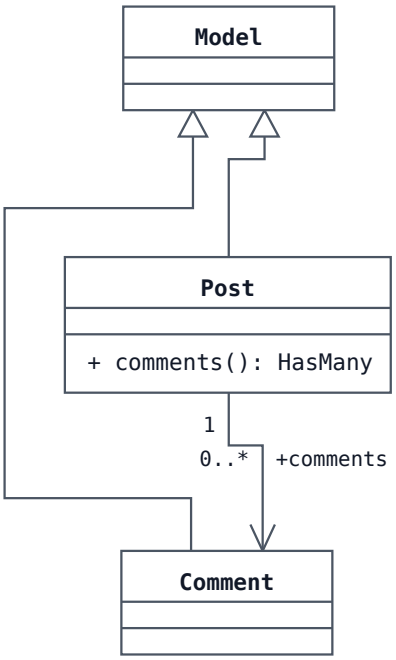
```php
<?php

namespace App\Enums;

enum PostStatus: string
{
    case Draft = 'draft';
    case Published = 'published';
}
```

```php
<?php

namespace App\Models;

use App\Enums\PostStatus;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Post extends Model
{
    use HasFactory;

    protected $fillable = ['title', 'body', 'status'];

    protected function casts(): array
    {
        return [
            'status' => PostStatus::class,
        ];
    }

    public function comments(): HasMany
    {
        return $this->hasMany(Comment::class);
    }
}
```

La ligne	Le mot du cours	Chapitre
`namespace App\Models;` puis `use ...\Model;`	Namespace et import. Le chemin du fichier suit le namespace (PSR-4)	2
`enum PostStatus: string`	Un enum dont chaque cas porte une valeur, ici une chaîne	6

La ligne	Le mot du cours	Chapitre
`class Post extends Model`	`extends` : héritage. `Post` reçoit les méthodes de `Model` sans en écrire une seule	5
`use HasFactory;`	`use` à l'intérieur d'une classe : un trait. Ses méthodes sont ajoutées à `Post`	6
`protected $fillable`	`protected` : lisible dans la classe et dans ses classes filles, pas depuis l'extérieur	1, 5
`protected function casts(): array`	Redéfinition. `Model` a déjà une méthode `casts()` ; `Post` donne la sienne	5
`'status' ⇒ PostStatus::class`	`::class` donne le nom complet d'une classe, sous forme de chaîne	2, 6
`public function comments(): HasMany`	Type de retour, et une association vers `Comment`	3, 4
`return $this→hasMany(...)`	`$this` désigne cet article-ci. `hasMany()` vient de `Model`	1, 5

Le même modèle en diagramme de classes :



`$fillable` est une liste blanche. `Post::create()` n'écrit que les colonnes qui y figurent, pour qu'un formulaire trafiqué ne puisse pas modifier n'importe quelle colonne.

> **NOTE**
>
> `protected $fillable` n'a pas de type, contrairement à toutes les propriétés écrites depuis le chapitre 1. Cette propriété vient d'une version de Laravel antérieure aux propriétés typées de PHP 7.4. Lui ajouter un type aujourd'hui casserait des milliers d'applications.

À l'usage :

```php
post=Post::create(['title'⇒'Bonjour','body'⇒'…','status'⇒'draft']);if(post =
Post::create(['title' ⇒ 'Bonjour', 'body' ⇒ '…', 'status' ⇒ 'draft']);

if (post=Post::create(['title'⇒'Bonjour','body'⇒'…','status'⇒'draft']);if(post→status
=== PostStatus::Draft) {
    foreach (post->commentsaspost→comments as post->commentsascomment) {
        echo $comment→body;
    }
}
```

`$post→status` ne contient pas la chaîne `'draft'` mais l'objet `PostStatus::Draft`. C'est le travail de `casts()`. Le même mécanisme, sans Laravel :

```php
<?php

declare(strict_types=1);

enum PostStatus: string
{
    case Draft = 'draft';
    case Published = 'published';
}

status=PostStatus::from('draft');echostatus = PostStatus::from('draft');

echo status=PostStatus::from('draft');echostatus === PostStatus::Draft ? 'oui' : 'non',
"\n";
echo $status→value, "\n";
```

```
oui
draft
```

> **ATTENTION**
>
> `Post::create()` s'écrit avec `::`, mais aucune méthode `create()` n'est déclarée dans `Model`. Eloquent redirige l'appel vers son constructeur de requêtes. Retenez la règle d'écriture, pas le mécanisme : `::` s'adresse à la classe, `→` à un objet.

Une chose à retenir : un modèle Eloquent est une classe fille de `Model` qui utilise un trait.

2. Un contrôleur

Un contrôleur reçoit une requête HTTP et renvoie une réponse.

```php
<?php

namespace App\Http\Controllers;

use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PostController extends Controller
{
    public function store(Request $request): RedirectResponse
    {
        $validated = $request->validate([
            'title' => ['required', 'string', 'max:255'],
            'body' => ['required', 'string'],
        ]);

        $post = Post::create($validated);

        return redirect()->route('posts.show', $post);
    }
}
```

La classe mère est dans votre projet, et elle est abstraite :

```php
<?php

namespace App\Http\Controllers;

abstract class Controller
{
    //
}
```

La ligne	Le mot du cours	Chapitre
`class PostController extends Controller`	`extends` encore. La classe mère est abstraite : on ne fait jamais `new Controller`	5
`store(Request $request)`	Paramètre typé par une classe. Vous n'écrivez jamais `new Request` : Laravel fabrique l'objet et vous le donne	4
`: RedirectResponse`	Type de retour : la méthode promet un objet de ce type	1

La ligne	Le mot du cours	Chapitre
`$request->validate([...])`	Appel de méthode sur un objet reçu. Si la validation échoue, la méthode lève une exception et Laravel l'attrape	5
`$validated`	Un tableau, pas un objet. Tout n'est pas un objet, même en POO	1
`redirect()->route(...)`	Une fonction qui renvoie un objet, sur lequel on appelle aussitôt une méthode	1

Une chose à retenir : un paramètre typé par une classe, c'est Laravel qui fabrique l'objet et vous le donne.

3. Une migration

Une migration décrit une table de la base de données en PHP.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('body');
            $table->string('status')->default('draft');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('posts');
    }
};
```

La ligne	Le mot du cours	Chapitre
`return new class extends Migration`	Une classe anonyme : elle hérite de `Migration` et n'a pas de nom, parce qu'on ne l'appellera nulle part ailleurs	5

La ligne	Le mot du cours	Chapitre
`Schema::create(...)`	Appel avec `::` : on s'adresse à la classe `Schema`	1
`function (Blueprint $table)`	Une fonction anonyme donnée en argument. Laravel l'appelle en lui passant un `Blueprint`	8
`$table->string('status')->default('draft')`	Chaînage : la première méthode renvoie un objet, la seconde s'appelle dessus	8
`up(): void` et `down(): void`	Deux méthodes publiques : appliquer la migration, l'annuler	1

Une chose à retenir : une migration est une classe fille de `Migration`, et la base de données s'y décrit avec des objets.

4. Une route

Une route relie une adresse à une méthode d'un contrôleur.

```php
<?php

use App\Http\Controllers\PostController;
use Illuminate\Support\Facades\Route;

Route::get('/posts/{post}', [PostController::class, 'show'])->name('posts.show');
Route::post('/posts', [PostController::class, 'store'])->name('posts.store');
```

La ligne	Le mot du cours	Chapitre
`use App\Http\Controllers\PostController;`	Import, en dehors de toute classe. C'est le `use` du chapitre 2, pas celui d'un trait	2
`Route::get(...)`	Appel avec `::` : on s'adresse à la classe `Route`	1
`PostController::class`	Le nom complet de la classe, sous forme de chaîne. Aucun objet n'est créé ici	2, 6
`[PostController::class, 'show']`	Un tableau de deux chaînes : la classe et la méthode. Laravel fabriquera l'objet à l'arrivée de la requête	4
`->name('posts.show')`	Chaînage : `get()` renvoie un objet, on lui donne son nom	8

Une chose à retenir : un fichier de routes ne crée aucun objet, il dit quelle méthode de quelle classe répondra à quelle adresse.

5. Une `Collection`

`Post::all()` ne renvoie pas un tableau mais un objet `Illuminate\Support\Collection`. Voici sa déclaration et celle de l'interface principale qu'elle réalise, recopiées du framework.

```php
class Collection implements ArrayAccess, CanBeEscapedWhenCastToString, Enumerable
```

```php
interface Enumerable extends Arrayable, Countable, IteratorAggregate, Jsonable,
JsonSerializable
```

La ligne	Le mot du cours	Chapitre
`implements`	Réalisation d'une interface : la classe s'engage à fournir les méthodes du contrat	6
`Countable`	C'est pour cela que `count($posts)` fonctionne	6, 8
`IteratorAggregate`	C'est pour cela que `foreach ($posts as $post)` fonctionne	6, 8
`JsonSerializable`	C'est pour cela que `json_encode($posts)` renvoie ce qu'on attend	6
`interface Enumerable extends ...`	Une interface peut en étendre d'autres. Une seule ligne en apporte cinq	6

C'est votre collection du chapitre 9, en plus grand :

```php
titles=Post::all()->filter(fn(Posttitles = Post::all()
    →filter(fn (Post titles=Post::all()->filter(fn(Postpost) ⇒
post->status===PostStatus::Published)->map(fn(Postpost→status === PostStatus::Published)
    →map(fn (Post post->status===PostStatus::Published)->map(fn(Postpost) ⇒ $post→title);
```

Une chose à retenir : trois interfaces natives du chapitre 7 expliquent à elles seules pourquoi une `Collection` se compte, se parcourt et se convertit en JSON.

6. Cinq lignes du framework

Le début de `Illuminate\Auth\SessionGuard`.

```php
class SessionGuard implements StatefulGuard, SupportsBasicAuth
{
    use GuardHelpers, Macroable;

    public readonly string name;protectedname;

    protected name;protectedlastAttempted;
```

La ligne	Le mot du cours	Chapitre
`implements StatefulGuard, SupportsBasicAuth`	Deux interfaces réalisées par une même classe	6

La ligne	Le mot du cours	Chapitre
`use GuardHelpers, Macroable;`	Deux traits utilisés dans une même classe	6
`public readonly string $name;`	`readonly` : la valeur est écrite une fois, dans le constructeur, et plus jamais ensuite	1
`protected $lastAttempted;`	Encore une propriété sans type, pour la même raison qu'au premier extrait	1, 5

Une chose à retenir : cinq lignes du framework contiennent quatre mots du cours et aucun mot nouveau.

7. La carte du bloc

Le mot du cours	Chapitre	Où vous le voyez dans Laravel
Classe, objet, `$this`	1	Chaque `Post`, `Request`, `Collection` manipulé
`protected`	1, 5	`$fillable`, `$hidden`, `casts()`
`::` sur une classe	1	`Post::all()`, `Route::get()`, `Schema::create()`
`readonly`	1	`SessionGuard::$name`
Namespace, `use`, PSR-4	2	Les premières lignes de tout fichier
Association et multiplicités	3	`hasMany()`, `belongsTo()`, `belongsToMany()`
Objet reçu en paramètre	4	`store(Request $request)`, `function (Blueprint $table)`
`extends`, redéfinition	5	`extends Model`, `extends Controller`, `extends Migration`
Classe abstraite	5	`App\Http\Controllers\Controller`
Exceptions	5	`validate()` qui échoue, et la page d'erreur qui suit
`implements`, interface	6	`Collection`, `SessionGuard`
Trait	6	`HasFactory`, `Notifiable`, `Macroable`
Enum	6	`casts()` avec `PostStatus::class`
Fonction anonyme, chaînage	8	`function (Blueprint $table)`, `→filter()→map()`

À retenir

- `::` s'adresse à la classe, `→` à un objet.
- Un fichier de Laravel commence presque toujours par des `use`, puis un `extends`.
- Un paramètre typé par une classe : Laravel fabrique l'objet et vous le donne.
- Une relation Eloquent est une association UML, avec ses multiplicités.
- Une ligne que vous ne savez pas nommer a un chapitre : la carte ci-dessus dit lequel.

Exercices

1. Ouvrez `vendor/laravel/framework/src/Illuminate/Http/Request.php`. Notez le `extends`, chaque trait et chaque interface, avec le chapitre correspondant. Comptez 15 minutes.
2. En une phrase et avec le vocabulaire du chapitre 7 : pourquoi `foreach ($post->comments as $comment)` fonctionne-t-il ?
3. Reprenez le diagramme de la section 1 et ajoutez-y `PostController`, avec une dépendance vers `Post`. N'ajoutez aucune relation qui n'apparaît pas dans les extraits de ce chapitre.

CHAPITRE 3

Framework

Bloc Laravel · séance 1 · ~5 min

Le dernier bloc dure trois semaines, à raison de trois séances de cinq périodes. Laravel est un **framework web**, un programme qui reçoit la demande du navigateur et exécute le code associé à l'adresse demandée.

Vous l'utilisez pour créer plusieurs pages, enregistrer des données, gérer des utilisateurs et mettre une application en ligne. Le cours travaille avec Laravel 13, sorti le 17 mars 2026, et PHP 8.4. Tous les exemples et toutes les commandes utilisent ces versions.

Dans le bloc POO, vous avez lu un **contrôleur**, une classe qui traite une demande du navigateur, et un **modèle**, une classe qui représente les données. Vous y avez utilisé `extends` pour **l'héritage**, `use` pour **l'import d'une classe** et `Post::latest()` pour lire des articles. Vous reprenez maintenant ces éléments dans un projet que vous créez et complétez vous-même.

Ce que contient cette séance

Pendant cette première séance, vous créez `premier-test-laravel`, puis une page qui affiche la liste de ses articles. Une **route** associe une adresse au contrôleur qui doit la traiter. Pour cette page, le contrôleur transmet les données à une **vue**, le fichier qui produit le HTML. Les six chapitres relient ces actions aux principaux fichiers du projet. Vous pourrez ensuite ajouter de nouvelles pages en sachant où placer leur code.

1. **Framework et librairie** : une **bibliothèque**, aussi appelée librairie, fournit des fonctions que votre programme utilise directement. Laravel exécute au contraire la méthode du contrôleur quand le navigateur demande une page.
2. **Laravel** : l'origine du framework, les personnes qui le maintiennent, la publication des versions et la durée des correctifs de sécurité.
3. **Environnement de développement** : l'installation de PHP et des programmes nécessaires sur votre machine. **Composer** installe les **paquets PHP**, des ensembles de fichiers distribués sous un même nom.
4. **Installation** : les commandes qui créent un projet Laravel 13 et permettent d'ouvrir sa page d'accueil dans le navigateur.
5. **MVC** : l'organisation **modèle-vue-contrôleur** répartit la lecture des données, le traitement de la demande et la production du HTML entre trois types de fichiers.
6. **Structure d'un projet Laravel** : les principaux dossiers d'un projet neuf, les fichiers que vous modifiez et ceux que les outils produisent.

La suite du bloc

Après cette séance, vous étudiez séparément les routes, les contrôleurs, les vues et les modèles. Vous les mettez ensuite en forme avec Tailwind CSS, puis vous écrivez des formulaires qui créent, modifient et suppriment des données, dans deux projets guidés : un refuge pour animaux et un blog. Vous reprenez enfin un projet existant, traduisez son interface et le mettez en ligne. Le projet final, « Meme Battle Arena », réunit ces compétences.

La plupart des exemples reprennent la même application de blog. La classe `Post` représente les articles et la classe `Comment` représente les commentaires. Quand le navigateur demande une page liée aux articles, `PostController` traite cette demande.

3.1 Framework et librairie

Bloc Laravel · séance 1 · ~30 min

Slides du chapitre (PDF) PDF

Quand vous créez une application web, vous devez souvent reconnaître l'adresse demandée, lire des données et produire du HTML. Vous devez aussi empêcher le navigateur d'exécuter le texte saisi par un utilisateur. Cette protection s'appelle l'**échappement HTML**.

Une **bibliothèque** fournit des fonctions que votre programme utilise à des endroits précis. Un **framework** contient le programme de démarrage de l'application. Quand le navigateur demande une page, ce programme lit l'adresse et exécute la méthode enregistrée pour cette adresse. Dans chacun des quatre programmes étudiés, vous pouvez repérer le fichier qui contient l'instruction d'appel.

À la fin de ce chapitre, vous serez capables de :

- **distinguer** une bibliothèque d'un framework en répondant à la question « qui appelle qui ? »
- **expliquer** comment Laravel lance une méthode de votre code
- **citer** ce qu'un framework vous fait gagner et ce qu'il vous fait perdre
- **situer** Laravel parmi les principaux frameworks web de 2026

1. Le problème : une page de blog à la main

Cette petite application de blog tient dans un seul fichier, `index.php`. Elle affiche la liste des articles à l'adresse `/posts`. Pour toute autre adresse, elle renvoie une erreur 404.

```php
<?php

// 1. La base de données, ici en mémoire
pdo=newPDO(’sqlite::memory:’);pdo = new PDO('sqlite::memory:');
pdo=newPDO(’sqlite::memory:’);pdo→exec('CREATE TABLE posts (id INTEGER PRIMARY KEY, title
TEXT)');
pdo->exec("INSERTINTOposts(title)VALUES(’Bienvenuesurleblog’),(’<script>alert(1)
</script>’)");//2.Leroutage:quelleURLae´te´demande´e?pdo→exec("INSERT INTO posts (title)
VALUES ('Bienvenue sur le blog'), ('<script>alert(1)</script>')");

// 2. Le routage : quelle URL a été demandée ?
pdo->exec("INSERTINTOposts(title)VALUES(’Bienvenuesurleblog’),(’<script>alert(1)
</script>’)");//2.Leroutage:quelleURLae´te´demande´e?path =
parse_url(SERVER[’REQUESTURI’],PHPURLPATH);if(_SERVER['REQUEST_URI'], PHP_URL_PATH);

if (SERVER[’REQUESTURI’],PHPURLPATH);if(path === '/posts') {
    posts=posts = posts=pdo→query('SELECT title FROM posts ORDER BY id DESC')→fetchAll();
    echo '<h1>Articles</h1>', PHP_EOL;
    foreach (postsasposts as postsaspost) {
        // 3. L'échappement HTML
        echo '<p>', htmlspecialchars($post['title']), '</p>', PHP_EOL;
    }
} else {
    http_response_code(404);
    echo 'Page introuvable', PHP_EOL;
}
```

Lancez-la avec le serveur intégré de PHP, puis ouvrez `http://127.0.0.1:8080/posts` dans le navigateur.

```bash
php -S 127.0.0.1:8080 index.php
```

Le navigateur affiche un titre et deux paragraphes. Il les obtient en envoyant une **requête**, c'est-à-dire une demande adressée au serveur. Cette demande contient une **méthode HTTP**, un mot qui indique l'action voulue, comme `GET`. Elle contient aussi un chemin, comme `/posts`. Le serveur renvoie une **réponse** composée d'un code, comme `200` ou `404`, et d'un contenu.

Pour lire le HTML reçu par le navigateur, ouvrez un second terminal et utilisez `curl`, un programme qui envoie une requête. Il affiche la réponse sans la transformer en page. L'option `-i` ajoute les **en-têtes**, des lignes qui décrivent la réponse. Elles contiennent notamment le code `404 Not Found`. Sous PowerShell, tapez `curl.exe` :

```
 curl http://127.0.0.1:8080/posts
<h1>Articles</h1>
<p>&lt;script&gt;alert(1)&lt;/script&gt;</p>
<p>Bienvenue sur le blog</p>
 curl -i http://127.0.0.1:8080/contact
HTTP/1.1 404 Not Found
Host: 127.0.0.1:8080
Date: Sat, 26 Sep 2026 14:18:07 GMT
Connection: close
X-Powered-By: PHP/8.4.21
Content-type: text/html; charset=UTF-8

Page introuvable
```

Le **routage** associe l'adresse demandée au code qui doit produire la réponse. Ici, la condition `if ($path === '/posts')` réalise cette association. L'échappement transforme ensuite `<script>` en `<script>`. Le navigateur affiche donc ce texte au lieu de l'exécuter comme une balise.

Ce fichier fonctionne tant que le site ne contient qu'une page. Avec vingt pages, la condition devient une suite de `elseif` qu'il faut modifier pour chaque nouvelle adresse.

Une application doit souvent conserver l'identité de la personne connectée entre deux requêtes. Une **session** enregistre cette information sur le serveur. Il faut également refuser les formulaires envoyés depuis un autre site et préparer l'envoi des courriels. Sans outil commun, vous devez remettre en place ces mécanismes dans chaque projet.

2. La bibliothèque : vous l'appelez

Une bibliothèque rassemble des classes et des fonctions écrites par d'autres personnes. Votre programme utilise seulement celles dont il a besoin. Le mot « librairie » est aussi employé, par calque de l'anglais *library*.

Afficher une date sous la forme « mercredi 23 septembre 2026 » ou « il y a trois jours » demande plusieurs opérations. La bibliothèque Carbon fournit les méthodes qui lisent, traduisent et mettent en forme cette date. Un **paquet** est un ensemble de fichiers PHP distribué sous un même nom. **Composer** est le programme qui installe les paquets demandés par un projet. Dans un dossier vide, utilisez-le pour installer Carbon :

```bash
composer require nesbot/carbon
```

```
./composer.json has been created
Running composer update nesbot/carbon
Loading composer repositories with package information
Updating dependencies
Lock file operations: 7 installs, 0 updates, 0 removals
  - Locking carbonphp/carbon-doctrine-types (3.2.1)
  - Locking nesbot/carbon (3.14.0)
…
Generating autoload files
6 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
No security vulnerability advisories found.
Using version ^3.14 for nesbot/carbon
```

Le signe `…` remplace les lignes intermédiaires de la sortie. Composer a installé sept paquets, car Carbon en utilise cinq et l'un de ces paquets en utilise un autre. Un paquet nécessaire au fonctionnement d'un programme s'appelle une **dépendance**. Composer installe Carbon ainsi que toutes les dépendances indiquées par ces paquets.

Écrivez ensuite `dates.php` à côté du dossier `vendor` :

```php
<?php

require __DIR__ . '/vendor/autoload.php';

use Carbon\Carbon;

publishedAt=Carbon::parse('2026-09-2309:00');echopublishedAt = Carbon::parse('2026-09-23
09:00');

echo publishedAt=Carbon::parse('2026-09-2309:00');echopublishedAt->locale('fr_BE')
->isoFormat('dddd D MMMM YYYY'), PHP_EOL;
echo $publishedAt->locale('fr_BE')->diffForHumans(), PHP_EOL;
```

```
$ php dates.php
mercredi 23 septembre 2026
il y a 3 jours
```

La méthode `diffForHumans()` compare la date enregistrée à la date du jour. La seconde ligne change donc selon le jour où vous lancez le script.

PHP exécute `dates.php` de haut en bas. À la ligne qui contient `Carbon::parse(...)`, il utilise la méthode fournie par Carbon et place la date obtenue dans `$publishedAt`. Les deux lignes suivantes utilisent encore Carbon pour mettre cette date en forme. Si vous retirez ces appels, aucune méthode de Carbon n'est exécutée. **Une bibliothèque fournit donc du code que votre programme utilise aux endroits choisis.**

3. Le framework : il vous appelle

Un **contrôleur** est une classe dont les méthodes préparent la réponse à une adresse. Vous avez déjà rencontré ce contrôleur dans le bloc POO :

```php
<?php

namespace App\Http\Controllers;

use App\Models\Post;
use Illuminate\View\View;

class PostController extends Controller
{
    public function index(): View
    {
        posts=Post::latest()->get();returnview('posts.index',['posts'⇒posts =
Post::latest()→get();

        return view('posts.index', ['posts' ⇒ posts=Post::latest()
->get();returnview('posts.index',['posts'⇒posts]);
    }
}
```

Ce contrôleur déclare la méthode `index()`, mais aucun fichier de votre projet ne contient `new PostController()` suivi de `→index()`. Quand le navigateur demande `/posts`, Laravel effectue ces deux opérations. Il obtient le nom de la classe et celui de la méthode dans la route enregistrée par `routes/web.php`.

Le **modèle** `Post` est une classe qui représente les articles enregistrés dans la table `posts`. Cette table contient les deux mêmes articles. Une **vue** est un fichier qui produit le HTML envoyé au navigateur. `resources/views/posts/index.blade.php` est une version courte de la vue que vous écrirez au chapitre MVC :

```xml
<h1>Articles</h1>
@foreach (postsasposts as postsaspost)
<p>{{ $post→title }}</p>
@endforeach
```

Dans cette vue, les doubles accolades `{{ }}` appliquent l'échappement HTML au titre de chaque article. Le projet s'appelle `premier-test-laravel`. Herd ou Laragon rend ce projet accessible à l'adresse `premier-test-laravel.test`. La commande `curl http://premier-test-laravel.test/posts` envoie une requête et affiche le HTML reçu :

```
$ curl http://premier-test-laravel.test/posts
<h1>Articles</h1>
<p>&lt;script&gt;alert(1)&lt;/script&gt;</p>
<p>Bienvenue sur le blog</p>
```

La réponse contient le même HTML qu'à la section 1, avec le même échappement. Votre code ne contient pourtant ni condition sur l'adresse ni création d'un objet `PDO`.

L'association entre l'adresse et la méthode se déclare dans `routes/web.php`. La première ligne importe la classe `PostController`. La seconde indique que le chemin `/posts` doit être traité par sa méthode `index()`.

```php
use App\Http\Controllers\PostController;

Route::get('/posts', [PostController::class, 'index']);
```

Artisan est l'outil en ligne de commande de Laravel, présenté au chapitre Installation. `php artisan route:list --path=posts` affiche la route enregistrée pour le chemin `/posts` :

```
$ php artisan route:list --path=posts

  GET|HEAD       posts ..................................... PostController@index

                                                    Showing [1] routes
```

Quand Laravel reçoit une requête pour `/posts`, il cherche ce chemin dans les routes enregistrées. La route contient `PostController::class` et `index`. Laravel crée donc un objet `PostController`, exécute sa méthode `index()` et transforme la valeur renvoyée en réponse pour le navigateur.

Dans `framework.php`, la fonction `run()` reproduit trois opérations de Laravel : chercher une route, créer le contrôleur indiqué et appeler sa méthode.

```php
<?php

// Le « framework » : écrit une fois, par quelqu'un d'autre
function run(array routes,stringroutes, string routes,stringpath): void
{
    if (!isset(routes[routes[routes[path])) {
        echo 'Page introuvable', PHP_EOL;
        return;
    }

    [class,class, class,method] = routes[routes[routes[path];
    controller=newcontroller = new controller=newclass();
    echo controller->controller→controller->method(), PHP_EOL;
}
```

La fonction `run()` reçoit la liste des routes et le chemin demandé. Si ce chemin ne figure pas dans `$routes`, elle affiche `Page introuvable` et s'arrête.

Pour `/posts`, l'instruction `[`$class$`,method] = `$routes$`[path];` copie les deux valeurs de la route dans deux variables. `$class` contient alors le nom `PostController`, tandis que `$method` contient le nom `index`.

L'instruction `new $class()` crée ensuite un objet de la classe nommée dans `$class`. Enfin, $controller->$method()$ exécute sur cet objet la méthode nommée dans `$method`. PHP permet donc d'utiliser des variables pour indiquer la classe à créer et la méthode à exécuter.

Le fichier `app.php` contient le code que vous écririez dans l'application : le contrôleur et la liste des routes. Pour simuler l'adresse demandée dans un navigateur, le script lit `$argv[1]`, le premier argument écrit après `php app.php`. Avec `php app.php /posts`, cette variable contient donc `/posts`.

```php
<?php

require __DIR__ . '/framework.php';

// Votre code : une classe, une méthode, une table des routes
class PostController
{
    public function index(): string
    {
        return 'Liste des articles';
    }
}

routes=['/posts'⇒[PostController::class,'index']];run(routes = ['/posts' ⇒
[PostController::class, 'index']];

run(routes=['/posts'⇒[PostController::class,'index']];run(routes, $argv[1]);
```

```
phpapp.php/postsListedesarticles php app.php /posts
Liste des articles
phpapp.php/postsListedesarticles php app.php /contact
Page introuvable
```

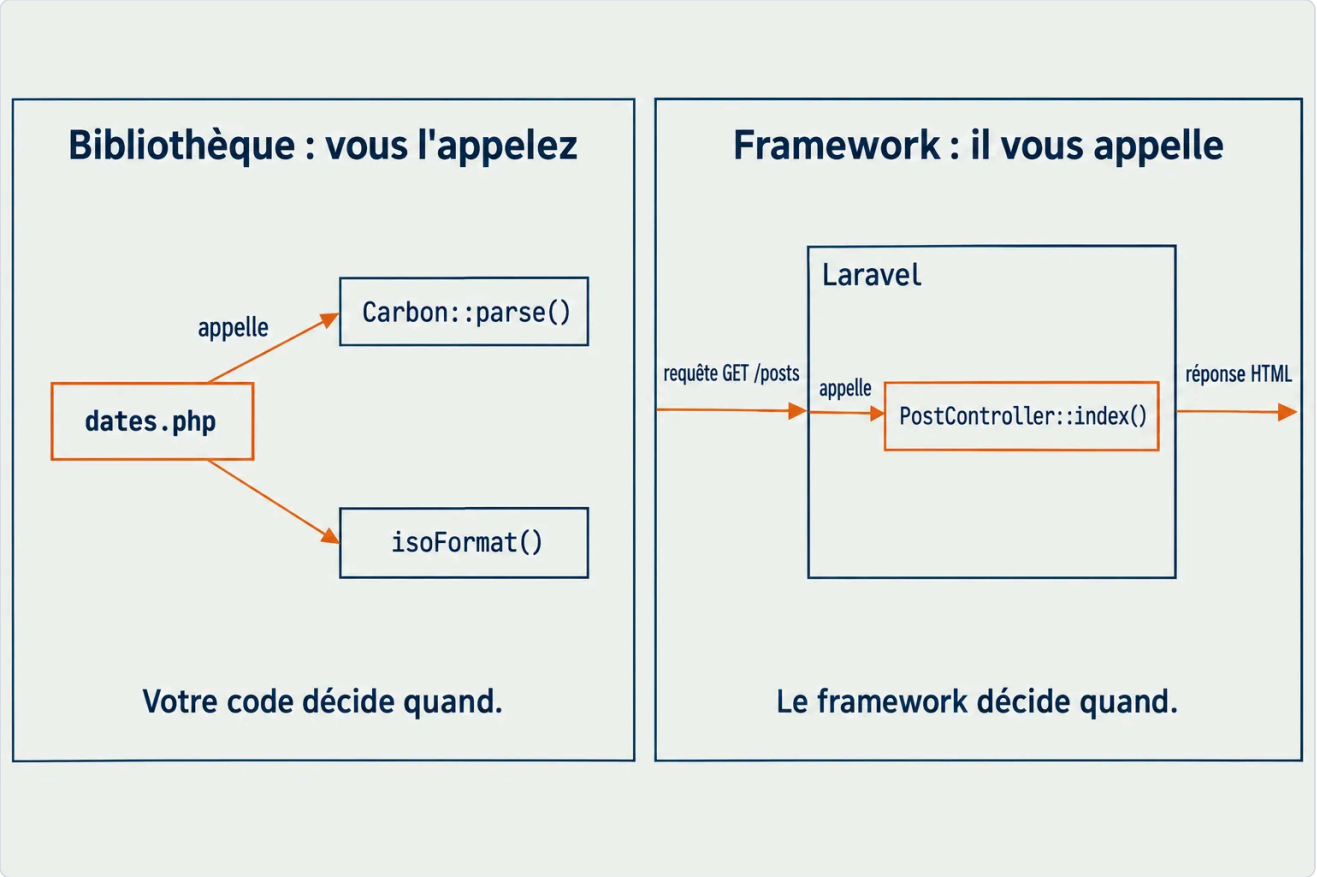
ASTUCE

Sans argument, `$argv[1]` n'existe pas : tapez `php app.php /posts`.

Votre fichier déclare `PostController`, sa méthode `index()` et la route associée à `/posts`. La fonction `run()` reçoit ensuite le chemin demandé. Pour `/posts`, elle crée un `PostController` et appelle `index()`. L'instruction qui lance la méthode se trouve donc dans le framework, tandis que la classe et la route se trouvent dans votre code.

Avec une bibliothèque, votre script commence par une instruction que vous avez écrite, comme `Carbon::parse()`. Avec Laravel, le point de départ se trouve dans le framework. Lorsqu'une requête arrive pour `/posts`, Laravel consulte la route, crée un `PostController`, puis exécute sa méthode `index()`.

Vous écrivez donc la route et la méthode, mais pas l'instruction qui appelle `index()`. Cette organisation s'appelle l'**inversion de contrôle**. À l'intérieur de `index()`, votre code utilise ensuite des fonctions de Laravel, comme `view()`, de la même manière qu'il utiliserait une bibliothèque.



> **NOTE**
>
> En 1985, Richard Sweet décrit ce renversement par une formule de Hollywood : « Ne nous appelez pas, c'est nous qui vous appellerons ». C'est la réponse faite aux acteurs après une audition. En anglais, on parle depuis du « principe d'Hollywood ».

4. Ce que ça change pour vous

Pour charger une classe ou une vue, Laravel construit son emplacement à partir de règles connues. Ces règles s'appellent des **conventions** : le framework fixe à l'avance les noms et les dossiers habituels. Vous n'avez donc pas à indiquer séparément chaque emplacement.

Laravel place les contrôleurs dans `app/Http/Controllers`, les modèles dans `app/Models` et les vues dans `resources/views`. Il associe aussi le modèle `Post` à la table `posts` en appliquant une règle de pluriel anglais.

Ce que vous gagnez :

- **Le framework fournit les mécanismes communs.** Ses classes gèrent notamment le routage, les sessions, la protection des formulaires et l'envoi de courriels.

- **Les projets emploient les mêmes dossiers.** Dans un autre projet Laravel, vous chercherez encore les contrôleurs dans `app/Http/Controllers`.
- **Certaines protections s'appliquent automatiquement.** Les doubles accolades de la vue échappent le HTML sans appel direct à `htmlspecialchars()`, comme le montre la sortie de la section 3.

Ce que vous perdez :

- **Le déroulement est plus difficile à suivre.** Dans `index.php`, vous lisez toutes les instructions dans leur ordre. Pour afficher `/posts`, Laravel charge environ trois cents classes, dont quatre seulement se trouvent dans vos fichiers. En cas d'erreur, vous devez repérer vos fichiers parmi ceux que le message affiche.
- **Votre code doit respecter les conventions.** Si le nom ou le dossier d'un fichier ne correspond pas à la règle attendue, Laravel ne charge pas l'élément concerné. Le message d'erreur n'indique pas toujours quelle convention a été enfreinte.
- **Le projet doit suivre les versions du framework.** Laravel publie une version majeure par an et fournit les correctifs de sécurité pendant deux ans. Vous devez donc adapter le projet à une nouvelle version au moins tous les deux ans.

5. Panorama 2026

Plusieurs langages proposent des frameworks web. Sur le serveur, ces frameworks reçoivent la requête envoyée par le navigateur et produisent la réponse.

Nom	Langage	Depuis	Une phrase
Laravel	PHP	2011	Le framework de ce cours, qui fournit de nombreux réglages par défaut.
Symfony	PHP	2005	Un framework PHP né en France, qui permet de déclarer précisément sa configuration.
Django	Python	2005	Un framework né dans la rédaction d'un journal américain, qui fournit les outils principaux d'une application.
Ruby on Rails	Ruby	2004	Le framework qui a popularisé les noms et les dossiers fixés par défaut plutôt que leur configuration séparée.
Express	JavaScript (Node.js)	2010	Un framework centré sur le serveur et le routage, auquel vous ajoutez les autres outils nécessaires.
Next.js	JavaScript (Node.js)	2016	Un framework construit autour de React, capable de produire des pages sur le serveur.
Spring	Java	2002	Un framework Java utilisé notamment dans les grandes entreprises.
ASP.NET Core	C#	2016	Le framework web de Microsoft, réécrit pour tourner hors de Windows.

Un **composant** est une partie d'interface réutilisable, comme un bouton ou une carte d'article. Dans le navigateur, plusieurs outils construisent l'interface en exécutant le code de ces composants.

Nom	Langage	Depuis	Une phrase
React	JavaScript	2013	Créé chez Facebook, il se présente comme une bibliothèque et non comme un framework.
Vue	JavaScript	2014	Un framework souvent associé à Laravel, que vous pouvez ajouter à une partie seulement d'une page.
Angular	TypeScript	2016	Un framework de Google qui a remplacé AngularJS.
Svelte	JavaScript	2016	Un framework qui transforme les composants en code JavaScript avant leur exécution dans le navigateur.

Le nom donné à un outil ne suffit pas toujours pour le classer. Avec React, vous écrivez des composants, puis React exécute leur code pour produire l'interface. React se présente malgré tout comme une bibliothèque, car il n'organise pas à lui seul toute l'application. Next.js ajoute notamment les routes, le démarrage du serveur et des règles pour organiser les fichiers. Il remplit donc le rôle d'un framework.

Le paquet `laravel/framework` déclare onze paquets Symfony parmi ses dépendances, dont `symfony/routing` et `symfony/http-foundation`. Il utilise aussi trois **polyfills**, des paquets qui apportent à une ancienne version de PHP une fonction plus récente. Laravel utilise Carbon pour les dates.

Un framework utilise donc souvent plusieurs bibliothèques. Il ajoute un programme de démarrage et des conventions qui déterminent le code à exécuter pour chaque requête.

À retenir

- Une bibliothèque fournit des fonctions que votre programme exécute à des endroits précis.
- Un framework reçoit la demande du navigateur, consulte la route et exécute la méthode indiquée.
- Pour distinguer les deux outils, repérez le code qui contient l'instruction d'appel.
- Les conventions associent des noms et des types de fichiers à des dossiers fixés par Laravel.
- Pour `/posts`, la route fournit à Laravel la classe `PostController` et la méthode `index`.

Exercices

1. Bibliothèque ou framework ?

Sur papier, classez ces huit outils. Pour chacun, écrivez une phrase qui nomme le code exécuté et l'instruction qui déclenche son exécution.

1. Carbon
2. Laravel
3. PHPMailer, qui envoie des courriels depuis un script PHP
4. Django
5. `league/csv`, qui lit et écrit des fichiers CSV
6. Pest

7. Symfony
8. React

Indice pour Pest : vous écrivez `test('…', function () { … })`. Cherchez quel programme exécute la fonction transmise à `test()`. Pour React, relisez la section 5.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/framework-et-librairie

2. Qui écrit, qui appelle ?

Reprenez le `PostController` de la section 3 et la ligne de `routes/web.php`. Pour chaque instruction, indiquez qui l'écrit, puis quel programme l'exécute et à quel moment.

- `Route::get('/posts', [PostController::class, 'index']);`
- `class PostController extends Controller`
- `public function index(): View`
- `$posts = Post::latest()->get();`
- `return view('posts.index', ['posts' => $posts]);`

Une instruction écrite dans votre fichier peut être exécutée par Laravel. Une déclaration de classe n'est pas une action. Pour cette ligne, indiquez quel code crée l'objet avec `new`. Comparez vos réponses avec `framework.php` et `app.php` de la section 3.

3. Cinq questions de révision

Quelle question distingue une bibliothèque d'un framework ?

Cherchez où se trouve l'instruction qui lance le code. Votre programme contient l'appel à une bibliothèque. Un framework contient l'appel à certaines méthodes de votre application.

Qu'est-ce que l'inversion de contrôle ?

Votre projet déclare une route et une méthode, mais ne contient pas l'instruction qui crée le contrôleur et exécute cette méthode. Quand le navigateur demande `/posts`, Laravel lit la route, crée `PostController` et exécute `index()`. Dans cette méthode, votre code peut ensuite utiliser Laravel avec `Post::latest()`.

Qu'est-ce qu'une dépendance ?

Une dépendance est un paquet nécessaire au fonctionnement de votre code. `composer require nesbot/carbon` a installé Carbon et six autres paquets.

Pourquoi un framework impose-t-il des conventions ?

Pour déduire certains noms et emplacements sans configuration supplémentaire. Laravel charge les contrôleurs dans `app/Http/Controllers` et associe le modèle `Post` à la table `posts`.

Laravel est-il écrit à partir de zéro ?

Laravel n'est pas écrit à partir de zéro. Il utilise Carbon pour les dates, onze paquets Symfony et trois polyfills. Il ajoute autour de ces bibliothèques son démarrage et ses conventions.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/framework-et-librairie

3.2 Laravel

Bloc Laravel · séance 1 · ~35 min

Slides du chapitre (PDF) PDF

Laravel est un **framework PHP**, c'est-à-dire un programme qui organise une application PHP et exécute certaines parties de votre code lorsqu'il reçoit une **requête HTTP**. Cette requête contient notamment l'adresse demandée par le navigateur. Taylor Otwell, un développeur de l'Arkansas, a créé Laravel en 2011. Quinze ans plus tard, une entreprise et une équipe développent le framework ainsi que des outils gratuits et payants. La version utilisée par un projet détermine les versions de PHP compatibles et les correctifs encore disponibles.

Compétences visées

- situer les grandes dates de Laravel, de la première bêta en 2011 à Laravel 13 en 2026.
- expliquer ce qui distingue Laravel d'un code PHP écrit à la main, sur un exemple.
- citer deux forces et deux limites de Laravel, avec un exemple pour chacune.
- lire la table des versions pour savoir si un projet reçoit encore des correctifs, puis ouvrir la bonne documentation.

1. D'où vient Laravel

En 2011, Taylor Otwell travaille avec CodeIgniter, un framework PHP très répandu à l'époque. CodeIgniter ne fournit pas l'**authentification**, le mécanisme qui vérifie l'identité de la personne connectée. Les développeurs doivent donc ajouter ce mécanisme à chaque projet. Taylor Otwell écrit son propre framework pour le fournir directement. La première bêta de Laravel sort le 9 juin 2011.

En mai 2013, Laravel 4 remplace les versions précédentes par une nouvelle organisation. Ses différentes parties deviennent des **paquets**, des ensembles de code PHP installés séparément. **Composer** est l'outil qui installe les paquets demandés par un projet PHP.

Pour certaines fonctions, Laravel utilise aussi des paquets du framework Symfony. La documentation de Symfony appelle ces paquets des **composants**. Ce rapprochement avait commencé plus tôt : Laravel 3.2 utilisait déjà un composant Symfony en 2012.

Dans le projet Laravel 13 que vous créerez au chapitre Installation, cette commande affiche les paquets Symfony installés :

```bash
composer show 'symfony/*' --name-only | grep -v -e polyfill -e contracts
```

```
symfony/clock
symfony/console
symfony/error-handler
symfony/event-dispatcher
symfony/finder
symfony/http-foundation
symfony/http-kernel
symfony/mailer
…
```

La sortie complète compte seize lignes. Les huit lignes ci-dessus en montrent une partie. Le paquet `symfony/http-foundation` fournit les classes qui représentent la requête HTTP reçue par l'application. Le paquet `symfony/console` fournit une partie du code utilisé par **Artisan**, l'outil Laravel qui exécute les commandes écrites sous la forme `php artisan`.

Laravel réutilise donc du code maintenu par Symfony pour plusieurs tâches. Il ajoute autour de ces paquets ses propres classes, ses règles de nommage et les méthodes que vous utilisez dans un projet Laravel.

Sous Windows, PowerShell n'a pas `grep`. Remplacez-le par `Select-String` :

```bash
composer show 'symfony/*' --name-only | Select-String -NotMatch 'polyfill|contracts'
```

NOTE

Histoire. Symfony a été créé par Fabien Potencier chez SensioLabs, une entreprise française, et publié en logiciel libre en octobre 2005. Laravel arrive six ans plus tard. Un projet Laravel 13 installe seize paquets Symfony, dont onze sont demandés directement par `laravel/framework`. Ce compte exclut les **polyfills**, qui reproduisent des fonctions absentes d'une version de PHP, et les **contrats**, qui définissent des interfaces communes entre paquets.

En mai 2014, Taylor Otwell lance Laravel Forge, un service payant qui prépare et configure des serveurs pour y déployer une application Laravel. Les revenus de Forge permettent ensuite à Taylor Otwell de travailler à plein temps sur Laravel.

Depuis Laravel 9, en février 2022, une **version majeure** sort chaque année, en début d'année. Ce type de version peut modifier ou supprimer un fonctionnement utilisé par le code existant. Avant de passer de Laravel 12 à Laravel 13, vous devez donc lire le guide de mise à jour et adapter les passages concernés.

Le 5 septembre 2024, Laravel lève 57 millions de dollars auprès du fonds d'investissement Accel. Une levée de fonds, c'est de l'argent apporté par des investisseurs en échange d'une part de l'entreprise. L'argent finance Laravel Cloud, un service d'hébergement lancé le 24 février 2025, le même jour que Laravel 12. Laravel 13 sort le 17 mars 2026.

- 9 juin 2011 : première bêta de Laravel
- Mai 2013 : Laravel 4, sur des composants Symfony
- 2013 : Laracasts
- Mai 2014 : Laravel Forge
- Février 2022 : Laravel 9, une version majeure par an
- 5 septembre 2024 : 57 millions de dollars d'Accel
- 24 février 2025 : Laravel 12 et Laravel Cloud
- 17 mars 2026 : Laravel 13

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/laravel

2. Les personnes

Laravel est développé par une équipe qui travaille à distance depuis plusieurs pays. Ses membres publient du code, de la documentation et des présentations.

Taylor Otwell a créé Laravel. Il dirige aujourd'hui l'entreprise Laravel.

- **Nuno Maduro** a créé Pest, le **lanceur de tests** qui exécute les tests et affiche leurs résultats. Vous l'utilisez depuis le bloc POO. Il a aussi créé Pint, qui remet votre code en forme, et Collision, qui affiche les erreurs dans le terminal.
- **Jess Archer** a créé Laravel Prompts, les questions interactives de l'installeur, et, avec Tim MacDonald, Laravel Pulse, un tableau de bord des performances. Elle a présenté Nightwatch, le service de surveillance des applications en production.
- **Dries Vints**, membre de l'équipe centrale, maintient les paquets libres de Laravel.
- **Tim MacDonald** est membre de l'équipe centrale et le premier contributeur des dépôts de Pulse et de Nightwatch.
- **Joe Dixon** a créé Reverb, le serveur de Laravel pour les pages mises à jour en temps réel, et encadre une équipe d'ingénieurs chez Laravel.

D'autres personnes ne travaillent pas dans l'équipe de Laravel, mais publient des outils utilisés dans de nombreux projets.

- **Jeffrey Way** a fondé Laracasts en 2013, le site de cours vidéo sur Laravel.
- **Caleb Porzio** a créé Livewire et Alpine.js, deux outils pour rendre une page interactive sans écrire d'application JavaScript séparée.
- **Jonathan Reinink** a créé Inertia, qui relie une application Laravel à une interface en Vue, React ou Svelte. Il est aussi l'un des auteurs de Tailwind CSS.
- **Freek Van der Herten** travaille chez Spatie, une entreprise d'Anvers qui maintient plus de 500 paquets libres pour PHP et Laravel. Certains, comme `spatie/laravel-permission`, sont dans un grand nombre de projets Laravel.
- **Marcel Pociot** fait partie de Beyond Code, l'entreprise qui a créé Herd.

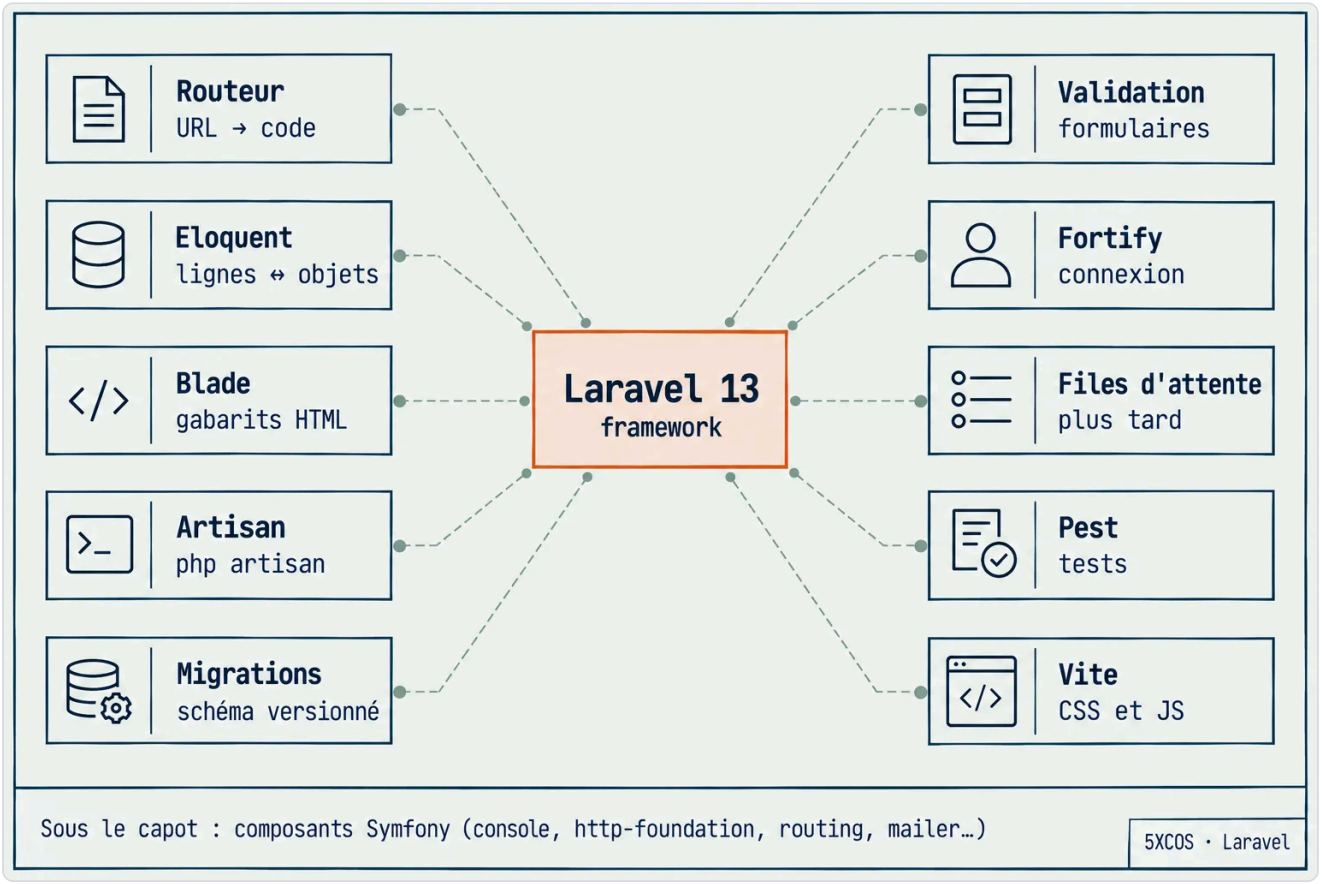
3. Ce que Laravel contient

Au chapitre Framework et librairie, vous avez vu qu'un framework reçoit la requête et consulte une **route**, une règle qui associe l'adresse demandée à du code. Cette route indique le **contrôleur**, la classe dont une méthode doit traiter la requête. Laravel fournit aussi les outils nécessaires aux tâches courantes d'une application web.

- Un **ORM**, pour *Object-Relational Mapper*, relie les tables de la base aux objets PHP. Lors d'une lecture, il transforme chaque ligne obtenue en objet, par exemple une ligne de `posts` en objet `Post`. Lors d'un enregistrement, il utilise les propriétés de cet objet pour créer ou modifier la ligne correspondante.
- Une **migration** est une classe PHP qui décrit une modification de la structure de la base, ses tables et leurs colonnes, par exemple « créer la table `posts` ». Elle est versionnée avec le reste du code.
- Une **file d'attente** enregistre une tâche pour qu'elle soit exécutée après la réponse au navigateur. Par exemple, le contrôleur enregistre la tâche « envoyer ce courriel », puis termine la requête sans attendre l'envoi. Un programme Laravel lancé en arrière-plan récupère ensuite la tâche et envoie le courriel.
- Un **cache** conserve temporairement un résultat déjà calculé. L'application peut le relire au lieu de recommencer le même calcul ou la même requête.
- Un **starter kit** est un projet de départ officiel qui contient déjà l'inscription, la connexion et leurs pages.

- Un **template** est un fichier qui contient le HTML de la page et des emplacements pour les données. Le moteur de templates Blade remplace ces emplacements, par exemple par le titre d'un article, avant d'envoyer le HTML au navigateur.

Besoin	Composant	Nom dans Laravel
Associer une URL à du code	le **routeur**	`Route::get()` dans `routes/web.php`
Lire et écrire en base avec des objets	l'ORM	Eloquent
Produire du HTML	le moteur de templates	Blade
Lancer des commandes dans le terminal	la **console**	**Artisan**, `php artisan`
Faire évoluer la structure de la base	les migrations	`php artisan migrate`
Vérifier les données d'un formulaire	la **validation**	`$request->validate()`
Inscrire et connecter des utilisateurs	l'authentification	Fortify, starter kits
Remettre un travail lent à plus tard	les files d'attente	Queue, `dispatch()`
Tester l'application	le lanceur de tests	Pest ou PHPUnit
Compiler le CSS et le JavaScript	l'outil de compilation	**Vite**



Laravel désigne à la fois le framework libre et l'entreprise qui développe des services autour de lui. Forge aide à configurer un serveur, Cloud héberge l'application, Nova fournit une interface d'administration et Nightwatch surveille une application en production.

Herd installe les outils du développement local et propose aussi des fonctions payantes. D'autres projets utilisés avec Laravel, comme Livewire, Inertia et Pest, restent gratuits. Ces produits ne font pas partie des fonctions du framework et certains sont vendus séparément.

4. Ce qui le distingue

Dans le bloc POO, vous avez déjà lu ce contrôleur :

```php
<?php

namespace App\Http\Controllers;

use App\Models\Post;
use Illuminate\View\View;

class PostController extends Controller
{
    public function index(): View
    {
        posts=Post::latest()->get();returnview(’posts.index’,[’posts’⇒posts =
Post::latest()→get();

        return view('posts.index', ['posts' ⇒ posts=Post::latest()
->get();returnview(’posts.index’,[’posts’⇒posts]);
    }
}
```

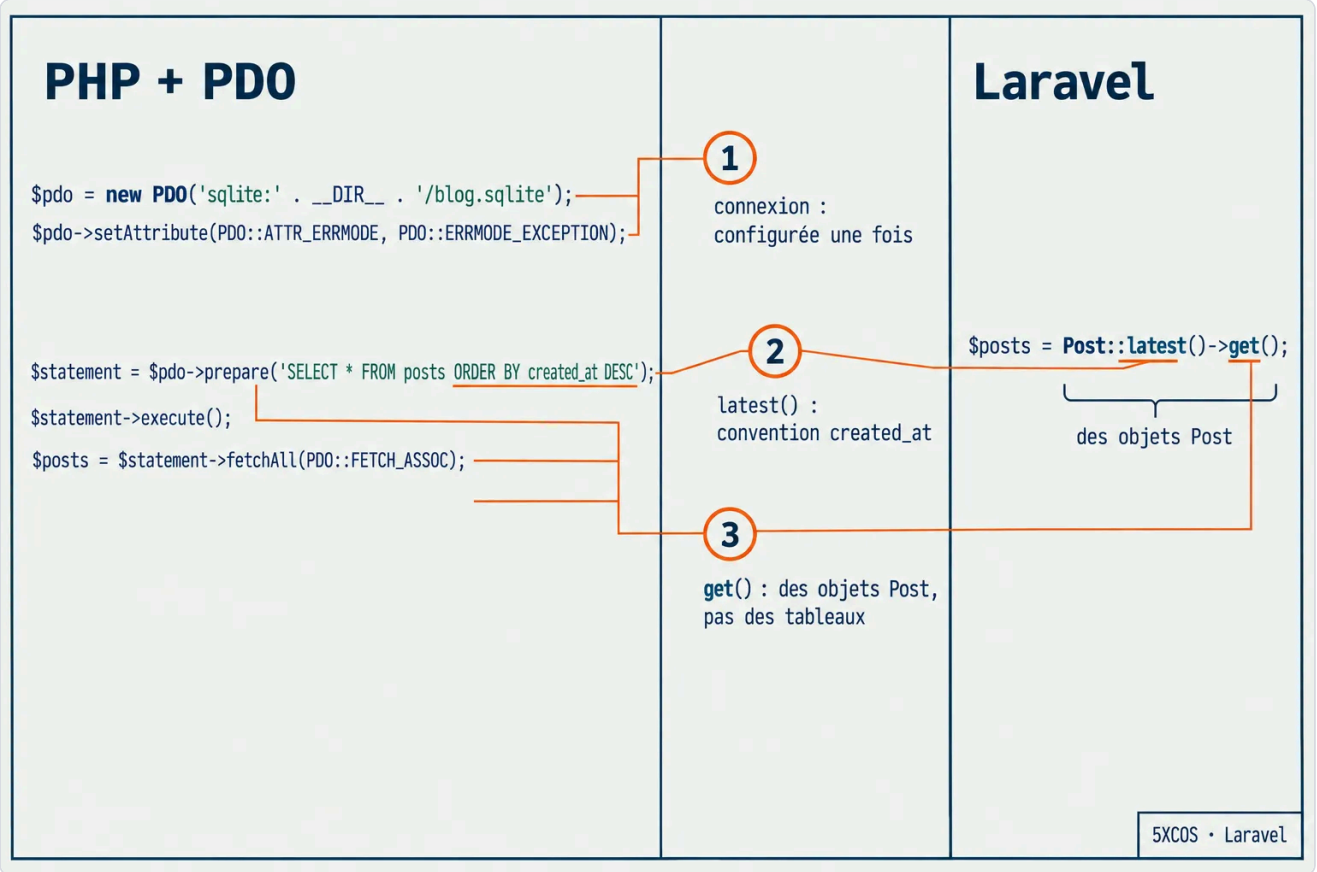
La ligne `Post::latest()→get()` lit tous les articles, du plus récent au plus ancien. **PDO** est une extension de PHP qui permet d'envoyer des requêtes SQL à une base de données. Sans framework, la même lecture dans une base SQLite qui contient deux articles s'écrit ainsi :

```php
<?php

pdo = new PDO('sqlite:' . __DIR__ . '/blog.sqlite');
pdo→setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

statement=statement = statement=pdo→prepare('SELECT * FROM posts ORDER BY created_at
DESC');
statement->execute();statement→execute();
statement->execute();posts = statement->fetchAll(PDO::FETCHASSOC);foreach(statement
→fetchAll(PDO::FETCH_ASSOC);

foreach (statement->fetchAll(PDO::FETCHASSOC);foreach(posts as $post) {
    echo $post['created_at'] . ' ' . $post['title'] . PHP_EOL;
}
```

```
2026-09-24 19:30:00 Installer Laravel 13
2026-09-20 18:00:00 Bienvenue sur le blog
```

Dans un projet Laravel 13 relié à la même base, une boucle placée après `Post::latest()->get()` affiche exactement les deux mêmes lignes. Le code Laravel obtient ce résultat autrement.

- Dans la version PDO, votre fichier crée la connexion, choisit SQLite, donne le chemin de la base et active le signalement des erreurs. Dans Laravel, ces réglages ne se trouvent pas dans le contrôleur. Le fichier `.env` contient les réglages propres à votre machine. Sa ligne `DB_CONNECTION=sqlite` choisit SQLite, tandis que `config/database.php` fournit le chemin par défaut. Laravel crée ensuite la connexion et configure la gestion des erreurs avant d'exécuter le contrôleur. C'est pourquoi la lecture tient ici dans `Post::latest()->get()`.
- Vous n'écrivez pas la requête SQL. À partir de `latest()`, Eloquent construit `order by "created_at" desc`, car Laravel utilise par défaut la colonne `created_at` pour la date de création.
- PDO renvoie ici des tableaux. Eloquent renvoie des objets `App\Models\Post`. Vous lisez donc un titre avec `$post->title`, et l'éditeur peut proposer les membres déclarés par la classe `Post`.



L'expression **expérience développeur** désigne ici ce qui facilite le travail quotidien de la personne qui programme. Dans cet exemple, Laravel réduit le nombre de lignes à écrire, place la configuration dans des fichiers prévus pour cela et renvoie des objets `Post` prêts à utiliser. Ces choix permettent d'obtenir plus vite le résultat attendu, mais cachent aussi une partie des opérations effectuées par PDO.

Des conventions. Une **convention** est une règle appliquée par défaut. Laravel associe la classe `Post` à la table `posts` grâce à une règle de nommage. Il place aussi les contrôleurs dans `app/Http/Controllers`. Lorsque votre projet respecte ces règles, vous n'avez pas besoin d'indiquer séparément le nom de la table ni l'emplacement habituel de ces classes.

Des outils regroupés. Le framework fournit l'authentification, la validation, les files d'attente, le cache, l'envoi de courriels et les tests. Leur documentation se trouve sur le même site et leurs fichiers suivent l'organisation d'un projet Laravel.

Une documentation par version. Chaque page présente d'abord un exemple de code, puis en explique l'utilisation. Une documentation distincte accompagne chaque version de Laravel.

Des ressources extérieures. Laracasts publie des cours, Laravel News des articles, Spatie des paquets et les conférences Laracon des présentations. Vous disposez ainsi d'exemples qui complètent la documentation officielle.

5. Forces et limites

Forces

- **La productivité.** Un projet neuf contient déjà un routeur, une base SQLite, des tests et un serveur de développement. Dans l'exemple précédent, le contrôleur lit les articles avec une seule ligne d'Eloquent.
- **Des outils réunis dans le même projet.** Les commandes de génération d'Artisan créent les fichiers aux emplacements attendus par Laravel. Eloquent fournit les données, puis Blade les affiche. La documentation présente ces outils ensemble et reprend les noms des dossiers du projet. Vous passez ainsi d'un outil à l'autre sans apprendre une nouvelle organisation pour chacun.
- **Les paquets disponibles.** Pour ajouter une fonction courante, comme les rôles, les paiements ou l'envoi de SMS, vous pouvez chercher un paquet conçu pour Laravel au lieu d'écrire toute la fonction vous-même.
- **Un calendrier connu.** Laravel existe depuis quinze ans, une entreprise finance son développement et une version majeure sort chaque année. Les dates de fin des correctifs permettent de prévoir les mises à jour d'un projet.

Limites

- **Certains appels cachent plusieurs étapes.** Quand vous lisez `Post::latest()`, vous pouvez chercher `latest()` dans `Post` et ne pas la trouver. PHP exécute alors `__callStatic()`, une méthode spéciale héritée de la classe `Model`. Cette méthode reçoit le nom `latest`, crée l'objet qui construit la requête SQL et lui transmet l'appel. Les classes `Cache` et `DB`, appelées **façades**, utilisent une idée comparable. La méthode `__callStatic()` de leur classe parente récupère un objet préparé par Laravel, puis exécute sur cet objet la méthode demandée. L'écriture `Cache::get()` réduit ainsi le code à taper, mais rend le trajet du programme moins visible lorsque vous cherchez une erreur.
- **Le modèle Active Record.** Avec Eloquent, un objet `Post` représente une ligne de `posts`. Il contient par exemple son titre et son texte. Le même objet peut modifier cette ligne lorsque vous appelez `$post→save()`. Cette organisation s'appelle **Active Record**. Elle réunit dans la classe `Post` les données de l'article et l'accès à la base. Une règle placée dans cette classe peut alors être plus difficile à tester seule, car `Post` dépend aussi d'Eloquent et, selon le test, d'une base de données.
- **La dépendance aux outils choisis.** Chaque paquet ajouté peut imposer des versions de PHP ou de Laravel et possède son propre calendrier de maintenance. Avant une mise à jour, vous devez donc vérifier que les paquets et services utilisés restent compatibles.

- **L'image de PHP.** Certains développeurs associent encore PHP aux limites de ses versions des années 2000. Laravel ne change pas cette opinion à lui seul, et vous pouvez rencontrer ce jugement en entretien.

Symfony demande souvent d'écrire ou de choisir plus directement la configuration qui relie les parties de l'application. Laravel fournit davantage de règles par défaut. Par exemple, la classe `Post` utilise automatiquement la table `posts`. Le choix de Symfony rend certains réglages plus visibles. Celui de Laravel réduit le code nécessaire lorsque ses conventions conviennent au projet. En 2025, dans l'enquête Stack Overflow, 8,9 % des répondants disaient utiliser Laravel, et 4 % Symfony.[1]

6. Versions et support

Une **correction de bug** répare un comportement qui ne correspond pas au résultat prévu. Un **correctif de sécurité** ferme une faille qu'un attaquant pourrait utiliser. Pour chaque version, Laravel publie la date de fin de ces deux types de correctifs.

Version	PHP	Sortie	Fin des corrections de bugs	Fin des correctifs de sécurité
10	8.1 à 8.3	14 février 2023	6 août 2024	4 février 2025
11	8.2 à 8.4	12 mars 2024	3 septembre 2025	12 mars 2026
12	8.2 à 8.5	24 février 2025	13 août 2026	24 février 2027
13	8.3 à 8.5	17 mars 2026	3e trimestre 2027	17 mars 2028

La règle officielle prévoit **une version majeure par an, 18 mois de corrections de bugs et 2 ans de correctifs de sécurité**. Deux versions reçoivent donc presque toujours des correctifs de sécurité en même temps. Leurs périodes de corrections de bugs ne se chevauchent que pendant quelques mois. Au 26 septembre 2026, Laravel 13 reçoit les deux types de correctifs. Laravel 12 ne reçoit plus que les correctifs de sécurité. Laravel 11 et Laravel 10 ne reçoivent plus aucun correctif.

La version de PHP impose une seconde limite. Même si Laravel 13 reçoit encore tous ses correctifs, vous ne pouvez pas l'installer chez un hébergeur qui ne propose que PHP 8.2.

La documentation se trouve à l'adresse `laravel.com/docs/13.x`, qui mène aujourd'hui à `laravel.com/framework/docs/13.x`.

- **Le sélecteur de version** affiche la documentation d'une version précise. Vérifiez qu'il correspond à la version du projet ouvert. La ligne `"laravel/framework"` de `composer.json` contient ce numéro.
- **Le menu de gauche** classe les pages par thème : Prologue, Getting Started, Architecture Concepts, The Basics, Digging Deeper, Security, Database, Eloquent ORM, AI, Testing, Packages.
- **Le bouton « Copy as markdown »** copie le texte de la page. Vous pouvez fournir ce texte à un assistant IA afin que sa réponse repose sur la documentation de la bonne version.

7. Ressources

- **La documentation officielle**, en anglais : https://laravel.com/docs/13.x

- **Laravel Learn**, deux mini-cours vidéo gratuits de l'équipe Laravel, dont un sur les bases de PHP : https://laravel.com/learn. L'ancienne adresse du Laravel Bootcamp y mène.
- **Laracasts**, des séries vidéo sur Laravel, PHP et les outils autour : https://laracasts.com
- **Laravel News**, l'actualité du framework et des paquets : https://laravel-news.com
- **Laravel Daily**, des articles et vidéos courts sur des cas concrets : https://laraveldaily.com
- **laravel.sillo.org**, un blog en français, à jour pour Laravel 13 : https://laravel.sillo.org

À retenir

- Taylor Otwell a créé Laravel en 2011 pour offrir ce que CodeIgniter n'avait pas, dont l'authentification.
- Laravel 4, en mai 2013, est découpé en paquets Composer et généralise l'usage des composants Symfony.
- Laravel regroupe le routeur, l'ORM Eloquent, les templates Blade, la console Artisan, les migrations, la validation, l'authentification, les files d'attente et les tests dans un même projet.
- Ses conventions réduisent la configuration à écrire, mais certains appels transmettent le travail à des objets que vous ne voyez pas dans votre classe.
- Chaque version majeure reçoit 18 mois de corrections de bugs et 2 ans de correctifs de sécurité, et la documentation se lit dans la version du projet.

Exercices

1. Lire la table des versions. À l'aide de la table des versions ci-dessus, répondez sur papier aux trois questions.

1. Un projet en Laravel 11 reçoit-il encore des correctifs de sécurité en octobre 2026 ?
2. Un client a un projet en Laravel 12. En octobre 2026, que reçoit-il encore, et jusqu'à quand ?
3. L'hébergeur d'un client ne propose que PHP 8.2. Quelle version de Laravel pouvez-vous installer, et quel problème aurez-vous en mars 2027 ?

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/laravel

2. Les dates et les personnes. Répondez dans votre tête, puis retournez la carte.

Quand sort la première bêta de Laravel, et qui l'a écrite ?

La première bêta sort le 9 juin 2011. Taylor Otwell l'a écrite pour ajouter ce qui manquait à CodeIgniter, dont l'authentification.

Sur quel autre framework Laravel s'appuie-t-il ?

Laravel s'appuie sur des composants Symfony. Le premier, HttpFoundation, arrive dès Laravel 3.2 en 2012, et Laravel 4 généralise leur utilisation en mai 2013. Le composant `symfony/console` fournit une partie du code utilisé par `php artisan`.

Qu'est-ce que Laravel Forge, et quand est-il lancé ?

Laravel Forge est un service payant qui prépare des serveurs pour y déployer une application Laravel. Il est lancé en mai 2014.

Quels événements marquent le 5 septembre 2024 et le 24 février 2025 ?

Le 5 septembre 2024, Laravel lève 57 millions de dollars auprès d'Accel. Le 24 février 2025 sortent Laravel 12 et Laravel Cloud.

Qui a créé Pest, et qui a créé Livewire ?

Nuno Maduro a créé Pest. Caleb Porzio a créé Livewire, ainsi qu'Alpine.js.

Quelle entreprise belge publie plus de 500 paquets pour Laravel, et où est-elle installée ?

Spatie publie ces paquets. Cette entreprise est installée à Anvers, où travaille Freek Van der Herten.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/laravel

---

1. Stack Overflow Developer Survey 2025, section « Web frameworks and technologies », tous répondants : https://survey.stackoverflow.co/2025/technology ↩

3.3 Environnements de développement PHP

Windows : Laragon, ton meilleur pote

Laragon, c'est le couteau suisse pour développer en PHP sous Windows. Plus rapide qu'un café un lundi matin, Laragon embarque PHP, Apache, MySQL et même Node.js en mode zéro prise de tête. Et tout ça, dans un dossier que tu peux déplacer à volonté, parce que toi aussi t'as le droit d'être bordélique.

Pourquoi Laragon est cool :

- **URLs sexy** : Oublie les `localhost/mon-super-projet` tout claqués, maintenant c'est `mon-super-projet.test`.
- **Isolation** : Garde ton Windows tout propre, enfin presque.
- **Facilité extrême** : Installer une nouvelle version de PHP ? Deux clics, même ton chat peut y arriver.

Installation rapide :

1. Télécharge Laragon sur le site officiel.
2. Lance l'installeur, choisis ton dossier favori (`C:\laragon` par défaut).
3. Clique sur "Installer", attends, prends une pause-café, et voilà !
4. Dans le menu PHP de Laragon, choisis la version 8.4 : c'est celle du cours, tout le groupe travaille avec.

macOS : Laravel Herd, l'environnement chic et rapide

Si tu préfères les pommes aux fenêtres, Laravel Herd est fait pour toi. Plus rapide que tes excuses quand ton code plante, Herd est spécialement conçu pour Laravel et PHP sur macOS.

Pourquoi Laravel Herd déchire :

- **Ultra-rapide** : PHP, nginx, dnsmasq boostés aux binaires statiques.
- **Prêt à coder** : Composer, Laravel et Node.js directement accessibles dans le terminal.
- **Pro attitude** : Gestion simplifiée des bases de données MySQL, Postgres et Redis avec la version Pro.

Installation simplissime :

1. Télécharge Laravel Herd sur le site officiel.
2. Ouvre le fichier DMG, glisse Laravel Herd dans le dossier Applications.
3. Lance Herd, donne-lui les permissions (allez, fais confiance), et c'est prêt à l'emploi.
4. Dans les réglages de Herd, onglet PHP, choisis la version 8.4 : c'est celle du cours, tout le groupe travaille avec.

Extensions indispensables pour VS Code (Laravel Edition)

Pour coder comme un vrai pro de Laravel sans avoir l'air trop noob, voici les extensions VS Code incontournables :

- **Laravel** : l'extension officielle, publiée par l'équipe Laravel. Autocomplétion des routes, des vues, des variables d'environnement et des règles de validation, navigation vers les fichiers, tout ça sans configuration. Installe-la en premier.
- **Laravel Extension Pack** : Le starter-pack ultime avec Blade Snippets, coloration syntaxique, autocomplétion, tout-en-un.

- **Intelephense** : Auto-complétion intelligente pour PHP, parce que coder en aveugle, c'est cringe.
- **PHP Namespace Resolver** : Gère les namespaces PHP automatiquement, car tu as mieux à faire.
- **Prettier** : Formatte ton code automatiquement pour éviter les débats inutiles avec tes collègues.
- **SQLite Viewer** : Visualise facilement tes bases SQLite sans sortir de VS Code, pour les flemmards qui s'assument.
- **Tailwind CSS IntelliSense** : Autocomplétion des classes Tailwind CSS, histoire de coder à la vitesse de la lumière.

Voilà, t'es équipé maintenant. Essaie de pas tout faire crasher !

3.4 Installation

Bloc Laravel · séance 1 · ~60 min

Slides du chapitre (PDF) PDF

Vous avez installé Herd sous macOS ou Laragon sous Windows. Utilisez maintenant cet environnement pour créer le projet `premier-test-laravel`, ouvrir sa page d'accueil et lancer ses tests. Le même projet sert au chapitre MVC et au chapitre Structure d'un projet Laravel.

À la fin de ce chapitre, vous serez capables de :

1. **vérifier** les outils nécessaires avec cinq commandes,
2. **créer** un projet Laravel 13 avec l'installeur et les réponses du cours,
3. **expliquer** ce que chaque étape de l'installation a produit dans le dossier,
4. **diagnostiquer** les pannes courantes à partir du message affiché.

1. Vérifier les prérequis

Laravel 13 utilise PHP et Composer. Il utilise aussi **Node.js**, un programme qui exécute du JavaScript hors du navigateur. Node.js fournit **npm**, l'outil qui installe les paquets JavaScript comme Composer installe les paquets PHP.

L'**installeur Laravel** est le programme `laravel`. Il crée les fichiers d'un projet à partir de vos réponses. Sous macOS, ouvrez l'application Terminal. Sous Windows, ouvrez PowerShell. Tapez les cinq commandes une par une.

```bash
php --version
composer --version
node --version
npm --version
laravel --version
```

```
PHP 8.4.21 (cli) (built: May 18 2026 08:45:15) (NTS clang 15.0.0)
Copyright (c) The PHP Group
Built by Laravel Herd
Zend Engine v4.4.21, Copyright (c) Zend Technologies
    with Zend OPcache v8.4.21, Copyright (c), by Zend Technologies
Composer version 2.8.6 2025-02-25 13:03:50
PHP version 8.4.21 (~/Library/Application Support/Herd/bin/php84)
Run the "diagnose" command to get more detailed diagnostics output.
v22.20.0
10.9.3
Laravel Installer 5.32.0
```

Les numéros exacts peuvent être plus récents. Vérifiez surtout la ligne qui commence par `PHP` : pour ce cours, elle doit indiquer 8.4. Laravel 13 fonctionne avec PHP 8.3, mais les paquets choisis pendant la création dépendent de la version de PHP utilisée. Afin que tout le groupe installe les mêmes versions, le cours utilise PHP 8.4.

Lorsqu'une commande répond « command not found » ou « n'est pas reconnu » :

- si `php`, `composer`, `node` ou `npm` manque, vérifiez l'installation de Herd ou de Laragon. Ces deux environnements fournissent les quatre commandes.
- si `laravel` manque ou affiche une version inférieure à 5.32, mettez l'installeur à jour.

Sous Herd, ouvrez le menu Settings et l'onglet PHP. Sur la ligne Laravel Installer, cliquez sur Update.

Sous Laragon ou sous Linux, installez-le avec Composer.

```bash
composer global require laravel/installer
```

```
Using version ^5.32 for laravel/installer
```

Sous Windows, le **PATH** est la liste des dossiers dans lesquels le terminal cherche les programmes demandés. Cette liste doit contenir `%APPDATA%\Composer\vendor\bin`. Fermez le terminal, rouvrez-le, puis relancez `laravel --version`.

Sous Linux, si ces outils ne sont pas encore installés, le script de php.new installe PHP 8.4, Composer et l'installeur en une commande.

```bash
/bin/bash -c "$(curl -fsSL https://php.new/install/linux/8.4)"
```

2. Créer le projet `premier-test-laravel`

Placez-vous dans le dossier utilisé par votre serveur local. Herd rend accessibles les projets placés dans `~/Herd`. Laragon fait de même avec le dossier `www` de son installation, situé dans `C:\laragon\www` avec le réglage par défaut. Un dossier nommé `premier-test-laravel` à cet endroit devient accessible à

l'adresse `http://premier-test-laravel.test`.

Sous macOS, avec Herd :

```bash
cd ~/Herd
```

Sous Windows, avec Laragon :

```bash
cd C:\laragon\www
```

Lancez ensuite l'installeur.

```bash
laravel new premier-test-laravel --git --no-boost
```

`premier-test-laravel` est le nom du dossier créé. L'option `--git` crée un **dépôt Git**, un projet dont Git enregistre l'historique dans le dossier caché `.git/`. Elle y enregistre l'installation en deux **commits**, deux états du projet accompagnés d'un message. L'option `--no-boost` crée le projet sans Laravel Boost, un paquet qui fournit des consignes aux assistants de code IA.

```
┌ Do you want to use a starter kit? ──────────────────────────┐
│ ○ Yes / ● No                                                │
└─────────────────────────────────────────────────────────────┘
```

Un **starter kit** est un ensemble de fichiers qui ajoute notamment les pages de connexion et d'inscription. Le projet du cours n'installe pas ce kit. Choisissez **No**, puis validez avec Entrée.

```
┌ Which frontend stack do you want to build on? ──────────────┐
│ › ● Blade                                                   │
│   ○ React                                                   │
│   ○ Svelte                                                  │
│   ○ Vue                                                     │
│   ○ Livewire                                                │
└──────────────────────────── No additional frontend scaffolding ┘
```

Gardez le choix proposé, **Blade**, puis validez. Blade est le **moteur de templates** de Laravel : il combine du HTML avec les données de l'application pour produire les pages envoyées au navigateur.

Avec ces réponses, le projet utilise aussi **Pest**, le programme qui lance les tests. Les données sont enregistrées avec **SQLite**, un système de base de données qui conserve ici toutes les tables dans un fichier. Les paquets JavaScript sont installés avec npm. L'installation prend environ une minute.

```
  • Creating Laravel application
    ✔ Application installed
    ✔ Application initialized
  • Running database migrations
    ✔ Database migrated
  • Initializing Git repository
    ✔ Repository initialized
  • Setting up Pest
    ✔ Pest installed
    ✔ Pest initialized
  …
  • Setting up frontend dependencies with npm
    ✔ Packages installed
    ✔ Assets built

  ┌ Application ready ──────────────────────────────────────────┐
  │ You can start your local development using:                 │
  │                                                             │
  │ 1. cd premier-test-laravel                                  │
  │ 2. composer run dev                                         │
  …
```

Le symbole `…` remplace les lignes de progression retirées de cette sortie.

Avec Herd, la fin de la sortie contient `Open: http://premier-test-laravel.test`.

> **ASTUCE**
>
> Si l'installeur répond `The "--no-boost" option does not exist.`, il est trop ancien. Mettez-le à jour comme à la section 1.

3. Ce que l'installeur vient de faire

Les coches correspondent aux opérations effectuées par l'installeur. Certaines opérations ont créé des dossiers ou des fichiers. D'autres ont lancé une commande.

Plusieurs commandes utilisent **Artisan**, l'outil de Laravel que vous démarrez avec le fichier `artisan` situé à la racine du projet. Entrez dans `premier-test-laravel`.

```bash
cd premier-test-laravel
```

La **clé d'application** est une longue chaîne aléatoire placée dans `.env`. Chaque projet possède sa propre clé, qui reste dans ce fichier.

Un **cookie** est une petite donnée conservée par le navigateur. Une **session** conserve les données d'un utilisateur entre deux pages. Laravel chiffre notamment les cookies et les sessions avec la clé d'application. Leur contenu reste ainsi illisible sans cette clé.

Une **migration** est un fichier PHP qui décrit une modification de la base de données. La commande `php artisan migrate` lit ces fichiers et crée ici les tables initiales.

Étape	Commande lancée pour vous	Ce qui apparaît
Paquets PHP	`composer install`	`vendor/`, une centaine de paquets
Configuration	copie de `.env.example`	`.env`, les réglages de cette machine
Clé d'application	`php artisan key:generate`	la ligne `APP_KEY=base64:…` dans `.env`
Base de données	création du fichier	`database/database.sqlite`
Migrations	`php artisan migrate`	les tables `users`, `cache`, `jobs` et leurs voisines
Git	`git init`, deux commits	`.git/`
Pest	`composer require pestphp/pest --dev`, puis `pest --init`	`tests/Pest.php`
Paquets JavaScript	`npm install`	`node_modules/`
Fichiers du navigateur	`npm run build`	`public/build/`

Vite prépare les fichiers CSS et JavaScript utilisés par le navigateur. Il lit les sources de `resources/css/` et `resources/js/`, puis repère les autres fichiers demandés par ces sources. Il produit les fichiers que le navigateur chargera. La commande `npm run build` effectue cette préparation une fois et écrit le résultat dans `public/build/`.

Les migrations fournies avec Laravel créent les tables `users`, `cache` et `jobs`.

Pest est installé avec l'option `--dev`. Une **dépendance de développement** est un paquet utile pour écrire ou tester le code, mais inutile au serveur qui affiche le site. Composer enregistre Pest sous la clé `require-dev` de `composer.json`, comme au chapitre 2 de la POO.

L'installeur a créé deux commits avant d'installer les paquets JavaScript. Vérifiez cet historique avec `git log`, puis repérez avec `git status` les changements encore absents d'un commit.

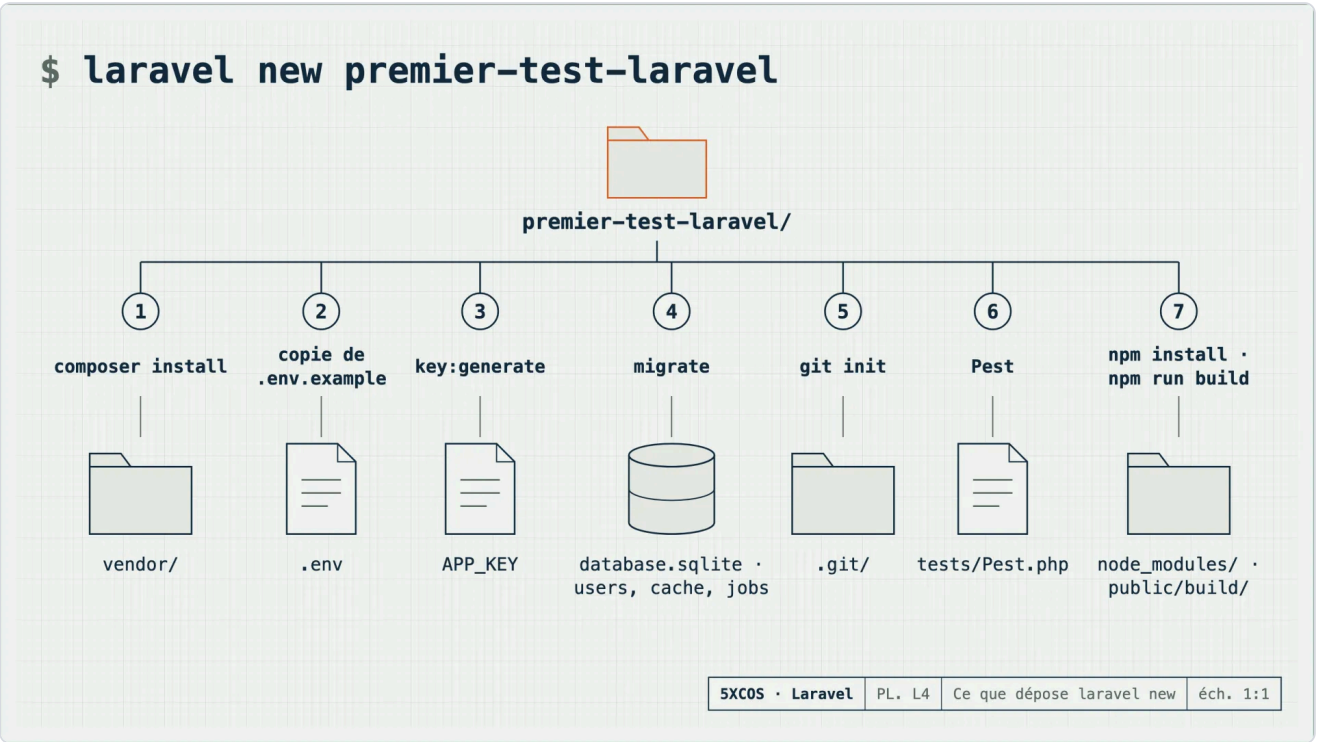
```bash
git log --oneline
git status --short
```

```
bd383a0 Install Pest
1304dbc Set up a fresh Laravel app
 M composer.json
?? package-lock.json
```

Les deux premières lignes montrent les commits créés par l'installeur. Leurs identifiants seront différents chez vous. L'installation avec npm a eu lieu après ces commits. Ses changements apparaissent donc encore dans `git status` : `package-lock.json`, qui contient la liste exacte des paquets JavaScript

installés, est nouveau. Le fichier `composer.json` a seulement été remis en forme. Enregistrez ces deux changements dans un troisième commit.

```bash
git add .
git commit -m "Installation de npm"
```



Le projet contient aussi deux fichiers identiques, `AGENTS.md` et `CLAUDE.md`. Leur texte fournit des consignes aux assistants de code IA qui analysent le projet. Ces consignes demandent l'installation de Laravel Boost, même si l'option `--no-boost` ne l'a pas installé. Refusez toute proposition de passer à PHP 8.5 : le groupe travaille avec PHP 8.4.

4. Voir la page d'accueil

Herd et Laragon rendent déjà le dossier `premier-test-laravel` accessible au navigateur. Ouvrez `http://premier-test-laravel.test`. Sous Laragon, si l'adresse ne répond pas, cliquez sur « Stop », puis sur « Start All ». Ce redémarrage charge la configuration du nouveau dossier.

Sans Herd ni Laragon, sous Linux par exemple, démarrez le serveur de développement avec `composer run dev`. Cette commande fonctionne aussi avec Herd ou Laragon.

```bash
composer run dev
```

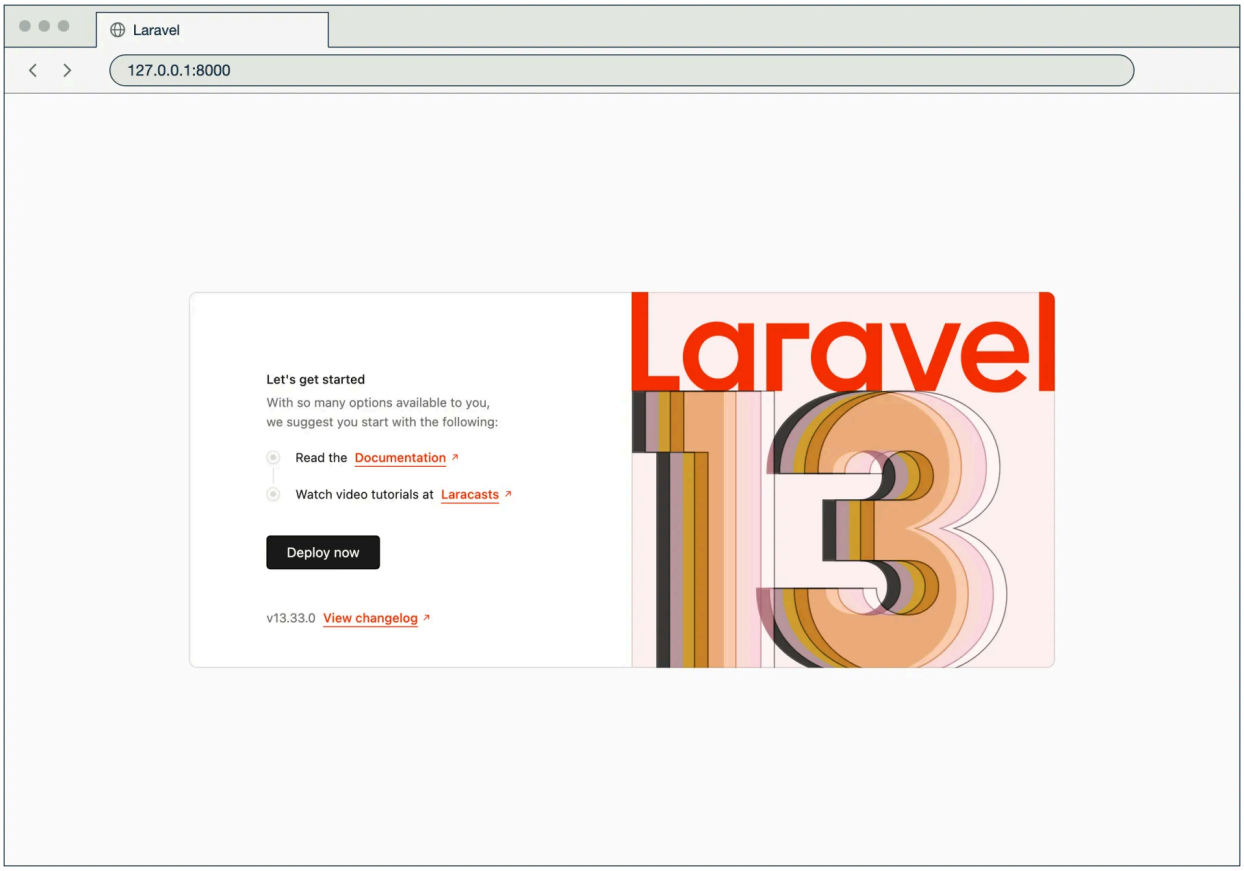
```
┌──────────────┐┌──────────────────────────────────────────────────────┐
│ 1 server     ││ php artisan serve                                    │
│ 2 queue      ├└──────────────────────────────────────────────────────┘
│ 3 logs       ├    INFO  Server running on [http://127.0.0.1:8000].
│ 4 vite       ├  Press Ctrl+C to stop the server
└──────────────┘└──────────────────────────────────────────────────────┘
 ↑/↓ scroll   tab tabs   c clear   / search   s stream   q quit
```

La sortie est raccourcie à droite. La commande `composer run dev` garde quatre programmes actifs pendant que vous travaillez. Chacun remplit une tâche différente :

- `server` démarre le serveur PHP. Il répond sur `http://127.0.0.1:8000`, qui désigne le même endroit que `http://localhost:8000`.
- `queue` exécute les tâches que l'application enregistre dans la **file d'attente**, comme l'envoi d'un courriel.
- `logs` affiche le **journal**, le fichier dans lequel Laravel écrit les erreurs et les autres messages de l'application.
- `vite` relance Vite après chaque modification du CSS ou du JavaScript, afin de reconstruire les fichiers du navigateur.

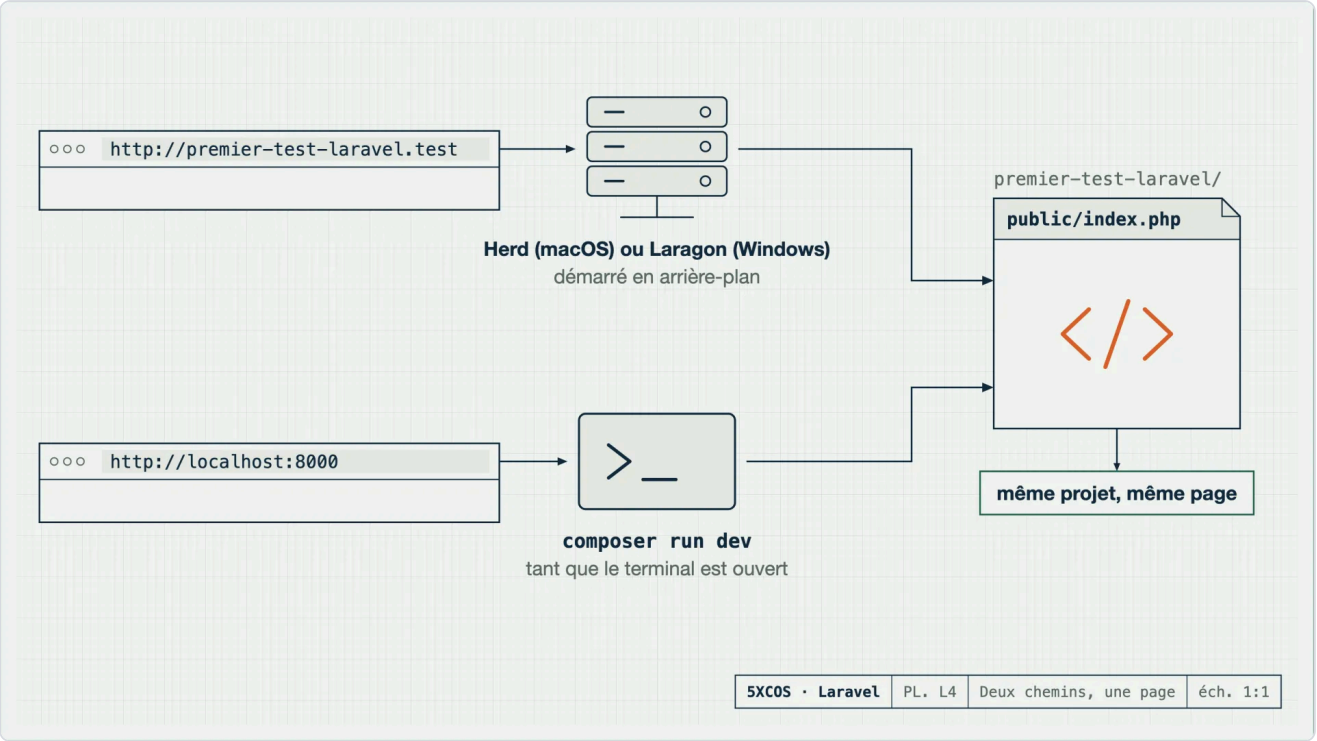
Sur macOS et Linux, appuyez sur `q` ou Ctrl+C pour arrêter les quatre programmes. Sous Windows, les messages de `server`, `queue` et `vite` apparaissent dans le même terminal. Chaque ligne commence par le nom du programme, et `logs` n'est pas lancé. Utilisez Ctrl+C pour tout arrêter.

Pendant le fonctionnement de ces programmes, vous ne pouvez pas taper une autre commande dans ce terminal. Ouvrez-en un second pour lancer Artisan ou les tests.



ASTUCE

Le fichier `.env` contient `APP_URL=http://localhost:8000`. Avec Herd ou Laragon, remplacez cette ligne par `APP_URL=http://premier-test-laravel.test`. Lorsque le code demande un lien complet, Laravel utilise cette adresse comme début du lien. C'est notamment le cas pour un lien placé dans un courriel.



5. Vérifier depuis le terminal

Dans le dossier `premier-test-laravel`, `php artisan --version` affiche la version de Laravel chargée par le projet.

```bash
php artisan --version
```

```
Laravel Framework 13.33.0
```

La commande `php artisan about` affiche les versions et les principaux réglages du projet.

```bash
php artisan about
```

```
Environment ..........................................................
Application Name ..................................................... Laravel
Laravel Version ...................................................... 13.33.0
PHP Version ........................................................... 8.4.21
Composer Version ...................................................... 2.8.6
Environment ..................................................... local
Debug Mode ..................................................... ENABLED
URL ..................................................... localhost:8000
…
Drivers ..............................................................
Broadcasting ..................................................... log
Cache ..................................................... database
Database ..................................................... sqlite
…
```

La sortie complète compte 29 lignes réparties en quatre blocs. Vérifiez `PHP Version`, puis `Database`, qui doit contenir `sqlite`. La ligne `URL` reprend la valeur de `APP_URL`. Après la modification de la section 4, elle affiche `premier-test-laravel.test`.

Un projet neuf contient deux tests. La commande `php artisan test` les lance.

BASH

```
php artisan test
```

```
  PASS  Tests\Unit\ExampleTest
✓ that true is true                                                0.01s

  PASS  Tests\Feature\ExampleTest
✓ the application returns a successful response                    0.10s

Tests:    2 passed (2 assertions)
Duration: 0.21s
```

Les deux lignes `PASS` confirment que PHP charge les paquets et la clé de `.env`, puis que la page d'accueil répond. La ligne `Database` de `php artisan about` vérifie séparément que le projet utilise SQLite.

6. La variante `composer create-project`

Composer peut aussi créer un projet Laravel sans passer par l'installeur. Vous rencontrerez cette commande dans des tutoriels, mais le cours la déconseille, pour les raisons données sous la sortie. N'exécutez pas `composer create-project laravel/laravel premier-test-laravel` maintenant, car le dossier `premier-test-laravel` existe déjà. Pour faire un essai séparé, remplacez `premier-test-laravel` par un autre nom, par exemple `essai`.

BASH

```
composer create-project laravel/laravel premier-test-laravel
```

```
…
> @php artisan key:generate --ansi

   INFO  Application key set successfully.

> @php -r "file_exists('database/database.sqlite') || touch('database/database.sqlite');"
> @php artisan migrate --graceful --ansi
…
```

`composer create-project` installe les paquets PHP, crée `.env`, ajoute la clé, crée le fichier SQLite et lance les migrations. Cette méthode conserve PHPUnit comme lanceur de tests à la place de Pest et ne crée pas de dépôt Git. Elle n'installe pas non plus les paquets JavaScript, donc `node_modules/` et `public/build/` sont absents.

Vous devez alors compléter le projet à la main : initialiser Git avec `git init`, lancer `npm install`, puis produire les fichiers du navigateur avec `npm run build`. Pest reste absent. Pour ces raisons, créez toujours vos projets avec `laravel new`.

7. Dépannage

ASTUCE

Mauvaise version de PHP. Toute commande `php artisan` s'arrête sur une erreur fatale qui contient ce message :

`Composer detected issues in your platform: Your Composer dependencies require a PHP version "⩾ 8.4.1". You are running 8.3.17.`

Le projet a été créé avec PHP 8.4, mais la commande `php` de ce terminal lance PHP 8.3.17. Choisissez PHP 8.4 dans Herd ou dans le menu PHP de Laragon. Fermez le terminal, rouvrez-le, puis vérifiez la version avec `php --version`.

ASTUCE

Clé d'application vide. La page affiche l'erreur 500 `MissingAppKeyException` et le message « No application encryption key has been specified. ». Ce message indique que la ligne `APP_KEY=` de `.env` est vide. Lancez `php artisan key:generate`, qui répond `INFO Application key set successfully.`, puis rechargez la page.

ASTUCE

Port 8000 occupé. Un **port** est un numéro qui dirige une connexion vers un programme précis de votre ordinateur. Un autre programme utilise déjà le port 8000, souvent un ancien `composer run dev` resté ouvert. Le serveur démarre alors sur le port suivant et affiche sa nouvelle adresse.

`Failed to listen on 127.0.0.1:8000 (reason: Address already in use) INFO Server running on [http://127.0.0.1:8001].`

Ouvrez l'adresse affichée, ici `http://127.0.0.1:8001`. L'adresse `localhost:8000` mène encore à l'autre programme. Arrêtez les anciens serveurs dans leurs terminaux pour libérer le port 8000.

ASTUCE

Projet cloné sans `vendor/`. Les fichiers `.gitignore` excluent `vendor/` et `.env` de l'historique Git. Ils sont donc absents d'un dépôt cloné. La première commande `php artisan` échoue alors sur `Failed opening required '…/vendor/autoload.php'`. La commande `composer install` recrée `vendor/`, tandis qu'une copie de `.env.example` fournit le nouveau `.env`.

À retenir

- Vérifiez avec `php --version` que le terminal utilise PHP 8.4.
- Créez le projet avec `laravel new premier-test-laravel --git --no-boost`, sans starter kit, puis gardez Blade, Pest et SQLite.
- Reliez chaque étape à son résultat dans `premier-test-laravel`, notamment `vendor/`, `.env`, `database.sqlite`, `node_modules/`, `public/build/` et `.git/`.
- Vérifiez les réglages avec `php artisan about` et le fonctionnement avec `php artisan test`.
- Repérez dans chaque message d'erreur la commande, le fichier ou la version qui pose problème.

Exercices

Exercice 1. Le projet `premier-test-laravel`. Créez le projet `premier-test-laravel` comme à la section 2, puis ouvrez sa page d'accueil. Lancez `php artisan about` et copiez la sortie dans un fichier `about.txt` placé sur votre bureau, hors du projet. Vérifiez que `Laravel Version` contient `13` et que `Database` contient `sqlite`.

Exercice 2. Le nom de l'application. Dans `.env`, remplacez `APP_NAME=Laravel` par `APP_NAME="Premier test"`, guillemets compris. Rechargez la page d'accueil et observez le titre de l'onglet. Retirez ensuite les guillemets, puis lancez `php artisan --version`. Notez le message obtenu et expliquez le rôle des guillemets. Remettez-les enfin.

Exercice 3. Les tests. Lancez `php artisan test` et vérifiez que les deux tests affichent `PASS`. Ouvrez `tests/Feature/ExampleTest.php`. Relevez l'adresse demandée par le test et le code de réponse attendu.

Exercice 4. À la maison, sur papier. Ouvrez `.gitignore` à la racine du projet, puis `database/.gitignore`. Parmi `vendor/`, `node_modules/`, `public/build/`, `.env` et `database/database.sqlite`, repérez les éléments absents de Git. Pour chacun, écrivez une phrase qui explique cette absence. Vérifiez si une commande peut le recréer ou s'il contient un secret propre à votre machine.

3.5 MVC

Bloc Laravel · séance 1 · ~60 min

Slides du chapitre (PDF) PDF

On peut afficher les articles d'un blog avec un seul fichier PHP de 26 lignes. Ce fichier contient alors la requête SQL, le HTML et le choix de l'ordre des articles. Une modification du tri, de la base ou de la présentation conduit donc à ouvrir le même fichier. Laravel sépare ces tâches dans plusieurs fichiers. Pour répondre à `/posts`, il exécute les instructions de chaque fichier dans un ordre précis.

À la fin de ce chapitre, vous serez capables de :

1. **nommer** le rôle des quatre parties qui traitent une requête.
2. **ranger** un extrait de code dans la partie qui remplit sa tâche.
3. **tracer** le traitement d'une requête, de l'adresse saisie jusqu'à la page affichée.
4. **dire** quels mélanges cette organisation évite, avec un exemple pour chacun.

1. Le code qui sent mauvais

Vous connaissez la requête et la réponse depuis le chapitre Framework et librairie. Le navigateur envoie une méthode et un chemin, par exemple `GET /posts.php`. Le serveur renvoie alors un code tel que `200` et une page HTML.

Le fichier `posts.php` produit la page d'accueil d'un petit blog sans utiliser de framework. Il lit trois articles dans la table `posts` de la base SQLite `database.sqlite`. Lisez le code sans le lancer.

```php
<?php
pdo = new PDO('sqlite:' . __DIR__ . '/database.sqlite');

order = (GET[’ordre’]??’’)===’ancien’?’ASC’:’DESC’;_GET['ordre'] ?? '') === 'ancien' ? 'ASC'
: 'DESC';

GET[’ordre’]??’’)===’ancien’?’ASC’:’DESC’;posts =
pdo->query("SELECTtitle,bodyFROMpostsORDERBYcreatedatpdo
    →query("SELECT title, body FROM posts ORDER BY created_at
pdo->query("SELECTtitle,bodyFROMpostsORDERBYcreatedatorder")
    →fetchAll(PDO::FETCH_ASSOC);
?>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Le blog</title>
</head>
<body>
    <h1 style="color: navy">Le blog</h1>
    <p><?= count(posts)?>articles</p><ul><?phpforeach(posts) ?> articles</p>
    <ul>
    <?php foreach (posts)?>articles</p><ul><?phpforeach(posts as post):?><li>
<strongstyle="color:navy"><?=post): ?>
        <li><strong style="color: navy"><?= post):?><li><strongstyle="color:navy"><
?=post['title'] ?></strong> : <?= $post['body'] ?></li>
    <?php endforeach; ?>
    </ul>
    <a href="?ordre=ancien">Du plus ancien au plus récent</a>
</body>
</html>
```

Dans l'adresse `?ordre=ancien`, `ordre` est un **paramètre d'URL**. Un paramètre ajoute une valeur nommée à une adresse. La ligne qui remplit `$order` lit cette valeur et choisit `ASC`. Sans cette valeur, elle choisit `DESC`.

Le serveur intégré de PHP peut recevoir les requêtes destinées à cette page. Dans le dossier qui contient `posts.php` et sa base, lancez-le dans un premier terminal :

```bash
php -S 127.0.0.1:8080
```

Dans un second terminal, la commande `curl` demande la page et affiche la réponse complète. Sous PowerShell, tapez `curl.exe` à la place de `curl`.

```bash
curl -s http://127.0.0.1:8080/posts.php
```

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Le blog</title>
</head>
<body>
    <h1 style="color: navy">Le blog</h1>
    <p>3 articles</p>
    <ul>
            <li><strong style="color: navy">Le trajet d'une requête</strong> : De l'URL à la
page HTML.</li>
            <li><strong style="color: navy">Installer Laravel 13</strong> : Composer, PHP
8.4 et un terminal.</li>
            <li><strong style="color: navy">Bienvenue sur le blog</strong> : Premier article
du blog.</li>
        </ul>
    <a href="?ordre=ancien">Du plus ancien au plus récent</a>
</body>
</html>
```

La page fonctionne, mais il faut déterminer comment la modifier sans casser une autre partie. Pendant cinq minutes, en binôme, examinez trois demandes : changer la couleur des titres, modifier l'ordre des articles et remplacer SQLite par MySQL. Pour chacune, notez le ou les fichiers à ouvrir ainsi que les lignes à modifier.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/mvc

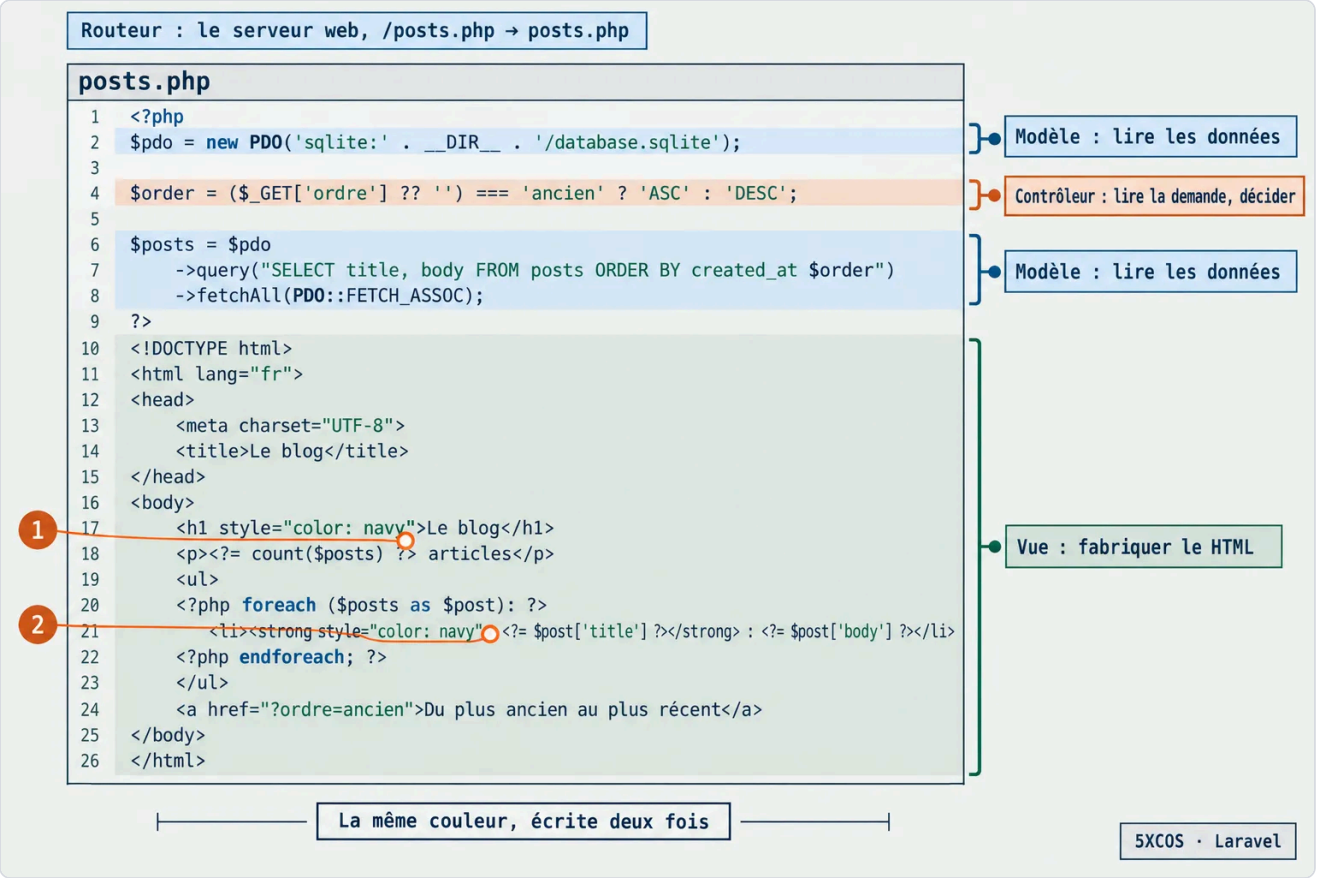
2. Séparer trois responsabilités

Une **responsabilité** est une tâche précise confiée à une partie du programme. Lorsqu'une demande change cette tâche, vous savez quelle partie modifier. `posts.php` en réunit trois : lire les données, choisir leur ordre et produire le HTML. L'organisation **MVC**, pour Modèle-Vue-Contrôleur, sépare ces tâches dans différentes parties de l'application.

- Le **modèle** lit et écrit les données, et porte les règles qui les concernent. Dans `posts.php`, ce sont les lignes 2 et 6 à 8 : la connexion à la base et la requête SQL.
- La **vue** fabrique ce que l'utilisateur voit, ici du HTML, à partir des données qu'on lui donne. Ce sont les lignes 10 à 26.
- Le **contrôleur** organise le traitement de la demande. Ici, il lit le paramètre `ordre` afin de choisir le sens du tri. Il demande ensuite les articles au modèle et transmet les articles obtenus à la vue. Dans `posts.php`, ces étapes sont mêlées au reste. Dans une application MVC, elles se trouvent dans une méthode du contrôleur.

Le **routeur** remplit un quatrième rôle : il relie une adresse à la méthode chargée de la traiter. Sans routeur applicatif, le serveur utilise ici le chemin demandé pour choisir directement un fichier. Ainsi, `/posts.php` lance le fichier `posts.php`. Cette organisation ne convient pas à une adresse comme `/posts/3`, car `3`

désigne un article et non un fichier différent. Un routeur peut associer toutes les adresses de cette forme à une même méthode, qui reçoit le numéro de l'article.



Avec ce découpage, la couleur ne concerne que la vue. Le passage à MySQL concerne l'accès aux données et sa configuration, sans modifier le HTML. Le tri concerne le contrôleur, qui choisit l'ordre, et le modèle, qui produit la requête SQL. La nature de chaque demande permet donc de repérer la partie à modifier.

- **Une modification touche le fichier de sa responsabilité.** Changer une couleur n'oblige plus à ouvrir un fichier qui contient du SQL.
- **La connexion à la base est configurée à un seul endroit.** Le modèle utilise cette configuration au lieu d'ouvrir une nouvelle connexion dans chaque page.
- **Deux personnes peuvent travailler sur des fichiers différents.** L'une modifie la vue pendant que l'autre modifie le modèle.

Le modèle s'occupe des données, la vue de l'affichage, le contrôleur coordonne les deux et le routeur choisit le contrôleur.

> NOTE
>
> En 1978 et 1979, Trygve Reenskaug, informaticien norvégien invité au centre de recherche Xerox PARC, travaille sur les interfaces graphiques du langage Smalltalk. Le 10 décembre 1979, sa note « Models-Views-Controllers » donne son nom à MVC. Les frameworks web ont repris l'idée plus tard, en l'adaptant aux requêtes et aux réponses.

3. La même page en Laravel

Dans un projet Laravel 13 neuf, la même liste peut se limiter aux articles, sans compteur, couleur ni lien de tri. Artisan, vu au chapitre Installation, permet de créer les fichiers nécessaires. La commande `make:model` crée une classe de modèle vide.

Une **migration** est un fichier qui décrit une modification de la base de données. L'option `-m` demande aussi la création d'une migration pour la table associée au modèle.

```bash
php artisan make:model Post -m
```

```
   INFO  Model [app/Models/Post.php] created successfully.

   INFO  Migration [database/migrations/2026_09_26_093317_create_posts_table.php] created successfully.
```

Sous Windows, Artisan affiche le chemin complet du fichier, `C:\laragon\www\premier-test-laravel\app\Models\Post.php`, au lieu du chemin relatif au projet. Le fichier créé est le même.

La méthode `up()` de la migration créée ajoute la table `posts`. Sa méthode `down()` peut la supprimer. Ajoutez les colonnes `title` et `body`, utilisées par le modèle, puis exécutez les migrations qui n'ont pas encore été appliquées :

```bash
php artisan migrate
```

```
   INFO  Running migrations.

  2026_09_26_093317_create_posts_table ......................... 3.11ms DONE
```

La commande `make:controller` crée un contrôleur vide :

```bash
php artisan make:controller PostController
```

```
   INFO  Controller [app/Http/Controllers/PostController.php] created successfully.
```

La table `posts` contient ensuite les trois articles utilisés dans `posts.php`. Lorsqu'une requête demande `/posts`, Laravel consulte la route, exécute le contrôleur, utilise le modèle et produit la vue.

Le routeur : `routes/web.php`

```php
<?php

use App\Http\Controllers\PostController;
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    return view('welcome');
});

Route::get('/posts', [PostController::class, 'index']);
```

La première route est créée avec le projet et affiche la page d'accueil de Laravel. L'import rend le nom court `PostController` disponible dans ce fichier. La dernière ligne peut ainsi associer `/posts` à cette classe.

- `Route::get('/posts', …)` déclare une **route**, c'est-à-dire une association entre une requête et le code qui la traite. Cette route correspond aux requêtes `GET` dont le chemin est `/posts`.
- Le second argument, `[PostController::class, 'index']`, indique quel code traitera la requête. Le premier élément donne la classe `PostController` et le second la méthode `index`. La déclaration de la route ne crée pas encore de contrôleur. Laravel créera cet objet et appellera sa méthode lorsqu'il recevra une requête `GET` pour `/posts`.

Le chemin `/posts` ne correspond donc pas au nom d'un fichier sur le disque. Sa déclaration dans `routes/web.php` l'associe à la méthode `index()` de `PostController`.

Le contrôleur : `app/Http/Controllers/PostController.php`

```php
<?php

namespace App\Http\Controllers;

use App\Models\Post;
use Illuminate\View\View;

class PostController extends Controller
{
    public function index(): View
    {
        posts=Post::latest()->get();returnview('posts.index',['posts'⇒posts =
Post::latest()→get();

        return view('posts.index', ['posts' ⇒ posts=Post::latest()
->get();returnview('posts.index',['posts'⇒posts]);
    }
}
```

- `Post::latest()->get()` demande les articles au modèle. `latest()` ajoute un tri du plus récent au plus ancien sur la colonne `created_at`. `get()` envoie la requête à la base. Il renvoie une **collection**, c'est-à-dire un objet Laravel qui regroupe ici plusieurs objets `Post`.
- `view('posts.index', ['posts' => $posts])` choisit la vue et lui transmet les données. Dans le nom `posts.index`, `posts` désigne le dossier et `index` le fichier. Laravel utilise donc `resources/views/posts/index.blade.php`. Dans le tableau associatif, la clé `'posts'` crée la variable `$posts` utilisée par la vue.

Pour afficher les articles du plus ancien au plus récent, remplacez `latest()` par `oldest()` dans le contrôleur. Celui-ci exprime l'ordre voulu sans contenir le SQL qui réalise le tri. Il demande au modèle de récupérer les articles, puis transmet la collection à la vue. Il ne produit lui-même ni SQL ni HTML.

Le modèle : `app/Models/Post.php` et sa migration

Le fichier `app/Models/Post.php` contient la classe `Post`, qui hérite de `Model`. Cette classe mère appartient à **Eloquent**, la partie de Laravel qui relie les classes PHP aux tables de la base de données.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    //
}
```

La classe est vide, telle que `make:model` l'a créée. Grâce à son héritage, `Post` dispose des méthodes définies par `Model`. Une règle de nommage d'Eloquent associe automatiquement `Post` à la table `posts`, au pluriel anglais.

La méthode `up()` de la migration décrit la création de la table :

```php
// …

return new class extends Migration
{
    public function up(): void
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('body');
            $table->timestamps();
        });
    }

    // …
};
```

La méthode `down()`, coupée ici, supprime la table lorsque la migration est annulée. Les lignes `title` et `body` ont été ajoutées à la main. `id()` crée la clé primaire. `timestamps()` crée les colonnes `created_at` et `updated_at`. La méthode `latest()` utilise `created_at` pour trier les articles.

Eloquent doit toujours envoyer une requête SQL à la base. `Post::latest()` construit cette requête à partir du modèle `Post` et de la convention sur `created_at`. `toRawSql()` permet d'afficher le SQL obtenu avant son envoi :

```php
echo Post::latest()->toRawSql();
```

```
select * from "posts" order by "created_at" desc
```

Cette requête effectue le même tri que la ligne 7 de `posts.php`. Pour utiliser MySQL, modifiez les réglages de connexion dans le fichier `.env` du projet. La route, le contrôleur, le modèle et la vue restent inchangés. Eloquent produira la requête adaptée à la connexion choisie.

La vue : `resources/views/posts/index.blade.php`

```xml
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Le blog</title>
</head>
<body>
    <h1>Le blog</h1>
    <ul>
        @foreach (postsasposts as postsaspost)
            <li><strong>{{ post->title }}</strong> : {{ post->body }}</li>
        @endforeach
    </ul>
</body>
</html>
```

L'extension `.blade.php` indique un **template Blade**, un fichier qui décrit le HTML à produire. Il contient principalement du HTML, avec des instructions comme `@foreach` et `{{ ... }}`. Lorsque Laravel prépare la réponse, Blade exécute ces instructions avec les données reçues du contrôleur et produit le HTML envoyé au navigateur.

- `@foreach (postsaspost)` et `@endforeach` entourent le HTML répété pour chaque article. Cette instruction correspond au `foreach` de PHP, sans écrire les balises `<?php ?>`.
- `{{ $post->title }}` affiche le titre après l'avoir échappé, comme au chapitre Framework et librairie : `<`, `>` et l'apostrophe deviennent des codes HTML.
- `$post->title` s'écrit avec une flèche : chaque article est un objet `Post`, et non plus un tableau comme dans `posts.php`.

Le résultat

À la racine du projet, lancez `composer run dev` dans un premier terminal, comme au chapitre Installation. Si vous utilisez Herd ou Laragon, l'adresse du projet est `http://premier-test-laravel.test` au lieu de `http://127.0.0.1:8000`. Dans un second terminal, demandez la page avec `curl` :

```bash
curl -s http://127.0.0.1:8000/posts
```

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Le blog</title>
</head>
<body>
    <h1>Le blog</h1>
    <ul>
                    <li><strong>Le trajet d&#039;une requête</strong> : De l&#039;URL à la
page HTML.</li>
                    <li><strong>Installer Laravel 13</strong> : Composer, PHP 8.4 et un
terminal.</li>
                    <li><strong>Bienvenue sur le blog</strong> : Premier article du blog.
</li>
            </ul>
</body>
</html>
```

La réponse contient les trois articles, classés du plus récent au plus ancien. Dans le premier titre, l'apostrophe apparaît sous la forme `'` dans le HTML. Cette transformation est l'échappement effectué par `{{ }}`. Le navigateur affiche une apostrophe normale, tandis que le fichier `posts.php` affichait directement les caractères reçus de la base.

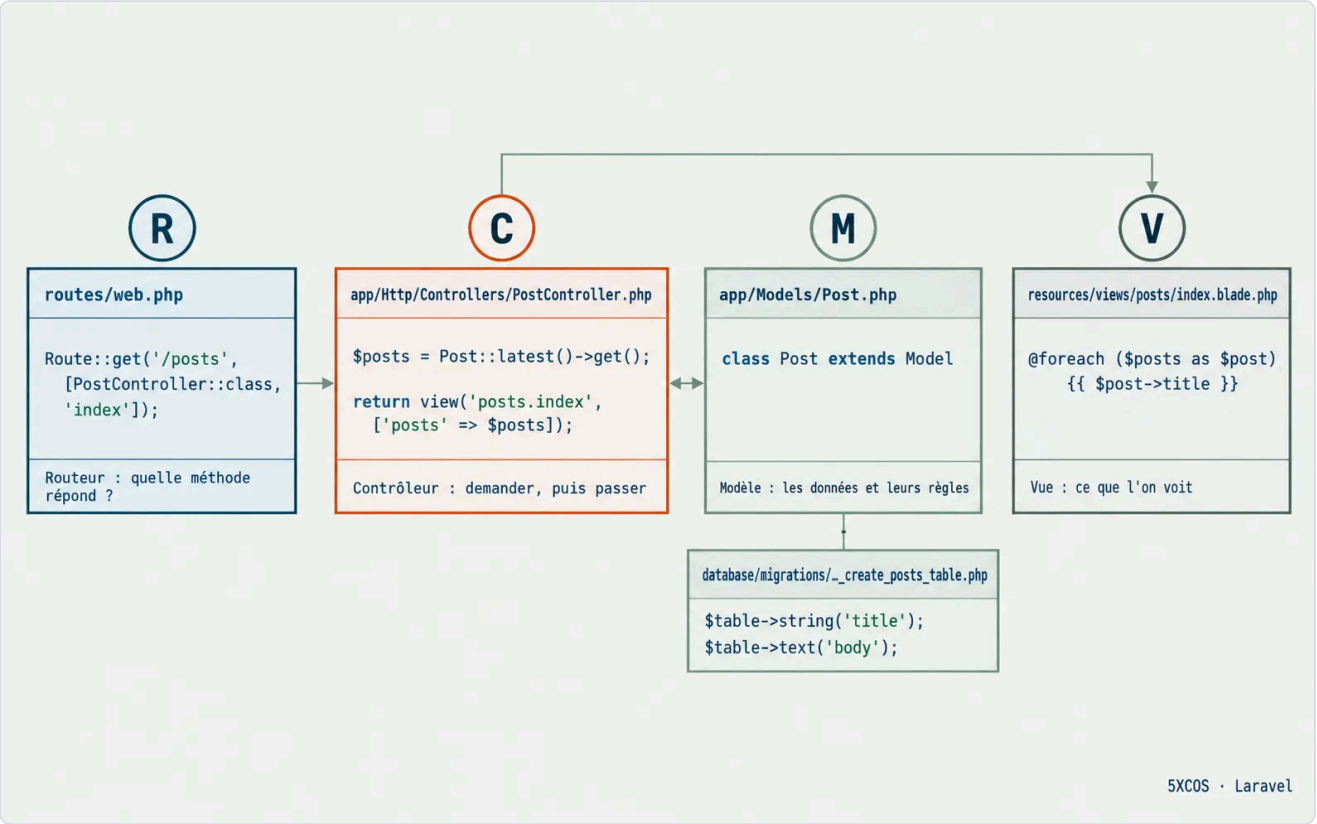
La commande `route:list` affiche toutes les routes enregistrées dans l'application :

BASH

```
php artisan route:list
```

```
  GET|HEAD  / ..................................................... routes/web.php:6
  GET|HEAD  posts ..................................... PostController@index
  GET|HEAD  storage/{path} storage.local › vendor/laravel/framework/src/Illum…
  PUT       storage/{path} storage.local.upload › vendor/laravel/framework/sr…
  GET|HEAD  up vendor/laravel/framework/src/Illuminate/Foundation/Configurati…

                                                          Showing [5] routes
```

La deuxième ligne indique que `posts` est associé à `PostController@index`, c'est-à-dire à la méthode `index()` de `PostController`. Une requête **HEAD** demande les mêmes en-têtes qu'une requête `GET`, mais sans le contenu de la réponse. Laravel l'accepte automatiquement pour cette route. Les quatre autres routes sont présentes dans le projet neuf.

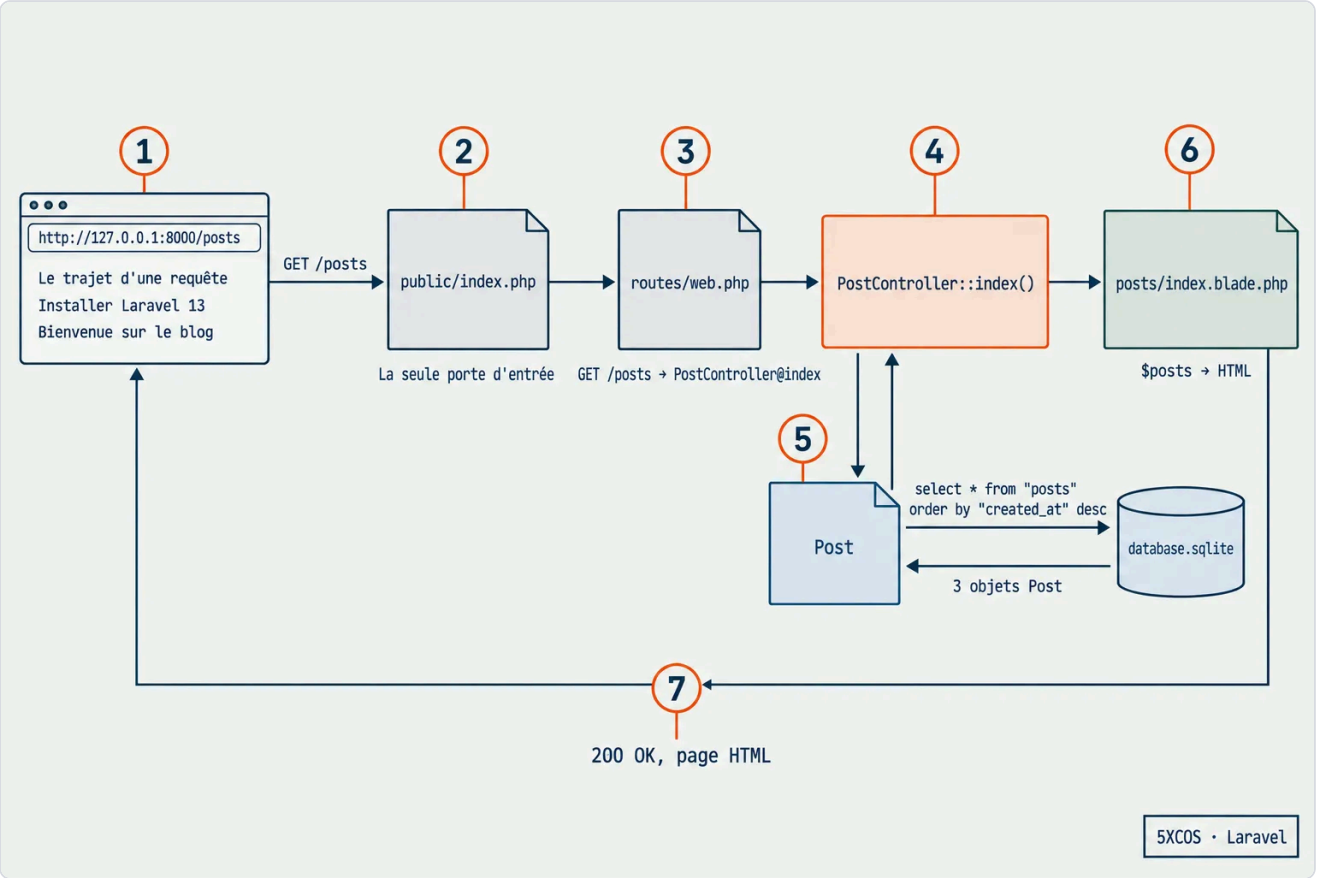


4. Le trajet d'une requête

1. Le navigateur envoie la requête `GET /posts` au serveur.
2. Le serveur transmet la requête à `public/index.php`, qui démarre Laravel.
3. Le routeur lit `routes/web.php` et trouve `PostController@index`.
4. Laravel crée un objet `PostController` et appelle `index()`.
5. `Post::latest()->get()` interroge la base et renvoie trois objets `Post`.
6. `view()` transmet les trois articles à `posts/index.blade.php`, puis Blade produit le HTML.
7. Laravel renvoie la réponse, code 200 et page HTML.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/mvc

Le fichier `public/index.php` n'apparaît dans aucune déclaration de route. Le serveur l'exécute pourtant pour chaque requête qui ne correspond pas à un autre fichier public. Son rôle est détaillé au chapitre Structure d'un projet Laravel.



Le contrôleur coordonne le traitement : il demande les articles au modèle, puis fournit `$posts` à la vue. La vue utilise uniquement les données reçues et n'exécute aucune requête vers la base. Le modèle récupère les données sans dépendre du fichier HTML qui les affichera.

5. Ce que MVC évite, et ce qu'il n'est pas

Laravel fournit des dossiers et des conventions, mais il peut aussi exécuter du code placé au mauvais endroit. Un tel code fonctionne tout en mélangeant de nouveau les responsabilités réunies dans `posts.php`.

Le premier mélange place **du SQL dans une vue**. Ce template interroge directement la base :

```xml
@php
    posts=DB::select('SELECTtitleFROMpostsORDERBYcreatedatDESC');@endphp@foreach(posts =
DB::select('SELECT title FROM posts ORDER BY created_at DESC');
@endphp
@foreach (posts=DB::select('SELECTtitleFROMpostsORDERBYcreatedatDESC');@endphp@foreach(posts
as $post)
    <li>{{ $post->title }}</li>
@endforeach
```

Ce code affiche les trois titres. Pour changer le tri, il faut cependant modifier le template. Une autre vue qui affiche les articles devrait aussi recopier la requête. Placez la lecture des articles avec le modèle, puis faites transmettre `$posts` à la vue par le contrôleur.

Le deuxième mélange place **du HTML dans un modèle**. Cette méthode produit un morceau de page :

```php
class Post extends Model
{
    public function toHtml(): string
    {
        return '<li><strong style="color: navy">' . $this->title . '</strong></li>';
    }
}
```

Changer une couleur oblige alors à modifier le modèle. Le titre n'est pas non plus échappé. Si la chaîne produite est insérée telle quelle dans la page, le navigateur exécutera un titre contenant `<script>`. Dans une vue Blade, `{{ }}` transforme ces caractères avant de produire le HTML.

Le troisième mélange place **de la logique métier dans un contrôleur de 200 lignes**. La **logique métier** regroupe les règles propres à votre application, comme « un article se lit à 200 mots par minute ». Ici, la règle est écrite dans le contrôleur :

```php
public function index(): View
{
    posts=Post::latest()->get();foreach(posts = Post::latest()->get();

    foreach (posts=Post::latest()->get();foreach(posts as post) {
        $words = str_word_count($post->body);
        $post->reading_time = max(1, (int) ceil($words / 200));
    }

    // … 190 lignes de plus : filtres, statistiques, envoi d'e-mails

    return view('posts.index', ['posts' => posts]);
}
```

La page d'un article aura aussi besoin du temps de lecture. Si le calcul reste dans `index()`, il faudra le copier dans une deuxième méthode. Une correction risque alors de n'être appliquée qu'à une seule copie. Comme le calcul dépend du texte d'un article, placez-le dans une méthode de `Post`. Le contrôleur peut ainsi demander les articles au modèle et transmettre le résultat à la vue.

La séparation MVC ne suffit pas à rendre tout le code correct. Une requête placée dans un modèle peut rester lente, et le HTML d'une vue peut rester difficile à lire. Laravel exécute les trois mélanges précédents sans erreur. Vous devez donc vérifier vous-même que chaque responsabilité se trouve dans la partie prévue.

6. Où ça mène

Chaque rôle peut traiter des cas plus complets.

- Une route peut recevoir une valeur dans son chemin, comme `/posts/{post}`, et porter un nom.

- Un contrôleur peut recevoir un formulaire, vérifier ses données et demander au navigateur d'ouvrir une autre adresse.
- Plusieurs vues Blade peuvent partager la même mise en page.
- Les modèles Eloquent peuvent être reliés, par exemple lorsqu'un article possède plusieurs commentaires.

À retenir

- Le modèle lit et écrit les données, et porte les règles qui les concernent.
- La vue fabrique le HTML à partir des données qu'elle reçoit, et `{{ }}` les échappe.
- Le contrôleur organise le traitement, demande les données au modèle et transmet le résultat à la vue.
- Le routeur relie une adresse à une méthode d'un contrôleur, dans `routes/web.php`.
- MVC sépare le code selon sa responsabilité. Laravel peut néanmoins exécuter du SQL placé dans une vue.

Exercices

1. Ranger huit extraits (sur papier, 10 minutes)

Pour chaque extrait, écrivez M pour modèle, V pour vue, C pour contrôleur ou R pour routeur. Pour choisir, repérez la responsabilité du code et le type de demande qui obligerait à le modifier.

1. `Route::get('/about', [PageController::class, 'about']);`
2. `<h2>{{ $post->title }}</h2>`
3. `$table->string('title');`
4. `return view('posts.show', ['post' => $post]);`
5. `@foreach ($comments as $comment)`
6. `class Comment extends Model`
7. `$posts = Post::latest()->get();`, dans une méthode `index()`
8. `public function readingTime(): int`, qui calcule le temps de lecture d'un article

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/mvc

2. Tracer `GET /posts/3` (sur papier, 5 minutes)

Pour l'instant, `http://127.0.0.1:8000/posts/3` répond `404 Not Found`, car aucune route ne correspond à ce chemin. Vous devez prévoir une page qui affiche uniquement l'article numéro 3.

Appliquez les sept étapes de la section 4 à `GET /posts/3`. Pour chaque fichier, indiquez s'il reste identique, s'il doit changer ou s'il faut le créer. La base contient déjà l'article 3.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/mvc

3. Les quatre fichiers dans `premier-test-laravel` (sur machine, 25 minutes, à faire avant le chapitre Structure, qui s'en sert)

Dans le projet `premier-test-laravel` créé au chapitre Installation :

1. Lancez `php artisan make:model Post -m`. Ajoutez ensuite `title` et `body` à la migration comme dans la section 3, puis lancez `php artisan migrate`.
2. Exécutez trois fois la commande ci-dessous pour ajouter trois articles. Modifiez chaque fois le titre, le texte et le nombre de jours. Utilisez 3 pour le premier article, puis 2 et 1. L'expression `now()->subDays(3)` produit la date d'il y a trois jours. Des dates différentes permettent à `latest()` de classer les articles.

```bash
php artisan tinker --execute="App\Models\Post::forceCreate(['title' => 'Bienvenue sur le blog', 'body' => 'Premier article du blog.', 'created_at' => now()->subDays(3)]);"
```

Tinker permet d'exécuter une instruction PHP dans le projet Laravel. Ici, `forceCreate()` crée un article avec les valeurs du tableau et l'enregistre dans la table `posts`.

Indice : un titre ou un texte qui contient une apostrophe, comme « Le trajet d'une requête » ou « De l'URL à la page HTML. », casse la chaîne entre apostrophes. Écrivez `d\'une` à la place de `d'une`, et `l\'URL` à la place de `l'URL`.

3. Lancez `php artisan make:controller PostController`, puis recopiez la méthode `index()`.
4. Ajoutez la route dans `routes/web.php` et importez `PostController` en haut du fichier. Créez ensuite `resources/views/posts/index.blade.php`.
5. Lancez `composer run dev`, puis vérifiez les deux résultats. La commande `php artisan route:list` doit afficher `posts` et `PostController@index`. L'adresse `http://127.0.0.1:8000/posts` doit afficher vos trois articles du plus récent au plus ancien. Avec Herd ou Laragon, utilisez `http://premier-test-laravel.test/posts`.

Si la page affiche une erreur, lisez la phrase située sous le nom de l'exception. Le message `View [posts.index] not found.` indique que Laravel ne trouve pas `resources/views/posts/index.blade.php`. Vérifiez notamment le nom du dossier `posts`. Le message `Target class [PostController] does not exist.` indique souvent que l'import manque dans `routes/web.php`. Dans ce cas, `php artisan route:list` affiche aussi `Class "PostController" does not exist`.

4. Six cartes

Que fait le modèle ?

Il fournit les méthodes qui lisent et écrivent les articles, ainsi que leurs règles. Dans le blog : `app/Models/Post.php`.

Que fait la vue ?

Elle fabrique le HTML à partir des données qu'elle reçoit. Dans le blog : `resources/views/posts/index.blade.php`.

Que fait le contrôleur ?

Il organise le traitement, demande les données au modèle et les transmet à la vue avec `view()`.

Que fait le routeur ?

Il relie une adresse à une méthode d'un contrôleur. Les routes s'écrivent dans `routes/web.php`.

Que fait `{{ $post->title }}` de plus qu'un simple `echo` ?

Il échappe le titre : `<`, `>` et l'apostrophe deviennent des codes HTML, et un `<script>` s'affiche au lieu d'être exécuté.

Les titres de la liste doivent passer en rouge. Quel fichier ouvrez-vous ?

La vue, `resources/views/posts/index.blade.php`. Ni le modèle, ni le contrôleur, ni la route ne contiennent de HTML.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/mvc

3.6 Structure d'un projet Laravel

Bloc Laravel · séance 1 · ~40 min

Slides du chapitre (PDF) PDF

Le projet `premier-test-laravel`, créé au chapitre Installation et complété au chapitre MVC, compte vingt-huit éléments à sa racine, dossiers cachés compris. Vous modifierez surtout le code de `app/`, les routes de `routes/` et les vues de `resources/views/`.

D'autres dossiers contiennent la configuration ou des fichiers créés pendant l'installation et le fonctionnement de l'application. Le modèle, le contrôleur et la vue créés à l'exercice 3 du chapitre MVC doivent être présents dans votre projet.

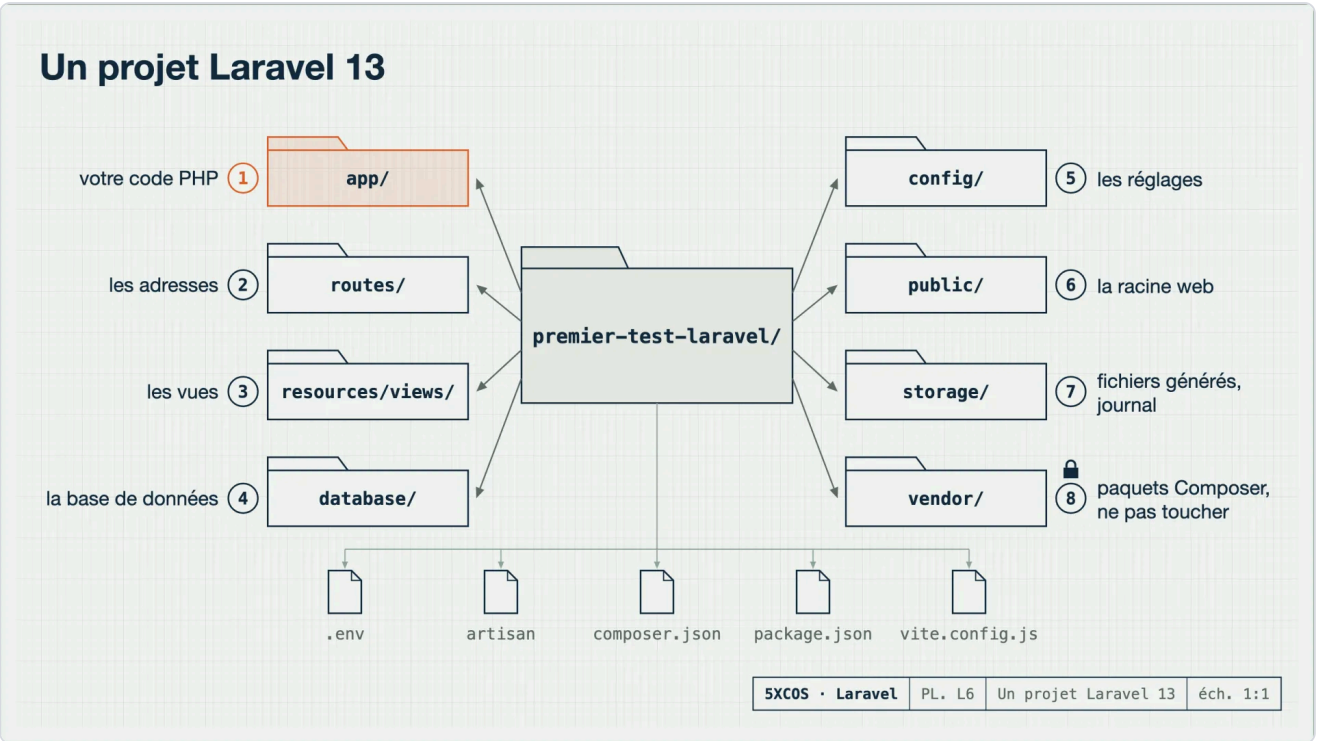
À la fin de ce chapitre, vous serez capables de :

1. **dire** à quoi sert chaque dossier de la racine d'un projet Laravel 13,
2. **retrouver** où ranger un contrôleur, un modèle, une vue ou une modification de la base, et avec quelle commande les créer,
3. **expliquer** le trajet d'une requête fichier par fichier, de `public/index.php` jusqu'à la vue,
4. **distinguer** les fichiers que vous modifiez de ceux que vous ne touchez jamais,
5. **reconnaître** `$fillable`, la liste des colonnes que Laravel peut remplir, sous ses deux écritures.

1. La racine

La **racine d'un projet** est le dossier qui contient tous ses fichiers et ses sous-dossiers. La commande `laravel new` a créé celle de `premier-test-laravel` en septembre 2026. Elle a utilisé **Pest**, le lanceur de tests, et **npm**, l'outil qui installe les paquets JavaScript.

```
premier-test-laravel/
├── .editorconfig      réglages de l'éditeur
├── .git/              historique git
├── .env               vos réglages, hors git
├── .env.example       modèle de .env, versionné
├── .gitattributes     réglages git
├── .gitignore         ce que git ignore
├── .npmrc             réglages npm
├── AGENTS.md          consignes pour agents IA
├── CLAUDE.md          même texte
├── README.md          présentation
├── app/               votre code PHP
├── artisan            console de Laravel
├── bootstrap/         démarrage
├── composer.json      dépendances PHP
├── composer.lock      versions exactes
├── config/            configuration
├── database/          la base de données
├── node_modules/      paquets npm
├── package-lock.json  versions exactes npm
├── package.json       dépendances JavaScript
├── phpunit.xml        configuration des tests
├── public/            ce que voit le navigateur
├── resources/         vues, CSS, JavaScript
├── routes/            les routes
├── storage/           fichiers générés
├── tests/             les tests Pest
├── vendor/            paquets Composer
└── vite.config.js     configuration de Vite
```



2. Les fichiers de la racine

`.env` et `.env.example`

Une **variable d'environnement** contient un réglage propre à la machine sur laquelle l'application fonctionne. Votre ordinateur peut utiliser SQLite tandis que le serveur utilise MySQL. Au démarrage, Laravel lit ces valeurs dans `.env`. Vous pouvez ainsi changer la base, l'adresse du site ou un mot de passe sans modifier les classes de l'application.

Le fichier `.env.example` contient les noms des variables nécessaires au projet et des valeurs utilisables sans secret.

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=
APP_DEBUG=true
APP_URL=http://localhost:8000
…
DB_CONNECTION=sqlite
…
SESSION_DRIVER=database
QUEUE_CONNECTION=database
CACHE_STORE=database
…
MAIL_MAILER=log
```

Dans cet exemple, `APP_KEY` est donc vide. Pendant l'installation, la commande a copié `.env.example` vers `.env`, puis elle a créé une clé propre au projet. Sur un serveur de production, `.env` contient les mots de passe de la base et des services. Le fichier `.gitignore` empêche Git d'enregistrer trois fichiers qui pourraient contenir ces secrets.

```
.env
.env.backup
.env.production
```

N'enregistrez jamais `.env` dans Git. Quand votre application a besoin d'une nouvelle variable, ajoutez son nom dans `.env.example` sans y placer de valeur secrète.

`artisan`

`artisan` est le fichier qui lance **Artisan**, l'outil en ligne de commande vu au chapitre Installation. Ses commandes peuvent créer un fichier, lancer un test ou afficher des informations sur le projet. La commande `php artisan about` affiche notamment les versions installées et les principaux réglages actifs.

```
  Environment ..........................................................
  Application Name ..................................................... Laravel
  Laravel Version ...................................................... 13.33.0
  PHP Version ........................................................... 8.4.21
  Composer Version ...................................................... 2.8.6
  Environment ........................................................... local
  Debug Mode ............................................................ ENABLED
  URL ................................................... localhost:8000
  …
```

Chez vous, la ligne `Application Name` affiche `Premier test`, le nom choisi à l'exercice 2 du chapitre Installation. Sous Herd ou Laragon, la ligne `URL` affiche `premier-test-laravel.test`.

`composer.json`

Le fichier `composer.json` indique les paquets PHP nécessaires au projet et la manière de charger vos classes. Dans `premier-test-laravel`, ses parties `require` et `autoload` contiennent ces lignes.

```json
"require": {
    "php": "^8.3",
    "laravel/framework": "^13.17",
    "laravel/tinker": "^3.0"
},
…
"autoload": {
    "psr-4": {
        "App\\": "app/",
        …
    }
},
```

Une **dépendance** est un paquet dont le projet a besoin pour fonctionner. Laravel apparaît donc dans la partie `require` de `composer.json`.

Un **espace de noms** est le début du nom complet d'une classe, comme `App\Models` dans `App\Models\User`. La **règle PSR-4** relie cet espace de noms au dossier qui contient le fichier de la classe. La ligne `"App\\": "app/"` permet ainsi de retrouver `App\Models\User` dans `app/Models/User.php`.

La partie `scripts` de `composer.json` associe des noms courts à des commandes. Vous lancez directement `composer run setup` pour préparer un projet copié depuis un dépôt Git. `composer run dev` démarre le serveur et `composer run test` lance les tests. D'autres noms commencent par `pre-` ou `post-`. Composer exécute ces scripts avant ou après une opération précise. Par exemple, `post-install-cmd` est exécuté après `composer install`.

`package.json` et `vite.config.js`

Le fichier `package.json` liste les paquets JavaScript du projet, dont **Vite**, l'outil qui prépare les fichiers chargés par le navigateur. La version installée est Vite 8. Le fichier `vite.config.js` lui indique de partir de `resources/css/app.css` et de `resources/js/app.js`. Vite lit aussi les fichiers auxquels ces deux sources font référence. La commande `npm run build` écrit le CSS et le JavaScript obtenus dans `public/build/`.

Le même fichier liste Tailwind CSS 4 et ses extensions.

Les autres fichiers

Le fichier `phpunit.xml` contient les réglages utilisés pendant les tests. `AGENTS.md` et `CLAUDE.md` contiennent le même texte pour les agents IA de programmation. Ce texte leur demande d'installer Laravel Boost, un paquet qui fournit à ces agents des informations sur le projet et les conventions de Laravel.

3. Les dossiers

`app/` : votre code

Dans un projet neuf, `app/` contient trois fichiers : `Http/Controllers/Controller.php`, `Models/User.php` et `Providers/AppServiceProvider.php`. Votre projet `premier-test-laravel` contient aussi `PostController.php` et `Post.php`, créés au chapitre MVC.

`Controller.php` contient la classe abstraite vide vue au chapitre Framework et librairie. Votre `PostController`, créé au chapitre MVC, en hérite. Le dossier `Models/` contient une classe par table, comme `Post` pour la table `posts`.

Le dossier `Providers/` contient les **fournisseurs de services**. Laravel exécute ces classes pendant son démarrage pour enregistrer ou configurer une classe utilisée ailleurs. Un fournisseur peut, par exemple, indiquer quelle classe doit envoyer les courriels de l'application.

Un projet neuf ne contient pas tous les sous-dossiers possibles de `app/`. Lorsqu'une commande Artisan crée un premier fichier d'un certain type, elle crée aussi le dossier prévu pour ce type s'il n'existe pas.

```bash
php artisan make:event DemoEvent
```

```
   INFO  Event [app/Events/DemoEvent.php] created successfully.
```

Avant cette commande, `app/` contient `Http`, `Models` et `Providers`. `make:event` crée le dossier `app/Events/` et le fichier `DemoEvent.php`. Supprimez ensuite `app/Events/`, puisque ce fichier sert uniquement à cet essai. Les commandes `make:job`, `make:mail` et `make:policy` créent de la même manière les dossiers prévus pour leurs fichiers.

Vous pouvez aussi créer `app/Services/` vous-même. Le **chargement automatique** permet à Composer de trouver une classe à partir de son nom complet. Composer trouvera `App\Services\Mailer` si la classe se trouve dans `app/Services/Mailer.php` et déclare l'espace de noms `App\Services`.

`routes/`

Dans un projet neuf, le dossier `routes/` contient seulement `web.php` et `console.php`. Dans `web.php`, chaque route associe une adresse et une méthode HTTP au code qui doit traiter la requête.

```php
Route::get('/', function () {
    return view('welcome');
});
```

Le fichier `console.php` permet de définir de courtes commandes Artisan, comme `inspire`. Le fichier `api.php` n'existe pas dans un projet neuf. La commande `php artisan install:api` le crée si l'application en a besoin.

La commande `php artisan route:list` affiche l'adresse, la méthode HTTP et le code associé à chaque route.

```
  GET|HEAD  / ..................................................... routes/web.php:6
  GET|HEAD  posts ....................................... PostController@index
  GET|HEAD  storage/{path} storage.local › vendor/laravel/framework/src/Illum…
  PUT       storage/{path} storage.local.upload › vendor/laravel/framework/sr…
  GET|HEAD  up vendor/laravel/framework/src/Illuminate/Foundation/Configurati…

                                                                Showing [5] routes
```

La route `posts` a été ajoutée au chapitre MVC. Laravel enregistre lui-même les deux routes `storage/{path}` : elles servent les fichiers du disque `local`, le dossier `storage/app/private`, à condition que l'adresse porte une signature que Laravel a fabriquée. Vous n'avez pas besoin de les modifier, et vous ne les utiliserez pas dans ce cours.

`resources/`

Le dossier `views/` contient les vues Blade dont le nom se termine par `.blade.php`. Les dossiers `css/` et `js/` contiennent les fichiers que Vite lit avant de produire ceux de `public/build/`.

Le nom passé à `view()` part de `resources/views/` et omet l'extension `.blade.php`. Dans `view('posts.index')`, `posts` désigne le sous-dossier et `index` désigne le fichier. Laravel ouvre donc `resources/views/posts/index.blade.php`, la vue créée au chapitre MVC.

Les fichiers de traduction ne se trouvent pas dans `resources/`. La commande `php artisan lang:publish` crée le dossier `lang/` à la racine pour les recevoir.

`database/`

Une **migration** est un fichier PHP qui décrit une modification de la structure de la base. Le dossier `migrations/` contient les trois fichiers exécutés par l'installeur pour créer `users`, `cache` et `jobs`. Il contient aussi la migration de `posts`, ajoutée au chapitre MVC.

Une **fabrique** crée des objets remplis de données fictives pour les tests. Celle des utilisateurs se trouve dans `factories/UserFactory.php`. Un **seeder** ajoute des données de départ dans la base. Le fichier principal est `seeders/DatabaseSeeder.php`.

Le fichier `database.sqlite` contient la base SQLite elle-même. La règle inscrite dans `database/.gitignore` empêche Git de l'enregistrer.

`config/`

Dix fichiers, un par sujet, de `app.php` à `session.php`. Extrait de `config/app.php` :

```php
'name' ⇒ env('APP_NAME', 'Laravel'),
…
'env' ⇒ env('APP_ENV', 'production'),
```

Les fichiers de `config/` lisent `.env` avec `env()`. Dans `env('APP_NAME', 'Laravel')`, la seconde valeur est utilisée lorsque `APP_NAME` n'en contient aucune. Les contrôleurs et les autres classes lisent ensuite ce réglage avec `config('app.name')`.

Un **cache de configuration** est un fichier qui contient des réglages déjà rassemblés. Sur un serveur de production, la commande `php artisan config:cache` écrit ces valeurs dans `bootstrap/cache/config.php` pour accélérer le démarrage. Au démarrage suivant, Laravel charge ce fichier au lieu de relire `.env` et les fichiers de `config/`. Un appel direct à `env('APP_NAME')` dans un contrôleur renvoie alors `null`, tandis que `config('app.name')` lit la valeur préparée. **Utilisez `env()` dans `config/` et `config()` dans les autres classes.**

`public/` : la racine web

La **racine web** est le dossier dans lequel le serveur cherche les fichiers demandés par le navigateur. Pour ce projet, Herd et Laragon utilisent `premier-test-laravel/public` à l'adresse `premier-test-laravel.test`. Le navigateur peut donc recevoir `favicon.ico`, `robots.txt` et les fichiers de `build/`. Il ne peut pas demander directement `.env` ni les classes de `app/`, qui se trouvent en dehors de `public/`.

Le fichier `public/index.php` reçoit les requêtes qui ne visent pas un fichier présent dans `public/`. Le fichier `.htaccess` donne cette règle au serveur Apache.

`bootstrap/`

Le fichier `app.php` prépare Laravel et lui indique notamment où trouver les routes. Le fichier `providers.php` donne la liste des fournisseurs de services à exécuter pendant ce démarrage.

```php
return [
    AppServiceProvider::class,
];
```

La commande `php artisan config:cache`, par exemple, écrit dans `bootstrap/cache/` le fichier que Laravel relira au prochain démarrage.

`storage/`, `tests/`, `vendor/`

Le dossier `storage/` reçoit les fichiers que Laravel produit pendant le fonctionnement. Les fichiers conservés par l'application vont dans `storage/app/`.

Laravel place ses fichiers de travail dans `storage/framework/`. Vous y trouvez les caches, les données conservées entre deux requêtes d'un même navigateur et les vues déjà transformées en PHP.

Un **journal** est un fichier dans lequel Laravel écrit ses erreurs et ses messages. Le journal principal se trouve dans `storage/logs/laravel.log`. La commande `composer run dev` lance **Pail**, l'outil qui lit ce journal et affiche les nouvelles lignes dans le terminal. Pail crée aussi `storage/pail/`.

Le dossier `tests/` contient `Feature/`, `Unit/`, `TestCase.php` et `Pest.php`. Un **test unitaire** vérifie une classe seule et se place dans `Unit/`. Un **test de fonctionnalité** vérifie plusieurs parties de l'application ensemble, par exemple en envoyant une requête HTTP. La commande `make:test` crée ce second type

de test dans `Feature/` par défaut.

Le dossier `vendor/` contient les paquets PHP installés par Composer. La commande `composer install` peut le recréer à partir de `composer.lock`.

Où ranger quoi

Je veux ajouter…	`php artisan …`	Fichier créé
un contrôleur	`make:controller CommentController`	`app/Http/Controllers/CommentController.php`
un modèle	`make:model Comment`	`app/Models/Comment.php`
une migration	`make:migration create_comments_table`	`database/migrations/2026_09_26_140158_create_comments_table.php`
une vue	`make:view comments.index`	`resources/views/comments/index.blade.php`
un test	`make:test CommentTest`	`tests/Feature/CommentTest.php`
une commande Artisan	`make:command PublishPosts`	`app/Console/Commands/PublishPosts.php`

Artisan place au début du nom de la migration la date et l'heure auxquelles vous lancez la commande.

4. Le trajet d'une requête, fichier par fichier

Quand le navigateur demande `http://premier-test-laravel.test/`, le serveur cherche d'abord un fichier correspondant dans la racine web. Comme cette adresse ne désigne aucun fichier public, le serveur exécute `public/index.php`.

```php
…
// Register the Composer autoloader...
require __DIR__.'/../vendor/autoload.php';

// Bootstrap Laravel and handle the request...
/** @var Application app*/app */
app*/app = require_once __DIR__.'/../bootstrap/app.php';

$app→handleRequest(Request::capture());
```

La première instruction charge `vendor/autoload.php`. Quand PHP rencontre un nom de classe, le chargeur de Composer utilise la règle PSR-4 pour trouver et ouvrir son fichier. La deuxième instruction ouvre `bootstrap/app.php`, qui crée l'objet représentant l'application.

```php
return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
    )
    …
```

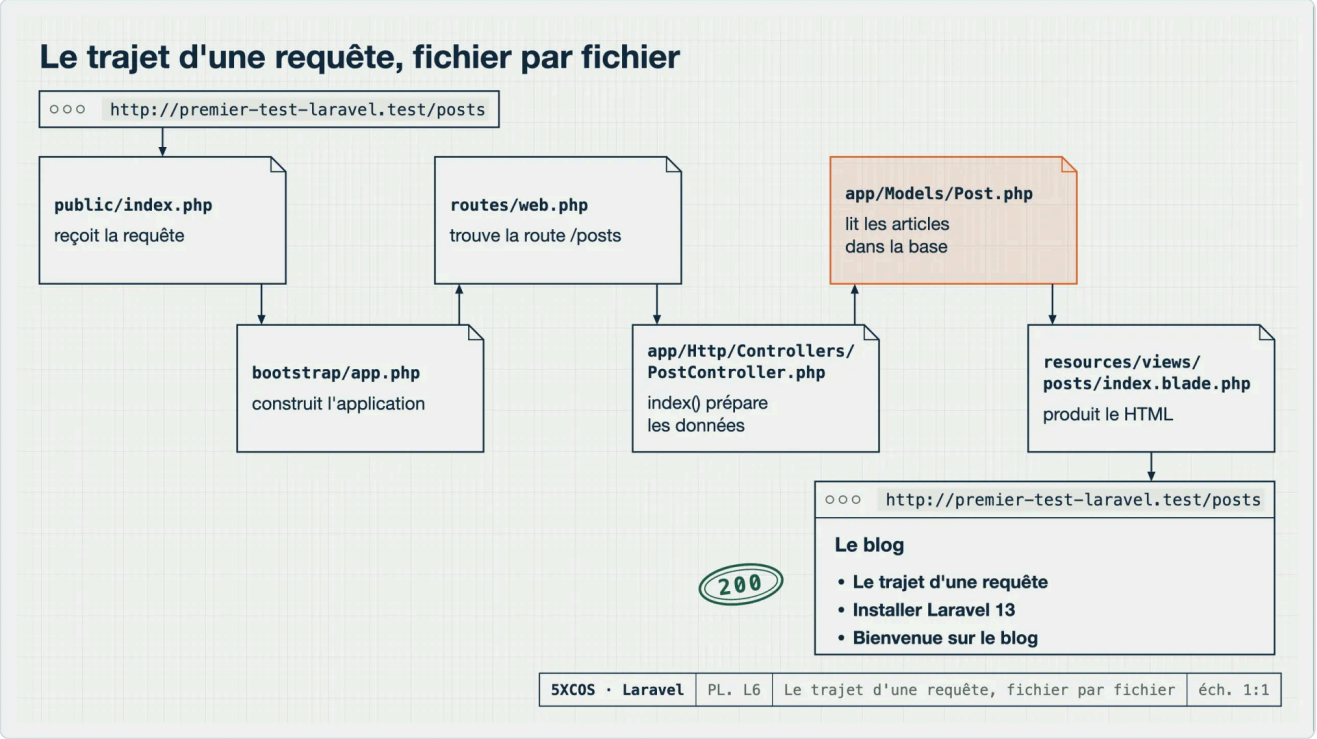
L'appel à `withRouting` donne à Laravel le chemin de `routes/web.php` et celui de `routes/console.php`. La ligne `health: '/up'` enregistre aussi la route `/up` affichée par `php artisan route:list`.

Dans `public/index.php`, `Request::capture()` rassemble l'adresse, la méthode HTTP et les données reçues. La méthode `handleRequest()` reçoit cet objet et compare ensuite l'adresse `/` aux routes de `routes/web.php`. La route correspondante exécute `view('welcome')`. Laravel ouvre alors `resources/views/welcome.blade.php` et utilise son contenu pour produire la réponse HTML.

Pour afficher `/posts`, le serveur commence aussi par `public/index.php`. Ce fichier active le chargement automatique, puis ouvre `bootstrap/app.php`. Laravel lit alors `routes/web.php` et trouve `PostController::index()`.

La méthode `index()` demande les articles à la classe `Post`, puis choisit la vue avec `view('posts.index')`. Laravel ouvre donc `resources/views/posts/index.blade.php`. Blade parcourt les articles reçus dans `$posts` et produit le HTML envoyé au navigateur.

Le fichier `artisan` ouvre lui aussi `bootstrap/app.php`. Il transmet toutefois une commande à `handleCommand` au lieu de transmettre une requête HTTP à `handleRequest`.



5. Ce qu'on modifie, ce qu'on ne touche jamais

Vous écrivez dans	Vous ne touchez jamais à
`app/`, `routes/`, `resources/`, `database/`, `tests/`	`vendor/`, recréé par `composer install`
`config/`	`node_modules/`, recréé par `npm install`
`.env` et `.env.example`	`bootstrap/cache/`, généré par Laravel
`composer.json` via `composer require`	`storage/framework/`, généré par Laravel
	`public/build/`, produit par `npm run build`

`composer install` recrée `vendor/`, tandis que `npm install` recrée `node_modules/`. Laravel recrée `bootstrap/cache/` et `storage/framework/` quand il en a besoin. La commande `npm run build` remplace le contenu de `public/build/`. Une modification manuelle dans l'un de ces dossiers disparaît donc à la génération suivante.

6. Deux écritures de `$fillable`

Le modèle `Post` lu au chapitre 0 de la POO déclarait `protected $fillable = ['title', 'body'];`. Celui que vous avez créé au chapitre MVC ne déclare encore rien. Dans Laravel 13, le fichier `User.php` utilise une autre écriture.

```php
#[Fillable(['name', 'email', 'password'])]
#[Hidden(['password', 'remember_token'])]
class User extends Authenticatable
```

`#[Fillable(...)]` est un **attribut PHP**, une information attachée à une classe que Laravel peut lire. Ici, cet attribut remplace la propriété `protected $fillable = [...]`. Les deux écritures indiquent les colonnes que Laravel peut remplir à partir d'un ensemble de données, comme celles d'un formulaire validé.

Le modèle `Post` créé au chapitre MVC ne contient encore ni cette propriété ni cet attribut. L'exercice utilise donc `forceCreate()`. Cette méthode remplit le nouvel article et l'enregistre sans consulter la liste des colonnes autorisées.

À retenir

- Vous placez le code PHP dans `app/`, les routes dans `routes/` et les vues dans `resources/views/`.
- Le fichier `.env` contient les réglages de votre machine et ne doit jamais être enregistré dans Git.
- Le dossier `public/` est la racine web, et `public/index.php` reçoit les requêtes qui ne visent aucun fichier public.
- Une commande `make:` crée le fichier demandé et son dossier si celui-ci n'existe pas encore.
- Les outils recréent `vendor/`, `node_modules/`, `bootstrap/cache/`, `storage/framework/` et `public/build/`. Vous ne modifiez donc pas leur contenu.

Exercices

1. Chasse au trésor

Dans `premier-test-laravel`, trouvez chaque fichier. Affichez ensuite la réponse de la carte pour vérifier son chemin.

Où la route `/` est-elle définie ?

`routes/web.php`

Quelle vue la page d'accueil affiche-t-elle ?

`resources/views/welcome.blade.php`

Quel fichier reçoit une requête qui ne vise pas un fichier public ?

`public/index.php`

Où changer le nom de l'application sur votre machine ?

`.env`, variable `APP_NAME`

Où la table `users` est-elle créée ?

`database/migrations/0001_01_01_000000_create_users_table.php`

Où est la classe mère des contrôleurs ?

`app/Http/Controllers/Controller.php`

Où sont listés les fournisseurs de services ?

`bootstrap/providers.php`

Où Laravel écrit-il ses erreurs ?

`storage/logs/laravel.log`

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/structure-dun-projet-laravel

2. Les générateurs

Dans `premier-test-laravel`, lancez `php artisan list make`, puis comptez les commandes dont le nom commence par `make:`. Lancez ensuite `php artisan make:job SendNewsletter` et `php artisan make:enum Status`. Regardez dans `app/` quel fichier chaque commande a créé et si un nouveau dossier est apparu. Supprimez enfin ces deux fichiers et le dossier devenu vide.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/structure-dun-projet-laravel

3. Sur papier : où ranger ces fichiers ?

Écrivez le chemin complet de chaque fichier dans `premier-test-laravel`. Les étiquettes, ou *tags*, servent à classer les articles.

1. `TagController.php`
2. le modèle `Tag.php`
3. `2026_10_05_120000_create_tags_table.php`
4. la vue appelée par `view('tags.index')`
5. `TagTest.php`, un test de fonctionnalité
6. `TagSeeder.php`

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/structure-dun-projet-laravel

3.7 Routing

Bloc Laravel · séance 2 · ~60 min

Slides du chapitre (PDF) PDF

Au chapitre MVC, vous avez écrit une seule route : quand le navigateur demande `/posts` avec une requête `GET`, Laravel appelle la méthode `index()` de `PostController`. Un vrai site a besoin de bien plus de routes. Pour afficher la fiche d'un livre, l'adresse contient son numéro, comme `/livres/12`. Pour enregistrer un formulaire, le navigateur envoie une requête `POST` plutôt que `GET`. Toutes ces routes se déclarent dans `routes/web.php`. À chaque requête, Laravel parcourt ces routes de haut en bas et garde la première qui correspond à l'adresse et au verbe demandés.

À la fin de ce chapitre, vous serez capables de :

1. **écrire** une route qui reçoit un ou plusieurs paramètres, obligatoires ou optionnels, et limiter les valeurs acceptées,
2. **prédire** quelle route répond à une requête, ou pourquoi Laravel répond 404 ou 405,
3. **corriger** deux routes déclarées dans le mauvais ordre, de deux façons,

4. **choisir** entre une fonction anonyme, un contrôleur, `Route::view` et `Route::redirect`,
5. **nommer** une route et produire son adresse avec `route()`.

1. La route du chapitre MVC

Voici cette route telle qu'elle figure dans `routes/web.php`, dans votre projet `premier-test-laravel` :

```php
Route::get('/posts', [PostController::class, 'index']);
```

La commande `php artisan route:list` affiche toutes les routes que Laravel connaît. Au chapitre MVC, cette liste contenait aussi des routes que vous n'aviez pas écrites, comme `storage/{path}` et `up`. Le code qui leur répond se trouve dans le dossier `vendor/`. La route `up`, par exemple, est déclarée dans votre projet par la ligne `health: '/up'` de `bootstrap/app.php`, vue au chapitre Structure d'un projet Laravel, mais c'est une classe de Laravel, dans `vendor/`, qui lui répond. L'option `--except-vendor` retire de la liste les routes dont le code est dans `vendor/`. Il ne reste que celles dont le code est dans vos fichiers.

```bash
php artisan route:list --except-vendor
```

```
  GET|HEAD       / ..........................................................
routes/web.php:6
  GET|HEAD       posts ......................................................
PostController@index

                                                                 Showing [2]
routes
```

Chaque ligne se lit de gauche à droite. Elle donne d'abord la méthode HTTP, qu'on appelle aussi le **verbe**, le mot employé dans ce chapitre. Viennent ensuite le chemin, écrit sans la barre oblique du début, puis l'endroit où se trouve le code qui répond. La route `/` est une fonction écrite directement dans le fichier des routes. La liste indique donc le fichier et la ligne où sa déclaration commence, `routes/web.php:6`. La route `posts` appelle un contrôleur, et la liste indique la classe et la méthode, `PostController@index`.

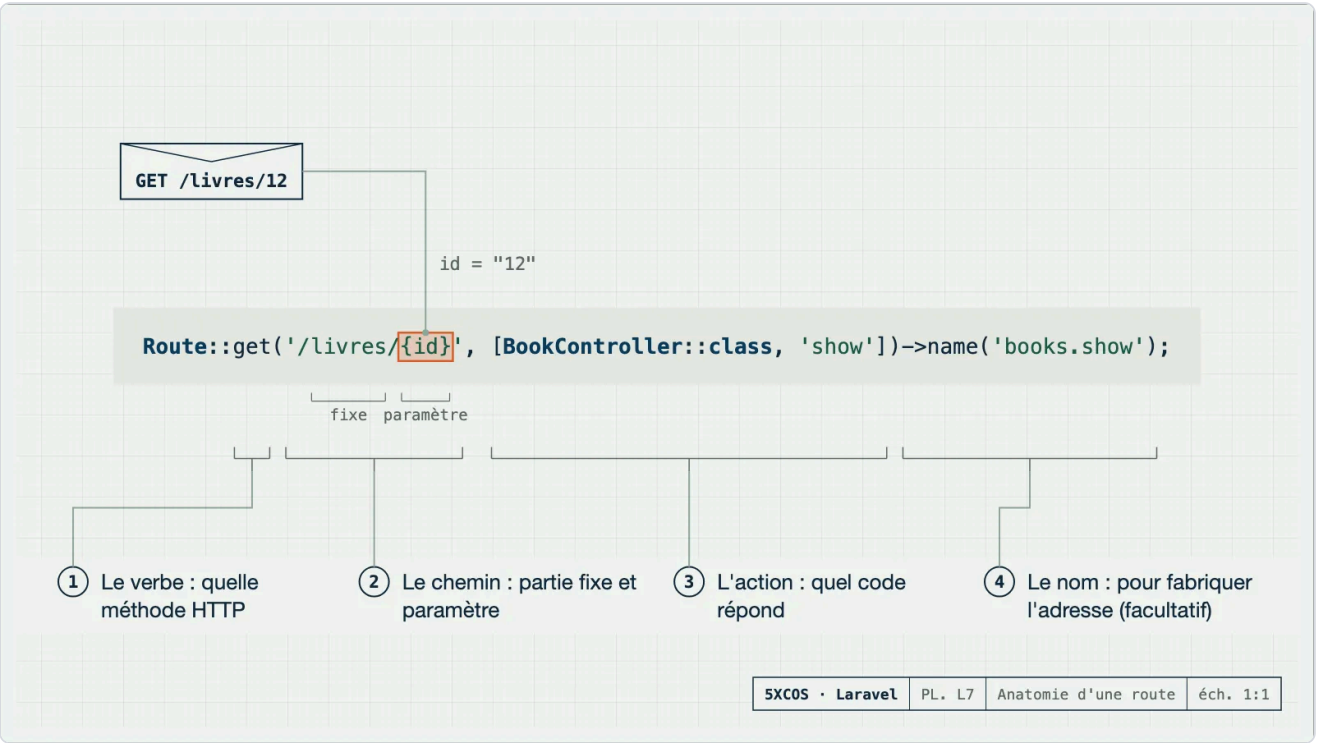
2. Une route, quatre morceaux

Tout au long de ce chapitre, vous complétez le même fichier `routes/web.php`. Les deux routes du projet, `/` et `/posts`, restent en tête du fichier, et chaque route que vous écrivez s'ajoute à la suite des précédentes. Quand une route déjà écrite reçoit une contrainte ou un nom, le texte vous demande de la remplacer. Les numéros de ligne affichés par `route:list` correspondent au fichier obtenu en suivant cet ordre.

Voici une route qui réunit les quatre éléments qu'une déclaration peut contenir. Ne la recopiez pas : la classe `BookController` n'existe pas dans votre projet, vous l'écrirez au chapitre Controllers.

```php
Route::get('/livres/{id}', [BookController::class, 'show'])->name('books.show');
```

- **Le verbe** : le nom de la méthode appelée sur `Route` indique le verbe HTTP que la route accepte. `Route::get` répond aux requêtes `GET`, `Route::post` aux requêtes `POST`.
- **Le chemin** : `'/livres/{id}'` contient une partie fixe, `livres`, et une partie entre accolades, `{id}`, qui accepte n'importe quelle valeur.
- **L'action** : le code que Laravel exécute quand l'adresse demandée correspond à cette route. Ici, c'est la méthode `show()` de `BookController`.
- **Le nom** : `->name('books.show')` donne à la route un nom que vous utiliserez dans votre code pour obtenir son adresse, au lieu de la recopier à la main. Le nom est facultatif, et la section 8 montre à quoi il sert.



Une fonction anonyme comme action

Une **fonction anonyme** est une fonction sans nom, écrite directement à l'endroit où elle sert. Vous en avez lu une au chapitre Lire du code Laravel du bloc POO, avec `function (Blueprint $table)`. Quand le second argument de `Route::get` est une fonction anonyme, Laravel l'exécute chaque fois qu'une requête correspond à la route, puis envoie au navigateur la valeur qu'elle a renvoyée. Ajoutez cette route à la fin de `routes/web.php`, sous la route `/posts` :

```php
Route::get('/ping', function () {
    return 'pong';
});
```

Demandez ensuite la page avec `curl`. Sous PowerShell, tapez `curl.exe` à la place de `curl`.

```bash
curl -s http://premier-test-laravel.test/ping
```

```
pong
```

Laravel place la chaîne renvoyée par la fonction dans le contenu de la réponse, ce que le navigateur affiche, et y joint le code 200, qui signifie que la requête a abouti. La route `/` créée avec le projet a la même forme : sa fonction anonyme renvoie `view('welcome')`.

Tout écrire dans `routes/web.php` ?

Vous pourriez aussi écrire la route `/posts` du chapitre MVC avec une fonction anonyme. Il suffit de copier le contenu de la méthode `index()` dans la fonction. Comme la route n'utilise plus `PostController` mais le modèle `Post`, la ligne `use` en haut du fichier importe `Post` à la place :

```php
use App\Models\Post;
use Illuminate\Support\Facades\Route;

// …

Route::get('/posts', function () {
    posts=Post::latest()->get();returnview('posts.index',['posts'⇒posts = Post::latest()
→get();

    return view('posts.index', ['posts' ⇒ posts=Post::latest()
->get();returnview('posts.index',['posts'⇒posts]);
});
```

Le navigateur reçoit exactement la même page qu'au chapitre MVC, au caractère près. La seule différence apparaît dans `php artisan route:list --except-vendor` : la route `posts` n'affiche plus un contrôleur, mais la ligne de `routes/web.php` où la fonction est écrite.

```
  GET|HEAD       / ..........................................................
routes/web.php:6
  GET|HEAD       ping .......................................................
routes/web.php:16
  GET|HEAD       posts ......................................................
routes/web.php:10

                                                                  Showing [3]
routes
```

Si vous avez fait l'essai, remettez ensuite la route `/posts` et l'import de `PostController` du chapitre MVC. Les deux écritures fonctionnent. Dans ce cours, vous relierez pourtant `/posts` à `PostController`, pour trois raisons :

- `routes/web.php` **se lit comme une liste.** Quand chaque route appelle un contrôleur, elle tient sur une ligne qui donne le verbe, le chemin et la méthode à appeler. Si chacune de trente routes contenait dix lignes de code, il faudrait parcourir trois cents lignes pour retrouver les adresses du site.
- **Chaque fichier garde son rôle, comme au chapitre MVC.** C'est le contrôleur qui demande les articles au modèle et choisit la vue. Le fichier des routes se contente d'indiquer quel contrôleur appeler pour chaque adresse.
- `route:list` **vous dit quel fichier ouvrir.** Avec un contrôleur, la liste affiche `PostController@index`, et vous savez tout de suite quelle méthode modifier. Avec une fonction anonyme, elle affiche seulement `routes/web.php:10`, un numéro de ligne qui change dès que vous ajoutez une route plus haut dans le fichier.

Dans ce cours, une route qui interroge un modèle ou prépare une vue avec des données appelle un contrôleur. Les exemples et les exercices de ce chapitre renvoient seulement une phrase, sans modèle ni vue. Une fonction anonyme suffit donc, et vous passerez aux contrôleurs au chapitre Controllers.

3. Comment Laravel choisit une route

Ajoutez ces quatre routes à la fin de `routes/web.php`, sous la route `/ping` :

```php
Route::get('/livres', function () {
    return 'Liste des livres';
});

Route::post('/livres', function () {
    return 'Livre enregistré';
});

Route::get('/livres/{id}', function (string id) {
    return "Fiche du livre $id";
});

Route::get('/auteurs/{name}', function (string name) {
    return "Livres de $name";
});
```

Pour choisir une route, Laravel découpe d'abord l'adresse en **segments**, les morceaux compris entre deux barres obliques. `/livres/12` compte ainsi deux segments, `livres` et `12`.

Laravel reprend ensuite les routes de `routes/web.php` une à une, de haut en bas, et se pose deux questions pour chacune. Le chemin de la route a-t-il autant de segments que l'adresse, avec les mêmes mots aux places fixes ? Un segment entre accolades, comme `{id}`, accepte n'importe quel mot à sa place. La route accepte-t-elle le verbe de la requête ? Dès qu'une route répond oui aux deux questions, Laravel exécute son action et ne regarde pas les routes suivantes.

L'ensemble des routes, parcouru dans cet ordre, s'appelle la **table de routage**. Le tableau suivant indique, pour sept requêtes, la route que Laravel retient, la valeur que reçoit la fonction et la réponse renvoyée :

Requête	Route retenue	Valeur reçue	Réponse
`GET /ping`	`GET /ping`	aucune	`pong`
`GET /livres`	`GET /livres`	aucune	`Liste des livres`
`POST /livres`	`POST /livres`	aucune	`Livre enregistré`
`GET /livres/12`	`GET /livres/{id}`	`$id` vaut `"12"`	`Fiche du livre 12`
`GET /auteurs/Nothomb`	`GET /auteurs/{name}`	`$name` vaut `"Nothomb"`	`Livres de Nothomb`
`GET /livres/12/avis`	aucune	aucune	404
`DELETE /livres`	aucune pour ce verbe	aucune	405

`/livres/12/avis` compte trois segments, et aucune route n'en déclare trois. Laravel répond alors par le code 404, « page introuvable » :

```bash
curl -s -i http://premier-test-laravel.test/livres/12/avis
```

```
HTTP/1.1 404 Not Found
…
```

Sans option, `curl` n'affiche que le contenu de la réponse. L'option `-i`, vue au chapitre Framework et librairie, affiche en plus la **ligne de statut**, la première ligne, qui porte le code, puis les en-têtes. Le contenu, remplacé ici par `…`, est une page HTML intitulée `Not Found`.

Pour `DELETE /livres`, deux routes déclarent bien le chemin `/livres`, mais l'une accepte `GET` et l'autre `POST`. Aucune n'accepte `DELETE`. L'option `-X DELETE` demande à `curl` d'envoyer une requête `DELETE` :

```bash
curl -s -i -X DELETE http://premier-test-laravel.test/livres
```

```
HTTP/1.1 405 Method Not Allowed
…
```

Laravel répond par le code 405, **Method Not Allowed**, c'est-à-dire « verbe non autorisé ». Comme votre `.env` contient `APP_DEBUG=true`, la page d'erreur donne le détail, avec ce message :

```
The DELETE method is not supported for route livres. Supported methods: GET, HEAD, POST.
```

4. Les paramètres de route

Un paramètre obligatoire

À l'exercice 2 du chapitre MVC, vous avez prévu une route pour `/posts/3`, avec un paramètre pour recevoir le `3`. Ce paramètre s'écrit entre accolades. Dans `'/livres/{id}'`, le segment `{id}` s'appelle un **paramètre de route**. Quand la requête `/livres/12` arrive, Laravel lit `12` à cette place dans l'adresse et le donne à la fonction anonyme, qui le reçoit dans sa variable `$id`. Ce paramètre fait partie du chemin, contrairement au paramètre d'URL `?ordre=ancien` du chapitre MVC, placé après le point d'interrogation.

```bash
curl -s http://premier-test-laravel.test/livres/12
curl -s http://premier-test-laravel.test/livres/abc
```

```
Fiche du livre 12
Fiche du livre abc
```

Laravel transmet le segment exactement comme il est écrit dans l'adresse, sous forme de chaîne de caractères, sans vérifier qu'il s'agit d'un nombre. `abc` est donc accepté comme numéro de livre, et vous verrez plus bas comment le refuser.

Plusieurs paramètres

Une route peut contenir plusieurs paramètres. Imaginez un agenda en ligne dont chaque page montre un mois précis, avec une adresse comme `/agenda/2026/10` pour octobre 2026. Ajoutez cette route à la fin du fichier :

```php
Route::get('/agenda/{year}/{month}', function (int year,intyear, int year,intmonth) {
    return "Agenda du mois monthdel'anne´emonth de l'année monthdel'anne´eyear";
})->whereNumber(['year', 'month']);
```

```bash
curl -s http://premier-test-laravel.test/agenda/2026/10
```

```
Agenda du mois 10 de l'année 2026
```

La fin de la déclaration, `->whereNumber(['year', 'month'])`, ajoute une condition à la route : les segments `year` et `month` ne doivent contenir que des chiffres. Pour l'adresse `/agenda/2026/octobre`, la condition n'est pas remplie, la route ne correspond pas, et Laravel répond 404. Ce type de condition s'appelle une **contrainte**, et la fin de cette section en présente d'autres.

Laravel transmet toujours les segments sous forme de chaînes, ici `"2026"` et `"10"`. Comme la fonction déclare `int $year` et `int $month`, PHP convertit ces chaînes en entiers au moment de l'appel.

Sans `whereNumber`, la route accepterait `/agenda/2026/octobre`. Laravel appellerait la fonction avec la chaîne `"octobre"` pour `$month`, et PHP ne pourrait pas la convertir en entier. La réponse porterait le code 500, erreur du serveur, et la page d'erreur afficherait une `TypeError` avec ce message :

```
Argument #2 ($month) must be of type int, string given
```

Laravel donne les valeurs à la fonction dans l'ordre où les accolades apparaissent dans le chemin, sans regarder le nom des variables. Si vous écrivez `function (int $month, int $year)`, la même adresse affiche `Agenda du mois 2026 de l'année 10`.

Un paramètre optionnel

Un point d'interrogation placé juste après le nom, à l'intérieur des accolades, comme `{day?}`, rend le paramètre **optionnel** : l'adresse peut contenir ce segment ou s'arrêter avant. La fonction doit alors prévoir une valeur par défaut, utilisée quand l'adresse ne fournit rien. Dans l'exemple suivant, une bibliothèque publie ses horaires, et sans jour dans l'adresse, la page montre ceux du lundi. Ajoutez cette route à la fin du fichier :

PHP
```php
Route::get('/horaires/{day?}', function (string $day = 'lundi') {
    $hours = match ($day) {
        'samedi' => 'de 9 h à 12 h',
        'dimanche' => 'fermé',
        default => 'de 9 h à 18 h',
    };

    return "Le $day : $hours";
});
```

BASH
```bash
curl -s http://premier-test-laravel.test/horaires
curl -s http://premier-test-laravel.test/horaires/samedi
curl -s http://premier-test-laravel.test/horaires/mardi
```

```
Le lundi : de 9 h à 18 h
Le samedi : de 9 h à 12 h
Le mardi : de 9 h à 18 h
```

`match` compare `$day` aux valeurs écrites à gauche des flèches, `'samedi'` puis `'dimanche'`, et renvoie le texte écrit à droite de la première valeur égale. La ligne `default` s'applique à tous les autres jours, comme `mardi`.

Si vous oubliez la valeur par défaut et écrivez `function (string $day)`, l'adresse `/horaires/samedi` fonctionne encore, car elle fournit un jour. Pour `/horaires`, en revanche, Laravel appelle la fonction sans argument, alors qu'elle en exige un. La réponse porte le code 500, et la page d'erreur affiche `ArgumentCountError` avec ce message :

```
Too few arguments to function …{closure:…/routes/web.php:36}(), 0 passed … and exactly 1
expected
```

Le nombre `36` est la ligne où commence la route `/horaires` dans le fichier. Remettez ensuite la valeur par défaut `'lundi'`.

Les contraintes

`whereNumber` n'est pas la seule contrainte. Chaque contrainte fixe une condition que la valeur d'un paramètre doit remplir. Si la valeur ne la remplit pas, Laravel considère que la route ne correspond pas et essaie la route suivante du fichier. Voici trois routes qui utilisent chacune une contrainte différente. Remplacez la route `/auteurs/{name}` écrite plus haut par la première, puis ajoutez les deux autres à la fin du fichier :

```php
Route::get('/auteurs/{name}', function (string name) {
    return "Livres de $name";
})->whereAlpha('name');

Route::get('/convertir/{amount}/{currency}', function (int amount, string currency) {
    $rate = ['usd' => 1.17, 'gbp' => 0.87];

    return "$amount EUR = " . $amount * $rate[$currency] . ' ' . strtoupper($currency);
})->whereNumber('amount')->whereIn('currency', ['usd', 'gbp']);

Route::get('/themes/{slug}', function (string slug) {
    return "Livres du thème $slug";
})->where('slug', '[a-z0-9-]+');
```

Les taux de change écrits dans `$rate` sont inventés pour l'exemple. Voici ce que renvoient ces trois routes pour huit adresses :

Adresse	Code	Réponse
`/auteurs/Nothomb`	200	`Livres de Nothomb`
`/auteurs/Nothomb2`	404	
`/convertir/100/usd`	200	`100 EUR = 117 USD`
`/convertir/100/gbp`	200	`100 EUR = 87 GBP`
`/convertir/100/chf`	404	
`/convertir/12.5/usd`	404	
`/themes/romans-policiers`	200	`Livres du thème romans-policiers`
`/themes/Romans`	404	

Contrainte	Valeurs acceptées	Refusé dans l'exemple
`whereNumber('amount')`	des chiffres seulement	`12.5`, à cause du point
`whereAlpha('name')`	des lettres sans accent seulement	`Nothomb2`, et aussi `Hergé`
`whereIn('currency', ['usd', 'gbp'])`	une valeur de la liste	`chf`
`where('slug', '[a-z0-9-]+')`	le motif donné	`Romans`, à cause de la majuscule

Le second argument de `where()`, `'[a-z0-9-]+'`, est une **expression régulière** : un motif qui décrit les caractères permis. Les crochets listent les caractères acceptés, ici les minuscules de `a` à `z`, les chiffres et le tiret. Le `+` qui suit exige au moins un de ces caractères.

La contrainte `whereIn` protège aussi le calcul. Sans elle, l'adresse `/convertir/100/chf` correspondrait à la route, la fonction chercherait `$rate['chf']`, une case absente du tableau, et Laravel afficherait une page d'erreur avec le code 500.

5. Le piège de l'ordre

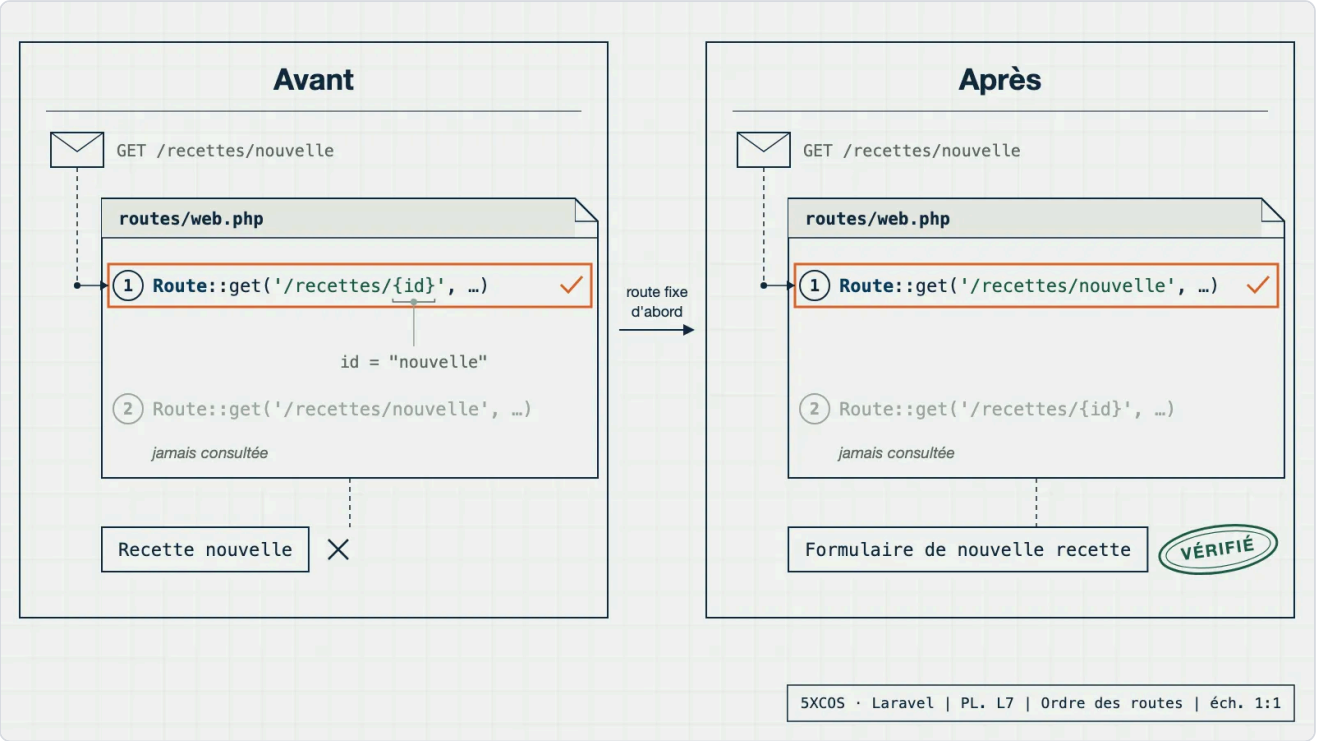
Imaginez un site de recettes de cuisine. Une première page affiche une recette à partir de son numéro, une seconde affiche le formulaire qui permet d'en ajouter une nouvelle. Ajoutez ces deux routes à la fin du fichier, dans cet ordre :

```php
Route::get('/recettes/{id}', function (string $id) {
    return "Recette $id";
});

Route::get('/recettes/nouvelle', function () {
    return 'Formulaire de nouvelle recette';
});
```

```bash
curl -s http://premier-test-laravel.test/recettes/12
curl -s http://premier-test-laravel.test/recettes/nouvelle
```

```
Recette 12
Recette nouvelle
```

La phrase « Formulaire de nouvelle recette » n'apparaît jamais. Pour l'adresse `/recettes/nouvelle`, Laravel essaie d'abord la route `/recettes/{id}`, déclarée en premier. Les deux chemins comptent deux segments, le premier vaut `recettes` des deux côtés, et `{id}` accepte n'importe quel mot, y compris `nouvelle`. Cette première route correspond donc, sa fonction reçoit `"nouvelle"` dans `$id`, et Laravel ne consulte jamais la seconde.



`php artisan route:list` ne vous aide pas à repérer ce problème : la commande trie les routes par ordre alphabétique de leur chemin, et non dans l'ordre où Laravel les essaie.

```bash
php artisan route:list --path=recettes
```

```
  GET|HEAD       recettes/nouvelle ..........................................
routes/web.php:60
  GET|HEAD       recettes/{id} ..............................................
routes/web.php:56

                                                                  Showing [2]
routes
```

L'option `--path=recettes` limite la liste aux routes dont le chemin contient `recettes`. Pour retrouver l'ordre dans lequel Laravel les essaie, regardez les numéros de ligne : `recettes/{id}` est déclarée à la ligne 56, donc avant `recettes/nouvelle`, déclarée à la ligne 60.

Vous pouvez corriger ce problème de deux façons. La première consiste à déclarer d'abord la route dont le chemin ne contient que des segments fixes, `/recettes/nouvelle`, puis la route à paramètre. Échangez les deux routes dans votre fichier :

```php
Route::get('/recettes/nouvelle', function () {
    return 'Formulaire de nouvelle recette';
});

Route::get('/recettes/{id}', function (string $id) {
    return "Recette $id";
});
```

```
Recette 12
Formulaire de nouvelle recette
```

La seconde façon garde l'ordre d'origine et ajoute une contrainte à `{id}`. Remettez les deux routes dans l'ordre de départ, avec la contrainte :

```php
Route::get('/recettes/{id}', function (string $id) {
    return "Recette $id";
})->whereNumber('id');

Route::get('/recettes/nouvelle', function () {
    return 'Formulaire de nouvelle recette';
});
```

```
Recette 12
Formulaire de nouvelle recette
```

Avec la contrainte, `{id}` n'accepte plus que des chiffres. Le mot `nouvelle` est refusé, la première route ne correspond pas, et Laravel essaie la suivante, qui affiche le formulaire. Une adresse comme `/recettes/abc`, qui ne correspond plus à aucune des deux routes, reçoit désormais un 404.

Déclarez les routes fixes avant les routes à paramètre qui commencent de la même façon. Laravel applique lui-même cette règle. Au chapitre Controllers, vous verrez `Route::resource()`, une instruction qui déclare d'un coup toutes les routes d'une ressource. Elle déclare `/recettes/create` avant `/recettes/{recette}`, précisément pour éviter ce piège.

6. Les verbes HTTP

Le chemin d'une requête désigne ce dont on parle, par exemple les livres avec `/livres`. On appelle cela une **ressource**. Le verbe indique ce qu'on veut en faire : lire, créer, modifier ou supprimer. Ces quatre opérations de base portent un nom, **CRUD**, formé des initiales anglaises Create, Read, Update et Delete.

Verbe	Usage	Action CRUD
`GET`	lire une page ou des données	Read, lire
`POST`	créer une ressource, envoyer un formulaire	Create, créer

Verbe	Usage	Action CRUD
`PUT` ou `PATCH`	modifier une ressource existante	Update, modifier
`DELETE`	supprimer une ressource	Delete, supprimer

Quand vous tapez une adresse dans le navigateur, il envoie toujours une requête `GET`. Pour qu'il envoie une requête `POST`, il faut un formulaire HTML, et vous en écrirez un au chapitre Views. En attendant, l'option `-X POST` demande à `curl` d'envoyer une requête `POST`. Essayez-la sur la route `POST /livres` déclarée à la section 3 :

```bash
curl -s -X POST http://premier-test-laravel.test/livres
```

```
<!DOCTYPE html>
…
        <title>Page Expired</title>
…
```

Au lieu de « Livre enregistré », Laravel répond par le code 419 et une page intitulée `Page Expired`. La requête a été refusée avant d'atteindre la fonction. Pour toute requête `POST`, `PUT`, `PATCH` ou `DELETE`, Laravel vérifie d'abord qu'elle a été envoyée depuis une page de votre propre site.

Cette vérification protège contre la **CSRF**, la falsification de requête entre sites. Imaginez une page piégée, sur un autre site, qui contient un formulaire caché dirigé vers `/livres` de votre application. Si l'un de vos utilisateurs ouvre cette page, son navigateur envoie le formulaire à votre site sans qu'il s'en rende compte. La requête de `curl` ne vient, elle non plus, d'aucune page de votre site, et Laravel la refuse de la même façon.

Pour reconnaître une requête venue de votre site, Laravel lit d'abord l'en-tête `Sec-Fetch-Site`. S'il vaut `same-origin`, la requête vient d'une page du même site, et Laravel l'accepte. Les navigateurs n'envoient cet en-tête que sur une connexion HTTPS, et votre projet est servi en HTTP. Dans vos formulaires HTML, vous ajouterez donc `@csrf`, une instruction qui insère un jeton secret que Laravel vérifie. Vous la verrez au chapitre Views. Sans elle, le formulaire reçoit la même réponse 419.

Pour tester une route `POST` avec `curl`, il suffit d'ajouter cet en-tête vous-même, avec l'option `-H` :

```bash
curl -s -X POST -H "Sec-Fetch-Site: same-origin" http://premier-test-laravel.test/livres
```

```
Livre enregistré
```

Que se passe-t-il dans l'autre sens, quand une requête `GET` arrive sur une route qui n'accepte que `POST` ? Laravel répond 405, comme pour `DELETE /livres` à la section 3. Pour le vérifier, ajoutez cette route à la fin du fichier :

```php
Route::post('/contact', function () {
    return 'Message reçu';
});
```

Ouvrez ensuite `http://premier-test-laravel.test/contact` dans le navigateur, ou demandez la ligne de statut avec `curl` :

```bash
curl -s -i http://premier-test-laravel.test/contact
```

```
HTTP/1.1 405 Method Not Allowed
…
```

Dans le navigateur, la page d'erreur affiche ce message :

```
The GET method is not supported for route contact. Supported methods: POST.
```

7. Les autres méthodes de `Route`

`Route::view` : une page sans logique

`Route::view` relie un chemin directement à une vue, sans fonction anonyme ni contrôleur. Elle convient aux pages dont le contenu ne change jamais et qui n'ont besoin d'aucune donnée, comme une page « À propos ». Créez le fichier `resources/views/about.blade.php` avec ce contenu :

```xml
<h1>À propos</h1>
<p>La bibliothèque est ouverte du lundi au samedi.</p>
```

Ajoutez ensuite la route à la fin de `routes/web.php` :

```php
Route::view('/a-propos', 'about');
```

```bash
curl -s http://premier-test-laravel.test/a-propos
```

```
<h1>À propos</h1>
<p>La bibliothèque est ouverte du lundi au samedi.</p>
```

Le premier argument de `Route::view` est le chemin de l'adresse, le second est le nom de la vue, écrit sans `.blade.php`, exactement comme vous l'écririez dans `view('about')`.

`Route::redirect` : envoyer ailleurs

Une **redirection** est une réponse qui demande au navigateur d'ouvrir une autre adresse, donnée dans l'en-tête `Location`. Le navigateur envoie alors une seconde requête, vers cette adresse. Une redirection sert par exemple quand une page a changé d'adresse : les anciens liens continuent de fonctionner. Laravel propose deux méthodes pour cela, `Route::redirect` et `Route::permanentRedirect`. Ajoutez ces deux routes à la fin du fichier :

```php
Route::redirect('/qui-sommes-nous', '/a-propos');

Route::permanentRedirect('/presentation', '/a-propos');
```

```bash
curl -s -I http://premier-test-laravel.test/qui-sommes-nous
curl -s -I http://premier-test-laravel.test/presentation
```

```
HTTP/1.1 302 Found
…
Location: /a-propos
…
HTTP/1.1 301 Moved Permanently
…
Location: /a-propos
…
```

L'option `-I`, en majuscule, n'affiche que la ligne de statut et les en-têtes, sans le contenu. La ligne `Location` donne l'adresse vers laquelle le navigateur est renvoyé.

`Route::redirect` répond par le code 302, qui annonce une redirection temporaire : l'ancienne adresse pourra resservir plus tard. `Route::permanentRedirect` répond par le code 301, qui annonce que la page a changé d'adresse pour de bon. Les moteurs de recherche remplacent alors l'ancienne adresse par la nouvelle dans leurs résultats.

`Route::match` et `Route::any`

`Route::match` accepte plusieurs verbes pour un même chemin, et `Route::any` les accepte tous. Ajoutez ces deux routes à la fin du fichier :

```php
Route::match(['get', 'post'], '/recherche', function () {
    return 'Résultats de la recherche';
});

Route::any('/tout', function () {
    return 'Tous les verbes';
});
```

```bash
curl -s http://premier-test-laravel.test/recherche
curl -s -X POST -H "Sec-Fetch-Site: same-origin" http://premier-test-laravel.test/recherche
curl -s -X DELETE -H "Sec-Fetch-Site: same-origin" http://premier-test-laravel.test/tout
```

```
Résultats de la recherche
Résultats de la recherche
Tous les verbes
```

Une requête `PUT` vers `/recherche` reçoit un 405, car `PUT` ne figure pas dans la liste donnée à `Route::match`. Vous rencontrerez rarement ces deux méthodes. Avec une route par verbe, vous voyez en lisant `routes/web.php` ce que chaque requête déclenche.

Il existe aussi `Route::resource()`, qui déclare en une ligne les sept routes nécessaires pour afficher, créer, modifier et supprimer des livres, par exemple. Chacune de ces routes appelle une méthode différente du même contrôleur. Vous la découvrirez donc au chapitre Controllers, une fois ce contrôleur écrit.

`Route::fallback` : la route de repli

Une **route de repli** répond aux requêtes `GET` qu'aucune autre route n'a retenues. Elle remplace la page 404 de Laravel par la vôtre. Ajoutez-la à la fin du fichier :

```php
Route::fallback(function () {
    return response("Cette page n'existe pas.", 404);
});
```

```bash
curl -s -i http://premier-test-laravel.test/nulle-part
```

```
HTTP/1.1 404 Not Found
…
Cette page n'existe pas.
```

La fonction `response()` fabrique une réponse à partir de deux éléments : le contenu, puis le code. Le second argument, `404`, est important. Sans lui, Laravel enverrait le code 200, et un navigateur ou un moteur de recherche croirait que l'adresse `/nulle-part` existe vraiment.

Laravel essaie toujours la route de repli en dernier, même si vous la déclarez en haut de `routes/web.php`. Elle modifie aussi une réponse vue plus haut. Une requête `GET` vers `/contact`, qui n'accepte que `POST`, ne reçoit plus le 405, mais la page de repli avec un 404 :

```bash
curl -s -i http://premier-test-laravel.test/contact
```

```
HTTP/1.1 404 Not Found
…
Cette page n'existe pas.
```

Retirez cette route de repli avant de continuer : elle masquerait les 404 et les 405 des exercices.

8. Les routes nommées

Les pages d'un site contiennent des liens vers d'autres pages, et certaines actions renvoient le visiteur vers une autre adresse avec une redirection. Vous pouvez écrire ces adresses à la main, comme `/livres/12`. Laravel propose une autre façon de faire : vous donnez un nom à la route, puis vous lui demandez l'adresse qui correspond à ce nom.

Nommer une route

Pour nommer une route, ajoutez `->name()` à la fin de sa déclaration, juste avant le point-virgule. Remplacez les routes `GET /livres` et `GET /livres/{id}` de la section 3 par celles-ci :

PHP

```php
Route::get('/livres', function () {
    return 'Liste des livres';
})->name('books.index');

Route::post('/livres', function () {
    return 'Livre enregistré';
});

Route::get('/livres/{id}', function (string $id) {
    return "Fiche du livre $id";
})->name('books.show');
```

La route `POST /livres`, qui les sépare, ne change pas. Une **route nommée** est une route qui a reçu un nom avec `->name()`, ici `books.index` pour la liste des livres et `books.show` pour la fiche d'un livre. `route:list` affiche ce nom avant l'action :

BASH

```bash
php artisan route:list --path=livres
```

```
  GET|HEAD   livres ..................................... books.index ›
routes/web.php:16
  POST       livres .....................................................
routes/web.php:20
  GET|HEAD   livres/{id} ................................... books.show ›
routes/web.php:24

                                                                Showing [3]
routes
```

La route `POST /livres` n'a pas reçu de nom, et sa ligne n'indique que le fichier et le numéro de ligne.

Obtenir l'adresse à partir du nom

La fonction `route()` reçoit le nom d'une route et la valeur de ses paramètres, puis renvoie l'adresse complète. Ainsi, `route('books.show', 12)` renvoie l'adresse de la fiche du livre 12. Vous pouvez appeler cette fonction depuis le terminal, dans une vue ou dans l'action d'une route.

Dans le terminal. Tinker, que vous avez utilisé au chapitre MVC, exécute une instruction PHP dans votre projet :

```bash
php artisan tinker --execute="echo route('books.show', 12);"
```

```
http://premier-test-laravel.test/livres/12
```

Dans le terminal, aucun navigateur n'envoie de requête. Laravel prend donc le début de l'adresse dans la variable `APP_URL` du fichier `.env`. Si l'adresse affichée ne commence pas par `http://premier-test-laravel.test`, écrivez `APP_URL=http://premier-test-laravel.test` dans `.env` et relancez la commande.

Dans une vue Blade. Créez le dossier `resources/views/books`, puis, dans ce dossier, le fichier `links.blade.php`. Ce template contient deux liens vers la fiche du livre 12. Le premier donne l'adresse écrite à la main, le second la demande à `route()`.

```xml
<h1>Le livre 12</h1>
<ul>
    <li><a href="/livres/12">Lien écrit à la main</a></li>
    <li><a href="{{ route('books.show', 12) }}">Lien écrit avec route()</a></li>
</ul>
```

Les doubles accolades insèrent dans le HTML le résultat de l'expression PHP qu'elles contiennent, comme `{{ $post->title }}` dans la vue du chapitre MVC. Ajoutez ensuite cette route à la fin de `routes/web.php`. Le nom de vue `books.links` désigne le fichier `links.blade.php` du dossier `books`, de la même façon que `posts.index` au chapitre MVC.

```php
Route::view('/liens', 'books.links');
```

```bash
curl -s http://premier-test-laravel.test/liens
```

```
<h1>Le livre 12</h1>
<ul>
    <li><a href="/livres/12">Lien écrit à la main</a></li>
    <li><a href="http://premier-test-laravel.test/livres/12">Lien écrit avec route()</a>
</li>
</ul>
```

Les deux liens mènent à la même page. Le second contient l'adresse complète, car `route()` reprend le début de l'adresse de la requête reçue.

Dans l'action d'une route. Imaginez une page « dernier livre arrivé » qui renvoie le visiteur vers la fiche de ce livre, ici le livre 12. Ajoutez cette route à la fin du fichier :

```php
Route::get('/dernier-livre', function () {
    return redirect()->route('books.show', 12);
});
```

Cette route n'a pas de nom. Sa fonction utilise le nom d'une autre route, `books.show`, pour construire l'adresse de la redirection. `redirect()->route('books.show', 12)` demande à `route()` l'adresse du livre 12, puis renvoie une réponse 302 qui contient cette adresse dans l'en-tête `Location`. La fonction `to_route('books.show', 12)` fait exactement la même chose, en plus court.

```bash
curl -s -I http://premier-test-laravel.test/dernier-livre
```

```
HTTP/1.1 302 Found
…
Location: http://premier-test-laravel.test/livres/12
…
```

À quoi sert le nom

Supposez que le site range désormais ses fiches à l'adresse `/catalogue/livres/12`. Dans `routes/web.php`, modifiez seulement le chemin de la route `books.show` :

```php
Route::get('/catalogue/livres/{id}', function (string $id) {
    return "Fiche du livre $id";
})->name('books.show');
```

Ne touchez ni au nom de la route, ni au template `links.blade.php`, ni à la route `/dernier-livre`. Demandez de nouveau la page `/liens` :

```bash
curl -s http://premier-test-laravel.test/liens
```

```
<h1>Le livre 12</h1>
<ul>
    <li><a href="/livres/12">Lien écrit à la main</a></li>
    <li><a href="http://premier-test-laravel.test/catalogue/livres/12">Lien écrit avec
route()</a></li>
</ul>
```

Le lien écrit avec `route()` pointe maintenant vers `/catalogue/livres/12`. Le lien écrit à la main pointe toujours vers `/livres/12`, une adresse qu'aucune route ne déclare plus :

BASH

```
curl -s -i http://premier-test-laravel.test/livres/12
```

```
HTTP/1.1 404 Not Found
…
```

La redirection de `/dernier-livre` a suivi, elle aussi :

BASH

```
curl -s -I http://premier-test-laravel.test/dernier-livre
```

```
HTTP/1.1 302 Found
…
Location: http://premier-test-laravel.test/catalogue/livres/12
…
```

Dans un vrai site, l'adresse de la fiche d'un livre figure dans beaucoup de vues et de contrôleurs. Si ces fichiers écrivent `/livres/12` à la main, vous devez les ouvrir un par un pour corriger chaque lien, et un lien oublié envoie le visiteur sur une page 404. S'ils utilisent le nom `books.show`, seule la déclaration de la route change dans `routes/web.php`. **Écrivez les liens et les redirections avec `route()` et le nom de la route.**

Par convention, le nom d'une route réunit le nom de la ressource et celui de l'action, séparés par un point : `books.index` pour la liste des livres, `books.show` pour la fiche d'un livre. Remettez ensuite le chemin `'/livres/{id}'`.

À retenir

- Laravel compare chaque requête aux routes de `routes/web.php` de haut en bas, et la première qui correspond répond.
- Une adresse qu'aucune route ne déclare reçoit un 404. Une adresse déclarée, mais pour un autre verbe, reçoit un 405.
- Un paramètre `{id}` arrive dans l'action sous forme de chaîne, et une contrainte comme `whereNumber` limite les valeurs qu'il accepte.

- Une route fixe se déclare avant une route à paramètre qui commence de la même façon.
- Une route qui interroge un modèle ou prépare une vue avec des données appelle un contrôleur.
- `route()` produit l'adresse d'une route à partir de son nom, dans une vue comme dans une redirection. Quand le chemin change, ces adresses suivent, alors qu'un lien écrit à la main mène à un 404.

Exercices

Comptez 45 minutes pour les exercices, en séance ou à la maison.

Faites les exercices 1 à 9 dans `routes/web.php` de `premier-test-laravel`, sans contrôleur, en ajoutant chaque route à la fin du fichier. Après chaque route, vérifiez l'adresse dans le navigateur ou avec `curl`.

1. Une première route

Écrivez une route `GET /hello` qui affiche « Hello World ! Ceci est ma première route Laravel. ». La route `/ping` de la section 2 sert de modèle.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/routing

2. Un paramètre

Écrivez une route `/profil/{firstName}` qui affiche « Bienvenue sur votre profil, Alice ! » pour l'adresse `/profil/Alice`. Pour un autre prénom, la phrase reprend celui qui figure dans l'adresse.

```bash
curl -s http://premier-test-laravel.test/profil/Alice
curl -s http://premier-test-laravel.test/profil/Bob
```

```
Bienvenue sur votre profil, Alice !
Bienvenue sur votre profil, Bob !
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/routing

3. Deux paramètres

Écrivez une route `/calcul/{a}/{b}` qui affiche « Le résultat est : 10 » pour `/calcul/4/6`. `/calcul/4/six` doit répondre 404. Inspirez-vous de l'agenda de la section 4.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/routing

4. Un paramètre optionnel

Écrivez une route `/bienvenue/{lang?}` :

- sans langue, elle affiche « Bienvenue ! »,
- avec `en`, elle affiche « Welcome! »,
- avec `es`, elle affiche « ¡Bienvenido! »,
- avec toute autre langue, elle affiche « Langue non supportée. ».

Les horaires de la section 4 utilisent le même mécanisme. Vérifiez les quatre cas :

```bash
curl -s http://premier-test-laravel.test/bienvenue
curl -s http://premier-test-laravel.test/bienvenue/en
curl -s http://premier-test-laravel.test/bienvenue/es
curl -s http://premier-test-laravel.test/bienvenue/de
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/routing

5. Une route POST

Écrivez une route `POST /formulaire` qui renvoie « Formulaire envoyé avec succès ! ». Lancez ces deux commandes, puis ouvrez `http://premier-test-laravel.test/formulaire` dans le navigateur. Notez le code obtenu dans chacun des trois cas.

```bash
curl -s -i -X POST -H "Sec-Fetch-Site: same-origin" http://premier-test-
laravel.test/formulaire
curl -s -i -X POST http://premier-test-laravel.test/formulaire
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/routing

6. Une redirection

Écrivez une route `/ici` qui renvoie le navigateur vers `/hello`, la route de l'exercice 1. Vérifiez que la réponse porte le code 302 et l'en-tête `Location: /hello` :

```bash
curl -s -I http://premier-test-laravel.test/ici
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/routing

7. L'ordre des routes

Une boutique déclare ces deux routes, dans cet ordre :

```php
Route::get('/commandes/{number}', function (string $number) {
    return "Commande n° $number";
});

Route::get('/commandes/panier', function () {
    return 'Votre panier';
});
```

1. Sans rien lancer, écrivez ce qu'affichent `/commandes/15` et `/commandes/panier`.
2. Ajoutez les deux routes à la fin de votre fichier, puis vérifiez votre prédiction.
3. Corrigez de deux façons : d'abord en changeant l'ordre, puis en gardant l'ordre et en ajoutant une contrainte. Après chaque correction, `/commandes/panier` doit afficher « Votre panier ».

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/routing

8. Des mentions légales

Créez la vue `resources/views/legal.blade.php` avec un titre `<h1>Mentions légales</h1>` et une phrase. Affichez-la à l'adresse `/mentions-legales` avec une seule ligne dans `routes/web.php`, sans fonction ni contrôleur.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/routing

9. Deux routes nommées

Donnez le nom `hello` à la route de l'exercice 1 et le nom `profile` à celle de l'exercice 2. Affichez ensuite leurs adresses avec Tinker, comme à la section 8 :

```bash
php artisan tinker --execute="echo route('hello');"
php artisan tinker --execute="echo route('profile', 'Alice');"
```

Remplacez ensuite la redirection `/ici` de l'exercice 6 par une route `GET /ici` dont la fonction renvoie le visiteur vers la route nommée `hello`, comme la route `/dernier-livre` de la section 8. Changez enfin le chemin `/hello` en `/bonjour`, sans toucher au nom, et vérifiez que l'en-tête `Location` suit ce changement :

```bash
curl -s -I http://premier-test-laravel.test/ici
```

Remettez ensuite le chemin `/hello`.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/routing

10. Quelle route répond ? (sur papier, 10 minutes)

Un musée déclare ces six routes, dans cet ordre :

```php
Route::get('/expositions', function () {
    return 'Toutes les expositions';
});

Route::get('/expositions/{id}', function (string id) {
    return "Exposition $id";
})->whereNumber('id');

Route::get('/expositions/prochaines', function () {
    return 'Prochaines expositions';
});

Route::post('/reservations', function () {
    return 'Réservation enregistrée';
});

Route::get('/visites/{lang?}', function (string lang = 'fr') {
    return "Visite guidée en lang";
});

Route::get('/billets/{price}/{count}', function (string price, int $count) {
    return "$count billet(s) au tarif $price";
})->whereAlpha('price')->whereNumber('count');
```

Pour chaque requête `GET`, écrivez la réponse affichée, ou le code 404 ou 405 :

1. `/expositions`
2. `/expositions/7`
3. `/expositions/prochaines`
4. `/reservations`
5. `/visites`
6. `/visites/nl`
7. `/billets/reduit/3`
8. `/billets/3/reduit`

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/routing

11. Six cartes

Deux routes correspondent à la même requête. Laquelle répond ?

La première déclarée dans `routes/web.php`. Laravel s'arrête à la première qui correspond.

Quelle différence entre un 404 et un 405 ?

404 : aucune route n'a ce chemin. 405 : une route a ce chemin, mais pas pour ce verbe.

Que faut-il ajouter à la fonction pour une route `/horaires/{day?}` ?

Une valeur par défaut : `string $day = 'lundi'`. Sans elle, `/horaires` provoque « Too few arguments ».

`/recettes/{id}` est déclarée avant `/recettes/nouvelle`. Comment corriger ?

Déclarer `/recettes/nouvelle` en premier, ou ajouter `→whereNumber('id')` à `/recettes/{id}`.

Quand utiliser `Route::view` ?

Pour une page qui affiche une vue sans données, comme « À propos ».

Pourquoi nommer une route ?

Pour fabriquer son adresse avec `route('books.show', 12)`. Si vous changez le chemin, les liens et les redirections écrits avec `route()` suivent, sans modifier les vues ni les contrôleurs.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/routing

3.8 Controllers

Bloc Laravel · séance 2 · ~55 min

Slides du chapitre (PDF) PDF

Au chapitre Routing, les routes `/livres` et `/livres/{id}` appelaient chacune une fonction anonyme écrite dans `routes/web.php`, qui renvoyait une phrase. Dans ce chapitre, vous déplacez le code de ces fonctions dans une classe, `BookController`, que vous créez avec Artisan. Vous y ajoutez ensuite des méthodes qui lisent la requête et renvoient une page, une redirection ou des données au format **JSON**, un texte structuré qu'un autre programme sait lire. Tout se fait dans `premier-test-laravel`, sans base de données.

À la fin de ce chapitre, vous serez capables de :

1. **créer** un contrôleur avec Artisan et relier ses méthodes à des routes,

2. **écrire** une méthode qui reçoit un paramètre de route ou la requête,
3. **choisir** ce que renvoie une méthode (texte, vue, redirection ou JSON) et écrire son type de retour,
4. **créer** un contrôleur à action unique et un contrôleur rangé dans un sous-dossier,
5. **diagnostiquer** une classe introuvable, une méthode absente et un `return` oublié.

1. Des fonctions anonymes au contrôleur

Vous avez créé `PostController` au chapitre MVC avec la commande `make:controller`. Un contrôleur est une classe dont les méthodes répondent aux requêtes : pour chaque requête, Laravel appelle une de ces méthodes, puis fabrique la réponse à partir de ce qu'elle renvoie. Une route désigne la méthode à appeler avec un tableau, comme `[PostController::class, 'index']`. Au chapitre Routing, vous avez adopté une règle : dès que le code d'une route interroge un modèle ou prépare une vue avec des données, placez ce code dans un contrôleur.

Pour l'instant, votre `routes/web.php` contient les routes écrites pendant le chapitre Routing, sans la route de repli, puis, à la fin du fichier, celles de ses exercices. Les routes des livres y appellent encore des fonctions anonymes. La commande `route:list` le confirme : à droite de chaque route, elle indique le fichier et la ligne où la fonction est écrite.

```bash
php artisan route:list --path=livres
```

```
  GET|HEAD   livres ..................................... books.index ›
routes/web.php:16
  POST       livres .....................................................
routes/web.php:20
  GET|HEAD   livres/{id} ................................... books.show ›
routes/web.php:24

                                                                Showing [3]
routes
```

Dans ce chapitre, vous ajoutez aux livres une page de recherche, une page d'avis et une fiche en JSON. Si vous écrivez pour chacune une fonction de cinq ou dix lignes dans `routes/web.php`, le code des pages finit par cacher les adresses, et il devient difficile de voir d'un coup d'œil quelles adresses le site accepte. Vous allez donc rassembler le code des livres dans un seul fichier, `app/Http/Controllers/BookController.php`. Dans `routes/web.php`, chaque route des livres tiendra alors sur une ligne.

2. Créer un contrôleur avec Artisan

La commande et le fichier produit

À la racine du projet, lancez la commande qui crée le contrôleur :

```bash
php artisan make:controller BookController
```

```
   INFO  Controller [app/Http/Controllers/BookController.php] created successfully.
```

Artisan a écrit ce fichier :

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class BookController extends Controller
{
    //
}
```

Vous avez déjà rencontré presque toutes ces lignes dans le bloc POO.

- `namespace App\Http\Controllers;` correspond au dossier `app/Http/Controllers/`. C'est la règle PSR-4, vue au chapitre L'atelier du bloc POO : quand PHP a besoin de la classe `App\Http\Controllers\BookController`, le chargement automatique de Composer transforme ce nom en chemin et ouvre `app/Http/Controllers/BookController.php`.
- `use Illuminate\Http\Request;` importe la classe qui représente une requête. Artisan l'ajoute d'avance, et elle servira à la section 4.
- `extends Controller` fait hériter `BookController` de la classe `Controller`, écrite dans `app/Http/Controllers/Controller.php`. Vous l'avez ouverte au chapitre Structure d'un projet Laravel : c'est une classe abstraite, et elle est vide. Laravel n'oblige pas un contrôleur à en hériter. Elle sert le jour où tous vos contrôleurs ont besoin d'une même méthode : vous l'écrivez une seule fois dans `Controller`, et chaque contrôleur en hérite.
- `//` marque l'endroit où vous écrirez les méthodes.

Déplacer les routes des livres

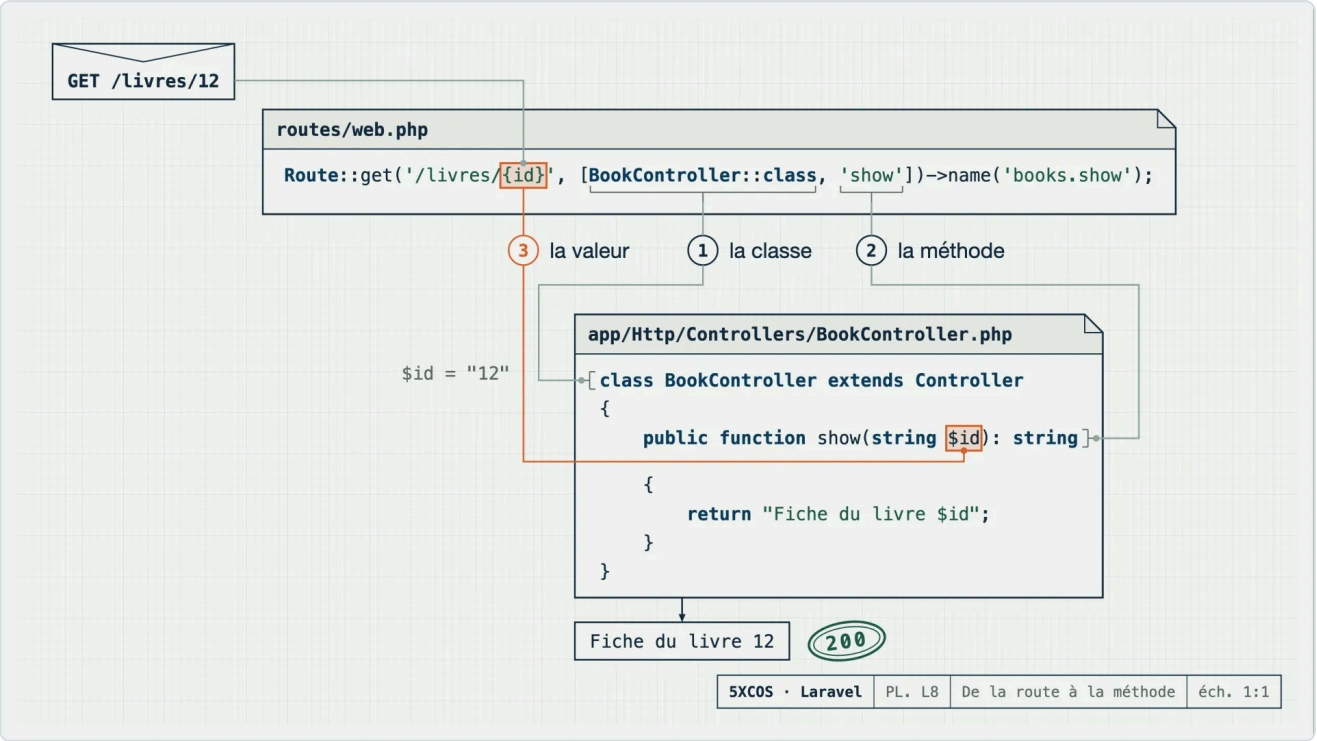
Remplacez le contenu de la classe par deux méthodes, qui renvoient les mêmes phrases que les fonctions anonymes du chapitre Routing :

```php
class BookController extends Controller
{
    public function index(): string
    {
        return 'Liste des livres';
    }

    public function show(string $id): string
    {
        return "Fiche du livre $id";
    }
}
```

En haut de `routes/web.php`, ajoutez la ligne `use App\Http\Controllers\BookController;` juste au-dessus de celle qui importe `PostController`. Remplacez ensuite les routes `GET /livres` et `GET /livres/{id}` par les deux lignes ci-dessous. La route `POST /livres`, placée entre les deux, reste telle quelle.

```php
use App\Http\Controllers\BookController;
use App\Http\Controllers\PostController;
use Illuminate\Support\Facades\Route;

// …

Route::get('/livres', [BookController::class, 'index'])->name('books.index');

Route::post('/livres', function () {
    return 'Livre enregistré';
});

Route::get('/livres/{id}', [BookController::class, 'show'])->name('books.show');
```

Les routes gardent leur nom, `books.index` et `books.show`. Le template `books/links.blade.php` et la redirection `/dernier-livre` du chapitre Routing, qui utilisent ces noms, continuent donc de fonctionner.



Pour une requête `GET /livres/12`, Laravel parcourt `routes/web.php` et retient la route `/livres/{id}`. Il lit la classe et la méthode dans le tableau `[BookController::class, 'show']`, crée un objet `BookController` et appelle `show()` avec la valeur `"12"`. Laravel place la phrase renvoyée par la méthode dans la réponse, et c'est elle que `curl` affiche :

```bash
curl -s http://premier-test-laravel.test/livres
curl -s http://premier-test-laravel.test/livres/12
```

```
Liste des livres
Fiche du livre 12
```

Les deux adresses renvoient les mêmes phrases qu'avant le déplacement. Ce qui change, c'est ce qu'affiche `route:list` :

```bash
php artisan route:list --path=livres
```

```
  GET|HEAD   livres ..................................... books.index ›
BookController@index
  POST       livres ..................................................
routes/web.php:19
  GET|HEAD   livres/{id} ................................. books.show ›
BookController@show

                                                          Showing [3]
routes
```

Pour les deux routes des livres, `route:list` indique maintenant la classe et la méthode appelées, `BookController@index` et `BookController@show`. La route `POST /livres` appelle toujours une fonction anonyme, et la liste indique donc toujours sa ligne dans `routes/web.php`, maintenant 19 au lieu de 20. L'import ajouté en haut du fichier a décalé cette route d'une ligne vers le bas, mais la route `GET /livres`, placée au-dessus, tient maintenant sur une ligne au lieu de trois.

3. Les paramètres de route dans une méthode

Une méthode de contrôleur reçoit les paramètres de route de la même façon qu'une fonction anonyme, comme au chapitre Routing. Laravel prend dans l'adresse les valeurs qui occupent la place des accolades, puis les donne aux paramètres de la méthode dans le même ordre. Le nom de vos variables ne joue aucun rôle.

Le numéro d'un livre est un nombre. Dans `BookController`, remplacez donc `string $id` par `int $id` :

```php
public function show(int $id): string
{
    return "Fiche du livre $id";
}
```

Avec l'adresse `/livres/12`, la page s'affiche toujours, car PHP convertit la chaîne `"12"` en entier au moment d'appeler `show()`. Avec `/livres/abc`, cette conversion est impossible. PHP lance alors une `TypeError`, l'exception qui signale une valeur du mauvais type, et Laravel renvoie une page d'erreur avec le code 500. Elle affiche ce message :

```
App\Http\Controllers\BookController::show(): Argument #1 ($id) must be of type int, string
given
```

Le message nomme la classe et la méthode, puis le paramètre en cause, `$id`. Vous corrigez ce problème comme au chapitre Routing, avec une contrainte sur la route. Dans `routes/web.php`, ajoutez `whereNumber('id')` à la route `books.show` :

```php
Route::get('/livres/{id}', [BookController::class, 'show'])->whereNumber('id')
->name('books.show');
```

```bash
curl -s -i http://premier-test-laravel.test/livres/abc
```

```
HTTP/1.1 404 Not Found
…
```

Avec la contrainte, `abc` ne correspond plus à la route `books.show`. Laravel ne trouve aucune autre route pour cette adresse, et il répond 404 sans appeler `show()`. **Quand une méthode déclare un paramètre `int`, ajoutez `whereNumber` à sa route.**

4. Lire la requête

Recevoir `Request $request`

Au chapitre MVC, `posts.php` lisait le paramètre d'URL `?ordre=ancien` dans le tableau `$_GET`. Dans un contrôleur, vous lisez ce type de valeur dans un objet de la classe `Illuminate\Http\Request`, celle que `make:controller` a importée. Cet objet contient tout ce que le navigateur a envoyé : le verbe, le chemin, les paramètres d'URL, et des informations techniques appelées en-têtes, que vous rencontrerez à la section 5.

Pour recevoir cet objet, ajoutez un paramètre `Request $request` à la méthode qui en a besoin. Avant d'appeler une méthode de contrôleur, Laravel regarde le type de chacun de ses paramètres. Quand un paramètre a le type `Request`, Laravel lui donne comme valeur l'objet qui représente la requête en cours. Ajoutez cette méthode à `BookController` :

```php
public function search(Request $request): string
{
    $word = $request->query('q', 'rien');

    return "Recherche de « $word », requête {$request->method()} sur {$request->path()}";
}
```

- `$request->query('q', 'rien')` renvoie la valeur du paramètre d'URL `q`. Si l'adresse ne contient pas `?q=`, elle renvoie le second argument, `'rien'`.
- `$request->method()` renvoie le verbe de la requête, ici `GET`.
- `$request->path()` renvoie le chemin de l'adresse demandée, sans la barre oblique du début, ici `livres/recherche`.

Dans une chaîne entre guillemets doubles, les accolades autour de `$request->method()` indiquent à PHP d'appeler la méthode et d'insérer dans la chaîne la valeur qu'elle renvoie.

Placer la route au bon endroit

L'adresse `/livres/recherche` correspond aussi à la route `/livres/{id}` : `recherche` pourrait très bien être la valeur de `{id}`. Au chapitre Routing, vous avez vu que Laravel garde alors la première des deux routes déclarées. Déclarez donc la route fixe, `/livres/recherche`, avant la route à paramètre. Ajoutez-la entre la route `POST /livres` et la route `books.show` :

```php
Route::get('/livres/recherche', [BookController::class, 'search'])->name('books.search');
```

```bash
curl -s "http://premier-test-laravel.test/livres/recherche?q=Nothomb"
curl -s http://premier-test-laravel.test/livres/recherche
```

```
Recherche de « Nothomb », requête GET sur livres/recherche
Recherche de « rien », requête GET sur livres/recherche
```

Les guillemets autour de l'adresse sont nécessaires : sans eux, le terminal traite le `?` comme un caractère spécial et ne transmet pas l'adresse telle quelle à `curl`.

Dans votre fichier, la contrainte `whereNumber` ajoutée à la section 3 suffirait déjà : `recherche` ne contient pas de chiffres, donc l'adresse ne correspond pas à `/livres/{id}`. L'ordre des routes devient décisif quand une route à paramètre n'a pas de contrainte. Si `/livres/{id}` passait en premier sans `whereNumber`, Laravel appellerait `show()` avec `"recherche"`, et vous retrouveriez l'erreur 500 de la section 3.

La requête et un paramètre de route ensemble

Une méthode peut recevoir à la fois la requête et un paramètre de route. Imaginez une page qui affiche les avis sur un livre, à l'adresse `/livres/12/avis`. Le visiteur choisit l'ordre des avis en ajoutant à l'adresse le paramètre d'URL `tri`, par exemple `?tri=note`. Ajoutez cette méthode à `BookController` :

PHP
```php
public function reviews(Request request,intrequest, int request,intid): string
{
    sort=sort = sort=request→query('tri', 'date');

    return "Avis sur le livre id,trie´sparid, triés par id,trie´sparsort";
}
```

Ajoutez la route juste après la route `books.show` :

PHP
```php
Route::get('/livres/{id}/avis', [BookController::class, 'reviews'])→whereNumber('id');
```

BASH
```bash
curl -s "http://premier-test-laravel.test/livres/12/avis?tri=note"
curl -s http://premier-test-laravel.test/livres/12/avis
```

```
Avis sur le livre 12, triés par note
Avis sur le livre 12, triés par date
```

Laravel repère le paramètre de type `Request` et lui donne la requête. Il donne ensuite la valeur `12`, lue dans l'adresse, au paramètre qui reste, `$id`. Si vous écrivez `reviews(int` $id, Request$`request)`, la réponse est la même, car Laravel repère toujours la requête par son type. La documentation de Laravel place la requête en premier, et ce cours suit cette habitude.

5. Ce que renvoie une méthode

Cinq retours, cinq réponses

Jusqu'ici, vos méthodes renvoient une chaîne. Une méthode peut renvoyer d'autres valeurs, et Laravel fabrique la réponse HTTP à partir de ce qu'elle renvoie.

Cette réponse commence par un code, comme 200 ou 302, suivi de lignes appelées **en-têtes**, qui décrivent la réponse. L'en-tête `Content-Type` annonce la nature du contenu : `text/html` pour une page, `application/json` pour des données JSON. L'en-tête `Location` donne l'adresse vers laquelle rediriger. L'option `-i` de `curl` affiche ce code et ces en-têtes.

La méthode renvoie	Type de retour	Réponse
`"Fiche du livre $id"`	`string`	200, `Content-Type: text/html; charset=utf-8`, la phrase
`view('books.links')`	`View`	200, `Content-Type: text/html; charset=utf-8`, le HTML produit par le template
`redirect()->route('books.show', 12)`	`RedirectResponse`	302, `Location: http://premier-test-laravel.test/livres/12`
`response()->json(['id' => 12, 'title' => 'Livre 12'])`	`JsonResponse`	200, `Content-Type: application/json`, `{"id":12,"title":"Livre 12"}`
`['id' => 12, 'title' => 'Livre 12']`	`array`	la même réponse que la ligne précédente

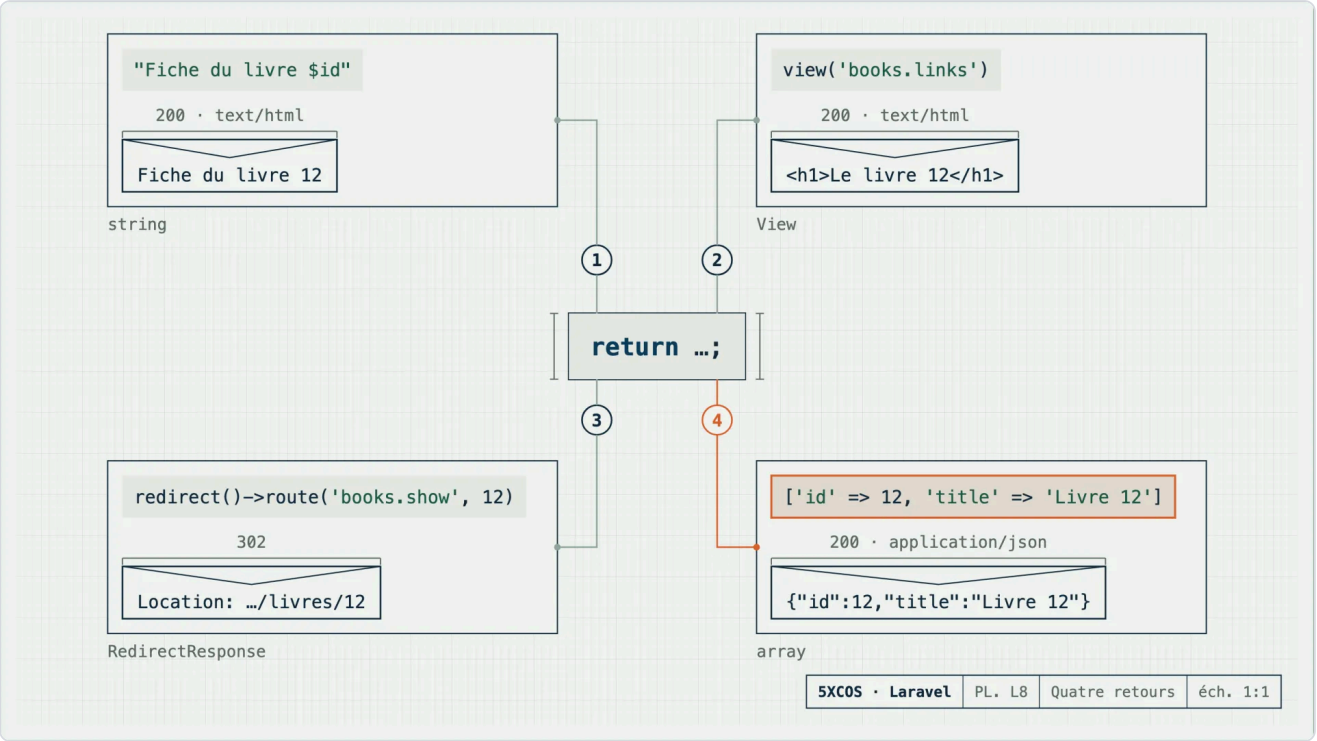
`string` et `array` sont des types de PHP, utilisables sans import. Les trois autres sont des classes de Laravel. Une méthode qui en utilise une comme type de retour demande l'import de la classe en haut du contrôleur, comme `View` dans le `PostController` du chapitre MVC.

Voici ces trois imports. Ne les collez pas dans `BookController` maintenant : vous ajouterez chaque import au moment d'écrire la méthode qui en a besoin.

```php
use Illuminate\Http\JsonResponse;
use Illuminate\Http\RedirectResponse;
use Illuminate\View\View;
```

Un import écrit deux fois dans le même fichier arrête PHP avec ce message, et Laravel renvoie alors une erreur 500 pour toutes les routes du contrôleur :

```
Cannot use Illuminate\Http\RedirectResponse as RedirectResponse because the name is already in use
```



Renvoyer du JSON

Un texte JSON décrit des données avec des accolades, des clés entre guillemets et des valeurs. Un programme, par exemple une application mobile, lit ces données directement. Il n'a pas à les chercher au milieu des balises d'une page HTML, qui mélange les données et leur mise en forme. Ajoutez cette méthode à `BookController` :

```php
public function data(int $id): array
{
    return ['id' => $id, 'title' => "Livre $id", 'available' => true];
}
```

Ajoutez la route après celle des avis :

```php
Route::get('/livres/{id}/donnees', [BookController::class, 'data'])->whereNumber('id');
```

```bash
curl -s -i http://premier-test-laravel.test/livres/12/donnees
```

```
HTTP/1.1 200 OK
…
Content-Type: application/json
…
{"id":12,"title":"Livre 12","available":true}
```

Laravel a converti le tableau PHP en texte JSON et a ajouté à la réponse l'en-tête `Content-Type: application/json`. Grâce à cet en-tête, le programme qui reçoit la réponse sait qu'il doit la lire comme du JSON, et non comme une page HTML. La conversion garde le type des valeurs : `true` reste un booléen et `12` un nombre, écrits sans guillemets, alors que le titre est une chaîne entre guillemets.

Rediriger avec un paramètre d'URL

Supposez que l'ancienne version du site proposait la recherche à l'adresse `/chercher/Nothomb`. Votre nouvelle page de recherche se trouve à `/livres/recherche?q=Nothomb`. Pour que les visiteurs qui ont gardé l'ancienne adresse dans leurs favoris arrivent quand même sur la bonne page, vous redirigez l'ancienne adresse vers la nouvelle. En haut de `BookController`, ajoutez l'import `use Illuminate\Http\RedirectResponse;` au-dessus de celui de `Request`, puis ajoutez cette méthode :

```php
public function legacySearch(string $word): RedirectResponse
{
    return redirect()->route('books.search', ['q' => $word]);
}
```

Ajoutez la route après celle des données :

```php
Route::get('/chercher/{word}', [BookController::class, 'legacySearch']);
```

```bash
curl -s -I http://premier-test-laravel.test/chercher/Nothomb
```

```
HTTP/1.1 302 Found
…
Location: http://premier-test-laravel.test/livres/recherche?q=Nothomb
…
```

Le chemin de la route `books.search` ne contient aucun paramètre entre accolades. **Quand vous donnez à `route()` une valeur dont le nom n'apparaît pas dans le chemin, Laravel l'ajoute à la fin de l'adresse comme paramètre d'URL**, après un `?`. Pour voir l'adresse produite sans passer par une redirection, affichez-la avec Tinker, que vous connaissez depuis le chapitre MVC :

```bash
php artisan tinker --execute="echo route('books.search', ['q' => 'Nothomb']);"
```

```
http://premier-test-laravel.test/livres/recherche?q=Nothomb
```

Le `return` oublié

Un oubli fréquent consiste à écrire la valeur sans `return`. Faites l'essai : dans `index()`, effacez le mot `return` pour ne garder que `'Liste des livres';`, puis ouvrez `/livres`. La méthode déclare le type de retour `: string` mais ne renvoie rien. PHP lance donc une `TypeError`, et Laravel affiche une page d'erreur avec le code 500 :

```
App\Http\Controllers\BookController::index(): Return value must be of type string, none
returned
```

Retirez maintenant aussi `: string` et rechargez la page. Cette fois, PHP ne signale rien : Laravel envoie une page vide avec le code 200, et aucun message ne vous indique quelle ligne chercher. Remettez le `return` et le type de retour. **Écrivez le type de retour de chaque méthode** : PHP vérifie alors la valeur renvoyée, et le message d'erreur nomme la méthode fautive.

6. Le contrôleur à action unique

Certaines adresses ne déclenchent qu'une seule action. Au chapitre Routing, par exemple, l'adresse `/dernier-livre` redirige le visiteur vers la fiche du livre 12, et rien d'autre. Pour ce code, une classe qui ne contient qu'une méthode est suffisante. Une telle classe s'appelle un **contrôleur à action unique**.

Dans un tel contrôleur, l'unique méthode porte un nom imposé, `__invoke`. Pour créer la classe avec cette méthode déjà écrite, ajoutez l'option `--invokable` à la commande :

BASH
```
php artisan make:controller LatestBookController --invokable
```

```
   INFO  Controller [app/Http/Controllers/LatestBookController.php] created successfully.
```

PHP
```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class LatestBookController extends Controller
{
    /**
     * Handle the incoming request.
     */
    public function __invoke(Request $request)
    {
        //
    }
}
```

`__invoke` est une méthode magique, c'est-à-dire une méthode que PHP appelle lui-même dans une situation précise, comme `__construct` et `__toString`, vues au chapitre Objets et classes du bloc POO. PHP appelle `__invoke` quand vous utilisez un objet comme une fonction, en écrivant par exemple `$controller()`. Quand une route donne seulement le nom de la classe, Laravel complète l'action avec le nom `__invoke`, puis appelle cette méthode comme il appellerait `show()`. La route n'a donc pas besoin de nommer de méthode.

Le commentaire `Handle the incoming request.` signifie « traite la requête reçue ». Pour rediriger vers le livre 12, la méthode n'a pas besoin de lire la requête : vous pouvez retirer le paramètre `Request $request` et son import. Remplacez le contenu du fichier par ceci :

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;

class LatestBookController extends Controller
{
    public function __invoke(): RedirectResponse
    {
        return redirect()->route('books.show', 12);
    }
}
```

Dans `routes/web.php`, ajoutez l'import de `LatestBookController` juste au-dessus de celui de `BookController`, puis remplacez la route `/dernier-livre` et sa fonction anonyme par cette ligne. La route donne la classe seule, sans tableau ni nom de méthode :

```php
Route::get('/dernier-livre', LatestBookController::class);
```

```bash
curl -s -I http://premier-test-laravel.test/dernier-livre
php artisan route:list --path=dernier
```

```
HTTP/1.1 302 Found
…
Location: http://premier-test-laravel.test/livres/12
…

  GET|HEAD       dernier-livre ..............................................
LatestBookController

                                                                  Showing [1]
routes
```

La réponse n'a pas changé. `route:list` affiche le nom de la classe sans `@` ni méthode, ce qui signale un contrôleur à action unique. Si la route a un paramètre, comme `{id}`, `__invoke` le reçoit comme n'importe quelle méthode, par exemple avec `__invoke(int $id)`. La contrainte s'écrit alors après la classe, comme après un tableau : `->whereNumber('id')`.

7. Ranger un contrôleur dans un sous-dossier

Quand un projet compte beaucoup de contrôleurs, vous pouvez ranger ceux des pages d'administration dans un sous-dossier, `Admin`. Pour créer un contrôleur dans ce sous-dossier, écrivez le nom du dossier devant celui de la classe, séparé par une barre oblique :

```bash
php artisan make:controller Admin/StatisticController
```

```
   INFO  Controller [app/Http/Controllers/Admin/StatisticController.php] created
successfully.
```

```php
<?php

namespace App\Http\Controllers\Admin;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class StatisticController extends Controller
{
    //
}
```

Artisan a créé le dossier `Admin/` et a écrit le namespace `App\Http\Controllers\Admin`, qui se termine par le nom de ce dossier, comme le demande la règle PSR-4. Il a aussi ajouté la ligne `use App\Http\Controllers\Controller;`. Sans elle, PHP chercherait la classe mère dans le même namespace que le contrôleur, et la page afficherait `Class "App\Http\Controllers\Admin\Controller" not found`.

Écrivez une méthode `index()` qui renvoie une phrase d'exemple :

```php
public function index(): string
{
    return 'Statistiques : 1 240 livres, 87 prêts en cours';
}
```

Dans `routes/web.php`, l'import contient le nom complet de la classe, avec `Admin`. Ajoutez-le en tête de la liste des imports, puis ajoutez la route à la fin du fichier :

```php
use App\Http\Controllers\Admin\StatisticController;

// …

Route::get('/admin/statistiques', [StatisticController::class, 'index']);
```

```bash
curl -s http://premier-test-laravel.test/admin/statistiques
php artisan route:list --path=admin
```

```
Statistiques : 1 240 livres, 87 prêts en cours

  GET|HEAD       admin/statistiques ..............................
Admin\StatisticController@index

                                                                  Showing [1]
routes
```

`route:list` affiche le nom de la classe sans le début commun `App\Http\Controllers\`, ici `Admin\StatisticController`. La partie `Admin\` vous indique le sous-dossier où ouvrir le fichier.

Le sous-dossier range le code, il ne protège pas la page : n'importe quel visiteur peut ouvrir `/admin/statistiques`. Vous limiterez l'accès à ce type de page dans le cours « Projet Web Dynamique », qui ajoute les comptes utilisateurs.

8. Quand ça ne marche pas

Trois erreurs reviennent souvent quand vous créez un contrôleur et le reliez à une route. Les deux premières s'affichent dans le navigateur : ouvrez l'adresse de la route concernée, et la page d'erreur de Laravel donne le nom de l'exception, puis une phrase qui en indique la cause. La troisième s'affiche dans le terminal.

L'import oublié. Retirez la ligne `use App\Http\Controllers\BookController;` de `routes/web.php`, puis ouvrez `/livres`. Laravel renvoie une page d'erreur avec le code 500 et ce message :

```
Target class [BookController] does not exist.
```

Sans l'import, `BookController::class` vaut simplement la chaîne `"BookController"`, sans le namespace. Laravel cherche donc une classe qui porte exactement ce nom, et aucune n'existe. `route:list` échoue lui aussi, pour la même raison :

```bash
php artisan route:list --path=livres
```

```
   ReflectionException

  Class "BookController" does not exist
…
```

Remettez l'import.

La méthode absente. Dans la route `books.index`, remplacez `'index'` par `'list'`, un nom qui n'existe pas dans `BookController`, puis ouvrez `/livres`. Laravel renvoie une erreur 500 :

```
Call to undefined method App\Http\Controllers\BookController::list()
```

Cette fois, `route:list` ne signale rien. Il affiche `books.index › BookController@list` sans vérifier que la méthode existe. Seule une requête vers la page révèle l'erreur. Remettez `'index'`.

Le contrôleur déjà créé. Lancez une seconde fois `make:controller` avec le même nom :

BASH

```
php artisan make:controller BookController
```

```
   ERROR  Controller already exists.
```

Artisan n'écrit rien, et votre fichier reste intact. Avec l'option `--force`, Artisan remplacerait votre fichier par un contrôleur vide, et toutes vos méthodes seraient perdues.

Si vous lancez `php artisan make:controller` sans nom, Artisan vous pose des questions, en commençant par `What should the controller be named?`, c'est-à-dire « Quel nom donner au contrôleur ? ». Dans ce cours, écrivez toujours le nom dans la commande : vous voyez ainsi dans votre historique exactement ce que vous avez créé.

9. Et `Route::resource()` ?

Au chapitre Routing, vous avez lu qu'une seule ligne, `Route::resource()`, peut déclarer toutes les routes d'une ressource, c'est-à-dire d'un type de données que le site affiche, crée, modifie et supprime, comme des recettes. Chacune de ces routes appelle une méthode du même contrôleur, et Laravel impose le nom de ces sept méthodes. Pour créer un contrôleur qui les contient déjà, ajoutez l'option `--resource` :

BASH

```
php artisan make:controller RecipeController --resource
```

```
   INFO  Controller [app/Http/Controllers/RecipeController.php] created successfully.
```

Le fichier contient sept méthodes vides : `index`, `create`, `store`, `show`, `edit`, `update` et `destroy`. Les méthodes qui travaillent sur une recette précise reçoivent son identifiant dans `string $id`. Celles qui traitent un formulaire envoyé reçoivent la requête dans `Request $request`, pour en lire les champs. Dans `routes/web.php`, ajoutez l'import de `RecipeController` juste sous celui de `PostController`. Remplacez ensuite les deux routes du chapitre Routing, `/recettes/{id}` et `/recettes/nouvelle`, par cette ligne :

```php
Route::resource('recettes', RecipeController::class);
```

```bash
php artisan route:list --path=recettes
```

```
  GET|HEAD        recettes ............................. recettes.index ›
RecipeController@index
  POST            recettes ............................. recettes.store ›
RecipeController@store
  GET|HEAD        recettes/create ...................... recettes.create ›
RecipeController@create
  GET|HEAD        recettes/{recette} ....................... recettes.show ›
RecipeController@show
  PUT|PATCH       recettes/{recette} ................... recettes.update ›
RecipeController@update
  DELETE          recettes/{recette} .................. recettes.destroy ›
RecipeController@destroy
  GET|HEAD        recettes/{recette}/edit .................. recettes.edit ›
RecipeController@edit

                                                                  Showing [7]
routes
```

Les sept noms suivent la convention `ressource.action` du chapitre Routing, de `recettes.index` à `recettes.destroy`. Laravel déclare `recettes/create` avant `recettes/{recette}`. Ainsi, l'adresse `/recettes/create` correspond à la route du formulaire, et le mot `create` n'est jamais pris pour la valeur de `{recette}`. C'est le piège que vous avez rencontré au chapitre Routing avec `/recettes/nouvelle`.

Laravel fabrique le paramètre `{recette}` et les sept noms de routes à partir du segment d'adresse `recettes` : c'est pourquoi ils restent en français, alors que le reste de votre code est en anglais. La méthode `show()` reçoit ce paramètre dans `$id` : Laravel donne les valeurs dans l'ordre du chemin, comme à la section 3. Les sept méthodes sont encore vides. L'adresse `/recettes/nouvelle` du chapitre Routing appelle donc maintenant `show()`, et la page reste blanche. Vous remplirez ces méthodes au chapitre 16, avec un modèle et des formulaires.

À retenir

- `php artisan make:controller NameController` crée la classe. Une route la désigne avec `[NameController::class, 'method']`, à condition d'importer la classe en haut de `routes/web.php`.
- Une méthode reçoit les paramètres de route dans l'ordre du chemin, et `Request $request` lui donne la requête.

- Une méthode peut renvoyer une chaîne, une vue, une redirection ou un tableau, que Laravel convertit en JSON. Grâce au type de retour, PHP signale une erreur si vous oubliez le `return`.
- Un contrôleur à action unique porte une méthode `__invoke`, et sa route donne la classe seule, sans tableau.
- `Target class … does not exist` signale un import manquant. `Call to undefined method` signale une route qui nomme une méthode absente du contrôleur, une erreur que `route:list` ne détecte pas.

Exercices

Comptez 70 minutes pour les exercices, en séance ou à la maison.

Faites les exercices 1 à 3 dans `premier-test-laravel`. Ajoutez chaque route à la fin de `routes/web.php` et chaque import en haut du fichier.

1. Le cinéma (20 minutes)

Créez le contrôleur du cinéma avec Artisan :

```bash
php artisan make:controller MovieController
```

Écrivez trois méthodes qui renvoient une chaîne et reliez-les à trois routes nommées. La route de la fiche et celle des séances n'acceptent que des chiffres pour le numéro du film. Inspirez-vous des sections 2 et 3.

Méthode	Route	Nom	Réponse pour le film 4
`index()`	`GET /films`	`movies.index`	`Trois films à l'affiche cette semaine.`
`show()`	`GET /films/{id}`	`movies.show`	`Fiche du film 4`
`showtimes()`	`GET /films/{id}/seances`	`movies.showtimes`	`Séances du film 4 : 14 h, 17 h et 20 h 30`

La phrase de `index()` contient une apostrophe. Écrivez-la entre guillemets doubles, `"Trois films à l'affiche cette semaine."`. Entre guillemets simples, PHP croit que la chaîne s'arrête après `l` et lance une `ParseError`, une erreur de syntaxe qui fait échouer les trois routes de `MovieController`.

Vérifiez les quatre adresses et la liste des routes :

```bash
curl -s http://premier-test-laravel.test/films
curl -s http://premier-test-laravel.test/films/4
curl -s http://premier-test-laravel.test/films/4/seances
curl -s -i http://premier-test-laravel.test/films/abc
php artisan route:list --path=films
```

Avant de relancer `php artisan make:controller MovieController`, écrivez ce que vous attendez, puis lancez la commande et comparez. Que ferait la même commande avec l'option `--force` ? Ne la lancez pas.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/controllers

2. Le parking (15 minutes)

Imaginez un parking qui coûte 1,50 EUR de l'heure, sans jamais dépasser 12 EUR pour la journée. Créez un contrôleur à action unique `ParkingRateController` et reliez-le à la route `GET /parking/{hours}`, qui n'accepte que des chiffres. Inspirez-vous de la section 6.

La méthode `__invoke` reçoit le nombre d'heures et renvoie une phrase. Artisan écrit `__invoke(Request $request)` : remplacez ce paramètre par `int $hours`, comme `show(int $id)` à la section 3. Pour le calcul, utilisez deux fonctions de PHP. `min($hours * 1.5, 12)` renvoie la plus petite des deux valeurs, donc le prix plafonné. `number_format($price, 2, ',', ' ')` écrit un nombre avec deux décimales et une virgule, comme `4,50`.

```bash
curl -s http://premier-test-laravel.test/parking/3
curl -s http://premier-test-laravel.test/parking/10
```

```
3 heure(s) de parking : 4,50 EUR
10 heure(s) de parking : 12,00 EUR
```

Modifiez maintenant la route pour qu'elle désigne `[ParkingRateController::class, 'index']`. Quel message obtenez-vous sur `/parking/3`, et pourquoi ? Rétablissez la route sans tableau avant de passer à la suite.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/controllers

3. La météo (25 minutes)

Créez un contrôleur `WeatherController`. Il garde des relevés inventés dans une propriété :

```php
private array $readings = ['wavre' => 15, 'namur' => 14, 'liege' => 13];
```

Écrivez trois méthodes et leurs routes. Inspirez-vous des sections 4 et 5.

Méthode	Route	Réponse
`index(Request $request): string`	`GET /meteo`, nommée `weather.index`	`Météo demandée pour namur` avec `?ville=namur`, `Météo demandée pour wavre` sans paramètre

Méthode	Route	Réponse
`reading(string $city): array`	`GET /meteo/{city}/releve`, pour `wavre`, `namur` et `liege` seulement	`{"city":"liege","temperature":13,"unit":"C"}` pour `liege`
`today(): RedirectResponse`	`GET /meteo/aujourdhui`	une redirection 302, avec `Location: http://premier-test-laravel.test/meteo?ville=namur`

Comme à la section 5, ajoutez l'import `use Illuminate\Http\RedirectResponse;` en haut de `WeatherController`. Sans lui, PHP cherche une classe `App\Http\Controllers\RedirectResponse`, et `/meteo/aujourdhui` répond par une erreur 500. La page affiche une `TypeError` avec ce message :

```
App\Http\Controllers\WeatherController::today(): Return value must be of type
App\Http\Controllers\RedirectResponse, Illuminate\Http\RedirectResponse returned
```

Pour lire la propriété dans une méthode, écrivez $this->readings[\texttt{city}]$. Vérifiez vos trois routes :

```bash
curl -s "http://premier-test-laravel.test/meteo?ville=namur"
curl -s http://premier-test-laravel.test/meteo
curl -s -i http://premier-test-laravel.test/meteo/liege/releve
curl -s -I http://premier-test-laravel.test/meteo/aujourdhui
```

Retirez enfin la contrainte de la route du relevé et demandez `/meteo/paris/releve`. Quel code obtenez-vous, et quelle ligne de `reading()` en est la cause ? Remettez ensuite la contrainte.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/controllers

4. Lire les routes et les erreurs (sur papier, 10 minutes)

À la fin des exercices, `php artisan route:list --except-vendor` affiche entre autres ces six lignes. Chez vous, le numéro de ligne de `ping` peut être différent.

```
  GET|HEAD        admin/statistiques ..............................
Admin\StatisticController@index
  GET|HEAD        films/{id} ..................................... movies.show ›
MovieController@show
  GET|HEAD        livres/{id}/donnees .....................................
BookController@data
  GET|HEAD        meteo ..................................... weather.index ›
WeatherController@index
  GET|HEAD        parking/{hours} .........................................
ParkingRateController
  GET|HEAD        ping ...................................................
routes/web.php:19
```

1. Pour chaque ligne, écrivez le fichier à ouvrir et la méthode, ou la fonction, que Laravel appelle.

2. Pour chacun de ces trois messages, écrivez la cause et la correction.
   - `Target class [MovieController] does not exist.`
   - `Call to undefined method App\Http\Controllers\WeatherController::readings()`
   - `App\Http\Controllers\MovieController::index(): Return value must be of type string, none returned`

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/controllers

5. Six cartes

Quelle commande crée `app/Http/Controllers/MovieController.php` ?

`php artisan make:controller MovieController`.

Que contient la classe `Controller` dont héritent vos contrôleurs ?

Rien. C'est une classe abstraite vide, prévue pour des méthodes communes à tous vos contrôleurs.

Comment s'écrit la route d'un contrôleur à action unique ?

Sans tableau : `Route::get('/parking/{hours}', ParkingRateController::class);`.

Comment lire `?q=Nothomb` dans une méthode ?

Recevoir `Request $request`, puis écrire `$request->query('q')`.

Que renvoie Laravel quand une méthode renvoie un tableau PHP ?

Une réponse JSON, avec l'en-tête `Content-Type: application/json`.

`Target class [MovieController] does not exist.` : que vérifiez-vous ?

La ligne `use App\Http\Controllers\MovieController;` en haut de `routes/web.php`.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/controllers

3.9 Views

Bloc Laravel · séance 3 · ~60 min

Slides du chapitre (PDF) PDF

Au chapitre MVC, la vue `posts/index.blade.php` affichait des articles avec `@foreach` et `{{ }}`. Un site compte le plus souvent plusieurs pages, qui répètent le même `<head>`, le même en-tête et le même pied de page. Dans ce chapitre, vous construisez le site d'un festival de musique inventé, « Les Nuits du Parc ». Vous apprenez à écrire ces lignes communes une seule fois, dans un fichier que toutes les pages réutilisent. Vous travaillez dans le projet `premier-test-laravel`, sans base de données.

À la fin de ce chapitre, vous serez capables de :

1. **afficher** les données d'un contrôleur avec `@if`, `@foreach`, `$loop` et `@forelse`,
2. **créer** un composant Blade, un morceau de template réutilisable, et lui transmettre des données,
3. **écrire** le layout d'un site, le template que toutes ses pages partagent, et l'appliquer à plusieurs pages,
4. **convertir** un layout écrit avec `@extends`, `@section`, `@yield` et `@include` en composants,
5. **trouver** la cause d'une erreur de vue, de variable ou de composant introuvable, et **protéger** un formulaire avec `@csrf`.

1. Trois pages qui se ressemblent

Commencez par la page des infos pratiques. La commande `make:view` crée le fichier d'une vue. Vous lui donnez le nom de la vue, avec un point à la place de la barre oblique : `festival.info` désigne le fichier `info.blade.php` du dossier `festival`. Si le dossier n'existe pas, la commande le crée aussi :

```bash
php artisan make:view festival.info
```

```
   INFO  View [resources/views/festival/info.blade.php] created successfully.
```

Le fichier créé contient une balise `<div>` et un commentaire HTML avec une citation tirée au hasard, qui change à chaque création :

```xml
<div>
    <!-- When there is no desire, all things are at peace. - Laozi -->
</div>
```

Un commentaire HTML fait partie de la page envoyée au navigateur, et tout visiteur peut le lire en affichant le code source. Remplacez tout le contenu du fichier par cette page complète :

```xml
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Infos pratiques · Les Nuits du Parc</title>
</head>
<body>
    <header>
        <p>Les Nuits du Parc</p>
        <nav>
            <a href="/concerts">Programme</a>
            <a href="/infos">Infos pratiques</a>
            <a href="/concerts/3">Concert de clôture</a>
        </nav>
    </header>
    <main>
        <h1>Infos pratiques</h1>
        <p>Le festival a lieu au parc communal, du vendredi au dimanche.</p>
    </main>
    <footer>
        <p>Les Nuits du Parc, édition 2026</p>
    </footer>
</body>
</html>
```

Cette page n'affiche aucune donnée venue d'un contrôleur. Vous la reliez donc à son adresse avec `Route::view`, qui renvoie directement la vue, sans passer par un contrôleur. Ajoutez cette route à la fin de `routes/web.php`, qui contient déjà les routes des chapitres Routing et Controllers :

```php
Route::view('/infos', 'festival.info')->name('info');
```

```bash
curl -s http://premier-test-laravel.test/infos
```

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Infos pratiques · Les Nuits du Parc</title>
…
    <main>
        <h1>Infos pratiques</h1>
        <p>Le festival a lieu au parc communal, du vendredi au dimanche.</p>
    </main>
…
</html>
```

Le site aura deux autres pages : le programme des concerts et la fiche d'un concert. Si vous les écriviez de la même façon, les trois fichiers répéteraient une vingtaine de lignes identiques. Seuls le `<title>` et le contenu de `<main>` changeraient d'une page à l'autre.

Le code qui sent mauvais

En binôme, cinq minutes. Le festival change de nom pour « Parc en Musique », et vous devez ajouter au menu un lien « Devenir bénévole ». Avec trois pages écrites comme `festival/info.blade.php`, quels fichiers ouvrez-vous, et combien de lignes modifiez-vous ?

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/views

Pour éviter de recopier ces lignes, les vues des concerts que vous écrivez maintenant ne contiennent que leur propre contenu, sans `<!DOCTYPE html>` ni en-tête. Le navigateur affiche tout de même un fragment de HTML qui n'a ni `<html>` ni `<head>` : il complète lui-même la structure manquante.

2. Passer des données à une vue

Vous connaissez depuis le chapitre MVC la fonction `view()`, son tableau de données et l'affichage échappé `{{ }}`. Au chapitre Routing, `Route::view` était réservé aux pages sans données. La page du programme des concerts, elle, affiche des données : sa route appelle donc un contrôleur. Créez-le avec Artisan :

```bash
php artisan make:controller ConcertController
```

```
   INFO  Controller [app/Http/Controllers/ConcertController.php] created successfully.
```

Artisan a écrit une classe vide qui importe `Request`. Remplacez tout le contenu du fichier. Les concerts, inventés, sont rangés dans une propriété, comme les relevés de `WeatherController` au chapitre Controllers. La méthode `index()` renvoie une `View` : l'import `use Illuminate\View\View;` remplace celui de `Request`.

```php
<?php

namespace App\Http\Controllers;

use Illuminate\View\View;

class ConcertController extends Controller
{
    private array $concerts = [
        1 => [
            'artist' => 'Duo Mistral',
            'stage' => 'Chapiteau',
            'time' => '18 h',
            'duration' => 45,
            'description' => 'Guitare et violon, <strong>entrée libre</strong>.',
            'guests' => [],
        ],
        2 => [
            'artist' => "L'Orchestre du Lac",
            'stage' => 'Grande scène',
            'time' => '20 h',
            'duration' => 90,
            'description' => 'Un orchestre de trente musiciens, <strong>gratuit</strong>
pour les moins de 12 ans.',
            'guests' => ['Ana Lopes', 'Karim Benali'],
        ],
        3 => [
            'artist' => 'Orage Mécanique',
            'stage' => 'Grande scène',
            'time' => '22 h',
            'duration' => 75,
            'description' => 'Rock et percussions pour la <strong>clôture</strong>.',
            'guests' => ['Les Tambours du Parc'],
        ],
    ];

    public function index(): View
    {
        return view('concerts.index', ['concerts' => $this->concerts]);
    }
}
```

La clé de chaque concert, de 1 à 3, lui sert de numéro. Créez la vue avec `php artisan make:view concerts.index`, puis remplacez son contenu par ce template :

```xml
<h1>Programme</h1>
<ul>
    @foreach (concertsasconcerts as concertsasconcert)
        <li>{{ concert['artist'] }}, {{ concert['stage'] }}, {{ $concert['time'] }}</li>
    @endforeach
</ul>
```

Dans `routes/web.php`, importez `ConcertController` au-dessus de `LatestBookController`, puis ajoutez la route à la fin du fichier :

```php
Route::get('/concerts', [ConcertController::class, 'index'])->name('concerts.index');
```

```bash
curl -s http://premier-test-laravel.test/concerts
```

```
<h1>Programme</h1>
<ul>
            <li>Duo Mistral, Chapiteau, 18 h</li>
            <li>L&amp;#039;Orchestre du Lac, Grande scène, 20 h</li>
            <li>Orage Mécanique, Grande scène, 22 h</li>
    </ul>
```

Un concert est un tableau, et non un objet comme `$post` au chapitre MVC : vous écrivez donc `$concert['artist']`, avec des crochets. `{{ }}` a remplacé l'apostrophe de « L'Orchestre » par `'`, le code HTML de ce caractère. C'est l'échappement vu au chapitre MVC : le navigateur affiche une apostrophe, mais ne peut pas la prendre pour du code.

Dans du code existant, vous rencontrerez aussi `compact('concerts')`, une fonction PHP qui fabrique le tableau `['concerts' ⇒ $concerts]` à partir d'une variable `$concerts`, et la méthode `→with('concerts', …)`. Ces deux écritures produisent la même page. Dans ce cours, vous écrivez le tableau, car il montre sur une seule ligne le nom de chaque variable et sa valeur.

La vue introuvable

Si le nom donné à `view()` ne correspond à aucun fichier, par exemple `view('concerts.program', …)`, Laravel renvoie une erreur 500 avec ce message :

```
View [concerts.program] not found.
```

Laravel cherche la vue dans `resources/views/`, en remplaçant chaque point par une barre oblique : `concerts.program` désigne `concerts/program.blade.php`, un fichier qui n'existe pas. Remettez `concerts.index`.

3. Les directives de tous les jours

Une **directive** est une instruction Blade qui commence par `@`, comme `@foreach`. Vous allez en utiliser plusieurs dans une nouvelle page, la fiche d'un concert. Dans `ConcertController`, ajoutez `show()` sous `index()` :

```php
public function show(int $id): View
{
    if (! isset($this->concerts[$id])) {
        abort(404);
    }

    return view('concerts.show', ['concert' => $this->concerts[$id]]);
}
```

La fonction `abort(404)` arrête le traitement et renvoie la page 404 de Laravel. Sans cet appel, pour l'adresse `/concerts/9`, PHP chercherait la clé `9` dans `$this->concerts`, ne la trouverait pas, et Laravel renverrait une erreur 500 au lieu d'une page 404. Ajoutez la route à la fin de `routes/web.php` :

```php
Route::get('/concerts/{id}', [ConcertController::class, 'show'])->whereNumber('id')
->name('concerts.show');
```

Créez la vue avec `php artisan make:view concerts.show` et écrivez ce template :

```xml
{{-- Ce commentaire Blade n'arrive pas dans la page --}}
<!-- Ce commentaire HTML arrive dans la page -->
<h1>{{ concert['artist'] }}</h1>
<p>{{ concert['stage'] }}, {{ concert['time'] }}</p>

@if (concert['duration'] ⩾ 90)
    <p>Long concert : {{ concert['duration'] }} minutes, avec entracte.</p>
@elseif (concert['duration'] ⩾ 60)
    <p>Concert de {{ concert['duration'] }} minutes, sans entracte.</p>
@else
    <p>Concert court : {{ concert['duration'] }} minutes.</p>
@endif

<p>{{ concert['description'] }}</p>
<p>{!! concert['description'] !!}</p>

<h2>Invités</h2>
<ul>
    @forelse (concert[’guests’]asconcert['guests'] as concert[’guests’]asguest)
        <li>{{ $guest }}</li>
    @empty
        <li>Aucun invité annoncé</li>
    @endforelse
</ul>
```

```bash
curl -s http://premier-test-laravel.test/concerts/2
```

```
<!— Ce commentaire HTML arrive dans la page —>
<h1>L&#039;Orchestre du Lac</h1>
<p>Grande scène, 20 h</p>

    <p>Long concert : 90 minutes, avec entracte.</p>

<p>Un orchestre de trente musiciens, &lt;strong&gt;gratuit&lt;/strong&gt; pour les moins de
12 ans.</p>
<p>Un orchestre de trente musiciens, <strong>gratuit</strong> pour les moins de 12 ans.</p>

<h2>Invités</h2>
<ul>
            <li>Ana Lopes</li>
            <li>Karim Benali</li>
    </ul>
```

Lisez la réponse de haut en bas :

- **Les commentaires.** Blade retire le commentaire `{{-- … --}}` avant d'envoyer la page : il n'apparaît pas dans la réponse. Le commentaire HTML `<!— … —>` y reste, et tout visiteur peut le lire.
- `@if`, `@elseif`, `@else`, `@endif` fonctionnent comme le `if` de PHP. Pour un concert de 90 minutes, la condition de `@if` est vraie : Blade n'affiche que le premier paragraphe.
- `{!! !!}` **affiche sans échapper.** Avec `{{ }}`, la balise `<strong>` est envoyée sous la forme `<strong>`, et le navigateur affiche les caractères `<strong>` comme du texte. Avec `{!! !!}`, la balise arrive intacte et le mot passe en gras.

N'utilisez `{!! !!}` que pour du HTML que vous avez écrit vous-même, comme ces descriptions. Affichez toujours un texte saisi par un visiteur avec `{{ }}`. Avec `{!! !!}`, un `<script>` qu'il aurait tapé serait exécuté dans le navigateur de chaque personne qui ouvre la page. C'est l'attaque que l'échappement empêchait déjà au chapitre MVC.

`@forelse` parcourt le tableau comme `@foreach`, et affiche le bloc placé après `@empty` quand le tableau est vide, ici pour le premier concert :

BASH

```
curl -s http://premier-test-laravel.test/concerts/1
```

```
…
    <p>Concert court : 45 minutes.</p>
…
<h2>Invités</h2>
<ul>
            <li>Aucun invité annoncé</li>
    </ul>
```

La variable `$loop`

Dans une boucle `@foreach`, Blade crée une variable `$loop`, un objet qui décrit le tour en cours. `$loop->iteration` donne le numéro du tour, en commençant à 1, et `$loop->count` le nombre total de tours. `$loop->first` vaut `true` au premier tour, et `$loop->last` au dernier. La variable `$loop` existe aussi dans une boucle `@forelse`.

Remplacez le contenu de `concerts/index.blade.php` :

```xml
<h1>Programme</h1>
<ul>
    @foreach (concertsasconcerts as concertsasid => concert)
        <li>
            {{ $loop->iteration }}/{{ $loop->count }}
            <a href="{{ route('concerts.show', $id) }}">{{ $concert['artist'] }}</a>
            @if (loop->last)
                (clôture)
            @endif
        </li>
    @endforeach
</ul>
```

```bash
curl -s http://premier-test-laravel.test/concerts
```

```
<h1>Programme</h1>
<ul>
            <li>
            1/3
            <a href="http://premier-test-laravel.test/concerts/1">Duo Mistral</a>
                    </li>
…
            <li>
            3/3
            <a href="http://premier-test-laravel.test/concerts/3">Orage Mécanique</a>
                        (clôture)
                    </li>
    </ul>
```

L'écriture `id =>concert` donne à chaque tour la clé et la valeur, comme en PHP. `route('concerts.show', $id)` place cette clé dans l'adresse de la fiche, par exemple `/concerts/1`.

La variable oubliée

Dans `show()`, remplacez un instant le `return` par `return view('concerts.show');`, sans tableau. Ouvrez `/concerts/2` : Laravel renvoie une erreur 500 avec ce message :

```
Undefined variable $concert (View: …/resources/views/concerts/show.blade.php)
```

Le message nomme le template, mais la cause est dans le contrôleur, qui n'a pas transmis la variable. Remettez le tableau.

4. Ce que Blade fait de votre fichier

`{{ }}` et `@if` ne font pas partie du langage PHP : PHP ne peut pas exécuter un template tel quel. La première fois qu'une page a besoin d'un template, Blade le traduit en un fichier PHP ordinaire, appelé **vue compilée**. Il l'enregistre dans `storage/framework/views/`, sous un nom de 32 caractères. Voici un extrait de celle de `concerts/show.blade.php` :

```php
<h1><?php echo e(concert['artist']);?></h1>…<?phpif(concert['artist']); ?></h1>
…
<?php if(concert['artist']);?></h1>…<?phpif(concert['duration'] ⩾ 90): ?>
…
<p><?php echo e(concert['description']);?></p><p><?phpechoconcert['description']); ?></p>
<p><?php echo concert['description']);?></p><p><?phpechoconcert['description']; ?></p>
```

`{{ }}` devient un `echo` qui passe par la fonction `e()`, celle qui transforme les caractères spéciaux du HTML, comme `<`, `>` et l'apostrophe, en codes HTML. `{!! !!}` devient un `echo` sans `e()`.

Aux requêtes suivantes, Laravel exécute directement la vue compilée. Quand vous modifiez le template, Laravel remarque que le fichier est plus récent que sa vue compilée, et il le traduit de nouveau. Si la page affiche malgré tout l'ancienne version, supprimez toutes les vues compilées avec cette commande :

```bash
php artisan view:clear
```

```
   INFO  Compiled views cleared successfully.
```

5. Un premier composant

Sur la page des infos pratiques, vous voulez mettre certains messages en évidence dans un cadre. Sur la page du programme, chaque concert s'affichera dans un petit bloc, une carte. Vous utiliserez ces deux morceaux de HTML sur plusieurs pages.

Un **composant** Blade est un morceau de template que vous écrivez une fois et réutilisez dans plusieurs vues. Son fichier se range dans `resources/views/components/`.

Pour l'afficher dans une vue, vous écrivez une balise qui porte son nom précédé de `x-`, par exemple `<x-alert>` pour le fichier `alert.blade.php`.

Créer le composant et son slot

Pour créer un composant, lancez `make:component` avec l'option `--view`. Artisan crée alors un seul fichier, le template du composant :

```bash
php artisan make:component alert --view
```

```
   INFO  View [resources/views/components/alert.blade.php] created successfully.
```

Remplacez le `<div>` et sa citation par ceci :

```xml
<div class="alert">
    {{ $slot }}
</div>
```

Dans `festival/info.blade.php`, appelez le composant sous le `<h1>` :

```xml
<h1>Infos pratiques</h1>
<x-alert>Les portes ouvrent à 17 h 30.</x-alert>
```

```
    <main>
        <h1>Infos pratiques</h1>
        <div class="alert">
    Les portes ouvrent à 17 h 30.
</div>
        <p>Le festival a lieu au parc communal, du vendredi au dimanche.</p>
    </main>
```

Blade a remplacé la balise `<x-alert>` par le contenu de `components/alert.blade.php`, et le texte écrit entre les deux balises est arrivé à la place de `{{ $slot }}`. Ce contenu s'appelle le **slot**.

Si vous modifiez `alert.blade.php`, toutes les pages qui utilisent `<x-alert>` changent en même temps. Et en lisant `<x-alert>Les portes ouvrent à 17 h 30.</x-alert>` dans une vue, vous comprenez ce que la page affiche sans ouvrir le composant.

Dans ce cours, un composant est un seul fichier de template, sans classe PHP. Laravel l'appelle **composant anonyme**, parce qu'aucune classe ne porte son nom.

Les props et leur valeur par défaut

Une alerte qui annonce un concert complet doit avoir un autre aspect qu'une simple information, donc une autre classe CSS. La vue indique au composant de quel type d'alerte il s'agit avec un attribut écrit dans la balise, comme `href` dans `<a href="…">`. Une donnée transmise ainsi s'appelle une **prop**, et le composant la déclare avec la directive `@props`. Remplacez le contenu de `alert.blade.php` :

```xml
@props(['type' => 'info'])

<div class="alert alert-{{ type }}">
    {{ slot }}
</div>
```

`'info'` est la valeur par défaut de `type` : Blade l'utilise quand la balise `<x-alert>` ne contient pas d'attribut `type`. Ajoutez une seconde alerte dans `festival/info.blade.php` :

```xml
<x-alert>Les portes ouvrent à 17 h 30.</x-alert>
<x-alert type="danger">Le concert de 22 h est complet.</x-alert>
```

```
        <div class="alert alert-info">
    Les portes ouvrent à 17 h 30.
</div>
        <div class="alert alert-danger">
    Le concert de 22 h est complet.
</div>
```

Les données que le composant accepte sont maintenant écrites en tête de son fichier : en l'ouvrant, vous voyez ce qu'il faut lui transmettre.

Les deux-points devant un attribut

Sur la page du programme, affichez maintenant chaque concert dans une carte. Créez le composant avec `php artisan make:component concert-card --view`, puis écrivez ce contenu :

```xml
@props(['artist', 'time'])

<article class="concert-card">
    <h2>{{ artist }}</h2>
    <p>{{ time }}</p>
</article>
```

Le nom de la balise reprend celui du fichier : pour afficher `concert-card.blade.php`, vous écrivez `<x-concert-card>`. Dans `concerts/index.blade.php`, remplacez la liste par un premier essai :

```xml
<h1>Programme</h1>
@foreach (concertsasconcerts as concertsasid => concert)<x-concert-cardartist="concert)
    <x-concert-card artist="concert)<x-concert-cardartist="concert['artist']"
time="$concert['time']" />
@endforeach
```

```
<h1>Programme</h1>
    <article class="concert-card">
    <h2>concert[&#039;artist&#039;]</h2>
    <p>concert[&#039;time&#039;]</p>
</article>
…
```

Le composant a reçu le texte `$concert['artist']` tel quel. **Pour passer la valeur d'une expression PHP, écrivez deux-points devant le nom de l'attribut.** Corrigez l'appel :

```xml
<x-concert-card :artist="concert['artist']":time="concert['artist']"
:time="concert['artist']":time="concert['time']" />
```

```
<h1>Programme</h1>
    <article class="concert-card">
    <h2>Duo Mistral</h2>
    <p>18 h</p>
</article>
    <article class="concert-card">
    <h2>L&#039;Orchestre du Lac</h2>
    <p>20 h</p>
</article>
…
```

Deux erreurs de composant

Retirez `:time="…"` de l'appel. La prop `time` n'a pas de valeur par défaut, et Laravel renvoie une erreur 500 :

```
Undefined variable $time (View: …/resources/views/components/concert-card.blade.php) (View:
…/resources/views/components/concert-card.blade.php)
```

Le message nomme deux fois le template du composant, parce que c'est lui qui affiche `$time`. La cause est pourtant dans la page qui l'appelle : `concerts/index.blade.php` ne transmet plus l'attribut `time`. Remettez `:time="…"`. Une valeur par défaut dans `@props` ferait disparaître l'erreur, mais cacherait l'oubli.

Dans la boucle, remplacez ensuite `<x-concert-card` par `<x-card`. Blade cherche alors le fichier `components/card.blade.php`, qui n'existe pas, et Laravel affiche :

```
Unable to locate a class or view for component [card].
```

Remettez `<x-concert-card`.

Passer le tableau entier

Au lieu de transmettre chaque information dans sa propre prop, vous pouvez transmettre tout le tableau du concert dans une seule prop. Le composant choisit ensuite ce qu'il affiche. Remplacez le contenu de `concert-card.blade.php` et la boucle de `concerts/index.blade.php` :

```xml
@props(['concert'])

<article class="concert-card">
    <h2>{{ concert['artist'] }}</h2>
    <p>{{ concert['stage'] }}, {{ $concert['time'] }}</p>
</article>
```

```xml
<h1>Programme</h1>
@foreach (concertsasconcerts as concertsasid ⇒ concert)<x-concert-card:concert="concert)
    <x-concert-card :concert="concert)<x-concert-card:concert="concert" />
    <a href="{{ route('concerts.show', $id) }}">Voir la fiche</a>
@endforeach
```

```
<h1>Programme</h1>
    <article class="concert-card">
    <h2>Duo Mistral</h2>
    <p>Chapiteau, 18 h</p>
</article>
    <a href="http://premier-test-laravel.test/concerts/1">Voir la fiche</a>
…
```

Vous retrouverez ces idées dans le cours Vue.js. Vue est une bibliothèque JavaScript qui construit l'interface d'une page dans le navigateur, et elle découpe elle aussi une page en composants. Un composant Vue déclare ses props avec `defineProps`, comme `@props` en Blade. Il reçoit son contenu à l'endroit marqué par la balise `<slot />`, comme `{{ $slot }}`. Et les deux-points devant un attribut transmettent là aussi la valeur d'une expression, écrite cette fois en JavaScript.

6. Slots nommés et attributs

Un slot nommé

Un **slot nommé** est un contenu supplémentaire que la vue transmet au composant dans une balise `<x-slot:…>`. Blade le place dans une variable du même nom : le contenu de `<x-slot:note>` devient la variable `$note` du composant. Ajoutez ces lignes à `concert-card.blade.php`, sous le paragraphe :

```xml
    @isset($note)
        <p class="note">{{ $note }}</p>
    @endisset
```

`@isset` n'affiche son bloc que si la variable existe. Dans la boucle de `concerts/index.blade.php`, donnez une remarque aux concerts de 90 minutes ou plus :

```XML
@foreach (concertsasconcerts as concertsasid ⇒ concert)<x-concert-card:concert="concert)
    <x-concert-card :concert="concert)<x-concert-card:concert="concert">
        @if ($concert['duration'] ⩾ 90)
            <x-slot:note>Entracte de 15 minutes</x-slot:note>
        @endif
    </x-concert-card>
    <a href="{{ route('concerts.show', $id) }}">Voir la fiche</a>
@endforeach
```

```
…
    <article class="concert-card">
    <h2>L&#039;Orchestre du Lac</h2>
    <p>Grande scène, 20 h</p>
            <p class="note">Entracte de 15 minutes</p>
    </article>
…
```

Pour les deux autres cartes, `$note` n'existe pas, et `@isset` ne produit rien.

Les attributs non déclarés

Blade place dans la variable `$attributes` tous les attributs de la balise que `@props` ne déclare pas. Sa méthode `merge()` les combine avec les attributs que le composant écrit lui-même. Pour l'attribut `class`, elle écrit les classes du composant, puis celles de la balise, à la suite. Remplacez le contenu de `alert.blade.php` :

```XML
@props(['type' ⇒ 'info'])

<div {{ attributes->merge([’class’⇒’alertalert-’.attributes→merge(['class' ⇒ 'alert
alert-' . attributes->merge([’class’⇒’alertalert-’.type]) }}>
    {{ $slot }}
</div>
```

Dans `festival/info.blade.php`, complétez l'appel de la seconde alerte :

```XML
<x-alert type="danger" class="top" id="sold-out">Le concert de 22 h est complet.</x-alert>
```

```
        <div class="alert alert-danger top" id="sold-out">
    Le concert de 22 h est complet.
</div>
```

La classe `top` s'ajoute à celles du composant, et `id` passe tel quel. L'attribut `type`, déclaré dans `@props`, n'apparaît pas dans le `<div>`.

Sans `@props`, Blade crée quand même une variable pour chaque attribut, mais le garde aussi dans `$attributes`. Supposez que le composant contienne `<div class="alert alert-{{ type }}" {{ attributes }}>`. Pour la seconde alerte, Blade recopie alors dans le `<div>` tous les attributs de la balise :

```
<div class="alert alert-danger" type="danger" class="top" id="sold-out">
```

Le `<div>` reçoit un attribut `type`, qui ne correspond à aucun attribut HTML, et deux attributs `class`, dont le navigateur ignore le second. Pour la première alerte, écrite sans `type`, la variable `$type` n'existe pas, et Laravel affiche l'erreur `Undefined variable $type`.

Déclarez avec `@props` chaque donnée de votre composant.

7. Le layout du site

Un **layout** est le template qui contient ce que les pages d'un site partagent : le `<head>`, l'en-tête, le menu et le pied de page. Ici, le layout est un composant. Chaque page l'utilise comme une balise qui entoure tout son contenu. Blade place ce contenu dans le `$slot` du layout, et le titre de la page dans une prop. Créez-le dans un sous-dossier `layouts` :

```bash
php artisan make:component layouts.app --view
```

```
   INFO  View [resources/views/components/layouts/app.blade.php] created successfully.
```

Remplacez son contenu par le HTML de la section 1. Écrivez `{{ $title }}` dans la balise `<title>`, et `{{ $slot }}` à la place du contenu de `<main>`. Écrivez aussi l'adresse de chaque lien du menu avec `route()`, comme au chapitre Routing :

```xml
@props(['title' => 'Festival'])

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>{{ title }} · Les Nuits du Parc</title>
</head>
<body>
    <header>
        <p>Les Nuits du Parc</p>
        <nav>
            <a href="{{ route('concerts.index') }}">Programme</a>
            <a href="{{ route('info') }}">Infos pratiques</a>
            <a href="{{ route('concerts.show', 3) }}">Concert de clôture</a>
        </nav>
    </header>
    <main>
        {{ slot }}
    </main>
    <footer>
        <p>Les Nuits du Parc, édition 2026</p>
    </footer>
</body>
</html>
```

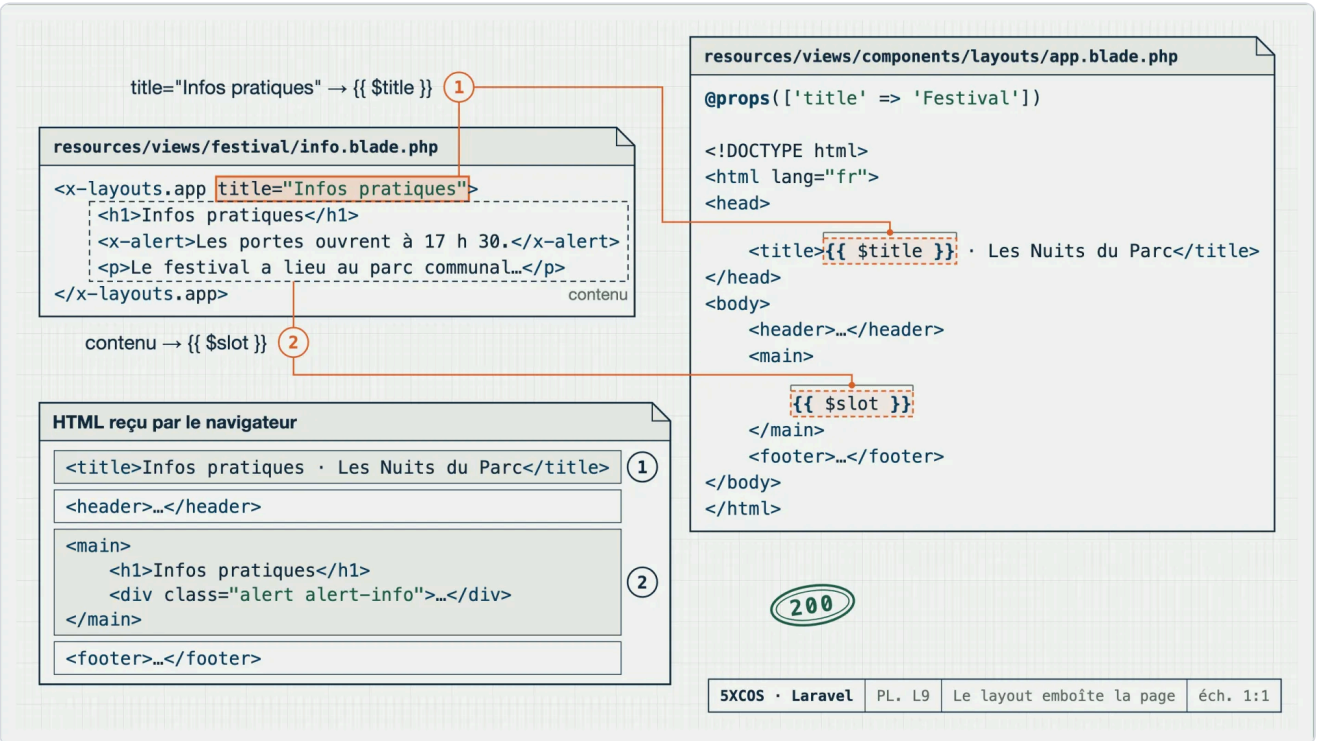
Un sous-dossier se note avec un point : pour afficher `components/layouts/app.blade.php`, vous écrivez `<x-layouts.app>`. Remplacez tout `festival/info.blade.php` par ceci :

```xml
<x-layouts.app title="Infos pratiques">
    <h1>Infos pratiques</h1>
    <x-alert>Les portes ouvrent à 17 h 30.</x-alert>
    <x-alert type="danger" class="top" id="sold-out">Le concert de 22 h est complet.</x-
alert>
    <p>Le festival a lieu au parc communal, du vendredi au dimanche.</p>
</x-layouts.app>
```

```bash
curl -s http://premier-test-laravel.test/infos
```

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Infos pratiques · Les Nuits du Parc</title>
</head>
<body>
    <header>
        <p>Les Nuits du Parc</p>
        <nav>
            <a href="http://premier-test-laravel.test/concerts">Programme</a>
            <a href="http://premier-test-laravel.test/infos">Infos pratiques</a>
            <a href="http://premier-test-laravel.test/concerts/3">Concert de clôture</a>
        </nav>
    </header>
    <main>
        <h1>Infos pratiques</h1>
    <div class="alert alert-info">
…
    <p>Le festival a lieu au parc communal, du vendredi au dimanche.</p>
    </main>
    <footer>
        <p>Les Nuits du Parc, édition 2026</p>
    </footer>
</body>
</html>
```

Blade a placé la valeur de l'attribut `title` à l'endroit de `{{ $title }}`, et tout ce qui se trouve entre `<x-layouts.app>` et `</x-layouts.app>` à l'endroit de `{{ $slot }}`. Le navigateur reçoit une page complète, et le fichier de la page ne contient que ce qui lui est propre.



Le titre dans un slot nommé

Entourez de la même façon le contenu de `concerts/index.blade.php` avec `<x-layouts.app title="Programme">`. Pour la fiche, le titre mélange un texte et une variable. Vous pourriez écrire l'attribut `:title="'Concert de ' . $concert['artist']"`, mais il mélange des guillemets simples, des guillemets doubles et l'opérateur `.` de PHP. Avec un slot nommé `title`, vous écrivez le titre comme dans le reste du template, avec `{{ }}`, et le résultat est le même. Dans `concerts/show.blade.php`, entourez le template existant de ces lignes :

```xml
<x-layouts.app>
    <x-slot:title>Concert de {{ $concert['artist'] }}</x-slot:title>

    {{-- le template de la section 3, inchangé --}}
</x-layouts.app>
```

```bash
curl -s http://premier-test-laravel.test/concerts/2 | grep "<title>"
```

```
    <title>Concert de L&#039;Orchestre du Lac · Les Nuits du Parc</title>
```

`$title` contient la valeur de l'attribut ou le contenu du slot nommé. Si la page ne donne ni l'un ni l'autre, Blade utilise la valeur par défaut, et le titre devient `Festival · Les Nuits du Parc`. Gardez l'attribut pour un titre simple.

Pour renommer le festival, vous modifiez maintenant trois lignes d'un seul fichier, `components/layouts/app.blade.php`. De même, pour ajouter une feuille de style à tout le site, il suffit d'écrire une balise `<link>` dans ce seul `<head>`.

8. L'ancienne façon, et pourquoi préférer les composants

Avant les composants, les layouts s'écrivaient avec quatre directives, que vous croiserez dans des projets existants et des tutoriels. Pour comparer, créez ces trois fichiers : ils écrivent la même page des infos pratiques de cette façon.

Le layout, `resources/views/layouts/base.blade.php` :

```xml
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>@yield('title', 'Festival') · Les Nuits du Parc</title>
</head>
<body>
    @include('partials.header')
    <main>
        @yield('content')
    </main>
    <footer>
        <p>Les Nuits du Parc, édition 2026</p>
    </footer>
</body>
</html>
```

Créez aussi l'en-tête, `resources/views/partials/header.blade.php`, et copiez-y le `<header>` du layout de la section 7, sans rien changer. Créez enfin la page, `resources/views/legacy/info.blade.php` :

```xml
@extends('layouts.base')

@section('title', 'Infos pratiques')

@section('content')
    <h1>Infos pratiques</h1>
    <x-alert>Les portes ouvrent à 17 h 30.</x-alert>
    <x-alert type="danger" class="top" id="sold-out">Le concert de 22 h est complet.</x-alert>
    <p>Le festival a lieu au parc communal, du vendredi au dimanche.</p>
@endsection
```

- **`@extends('layouts.base')`** indique quel layout la page utilise.
- **`@section('title', '…')`** et **`@section('content') … @endsection`** définissent deux blocs de contenu, appelés `title` et `content`.
- **`@yield('title', 'Festival')`** marque, dans le layout, l'endroit où Blade place le bloc `title` de la page, ou `Festival` si la page n'en définit pas.
- **`@include('partials.header')`** insère à cet endroit le contenu d'un autre template.

Ajoutez la route `Route::view('/ancien/infos', 'legacy.info');` à la fin de `routes/web.php`. Le HTML reçu est celui de `/infos`, balise pour balise. Seule l'indentation diffère.

```bash
curl -s http://premier-test-laravel.test/ancien/infos
```

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Infos pratiques · Les Nuits du Parc</title>
</head>
<body>
    <header>
        <p>Les Nuits du Parc</p>
…
    </header>
    <main>
            <h1>Infos pratiques</h1>
…
</html>
```

NOTE

Histoire de Laravel. Jusqu'à Laravel 6, les layouts s'écrivaient avec `@extends` et `@yield`. Laravel 7, sorti le 3 mars 2020, a introduit les composants appelés par une balise `<x-…>`. La documentation de Laravel 13 présente d'abord les layouts en composants, et décrit l'ancienne façon comme « the primary way of building applications prior to the introduction of components », c'est-à-dire la façon principale de construire une application avant l'arrivée des composants. Pour `@include`, elle précise qu'un composant peut insérer le même template, et qu'il permet en plus de lui transmettre des données et des attributs.

Ce que les composants apportent

`@extends` fonctionne toujours en Laravel 13. Deux différences font pourtant préférer les composants.

Avec les composants, une faute de nom provoque une erreur. Dans `legacy/info.blade.php`, écrivez `@section('contenu')` au lieu de `@section('content')` :

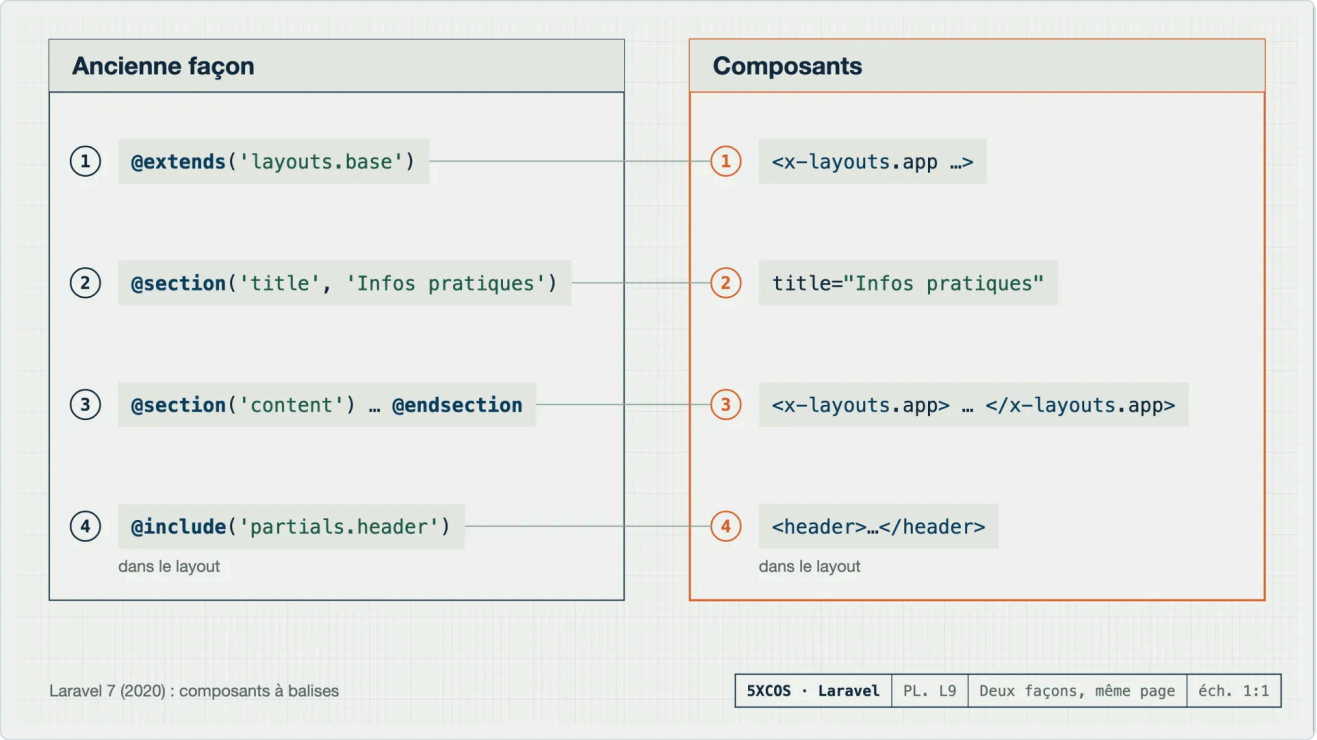
BASH

```
curl -s -i http://premier-test-laravel.test/ancien/infos
```

```
HTTP/1.1 200 OK
…
    <main>
            </main>
…
```

La page ne définit plus de bloc `content`. À l'endroit de `@yield('content')`, Blade n'insère donc rien, et Laravel renvoie la page avec le code 200, sans aucun message. Remettez `content`. Avec un composant, une faute dans le nom de la balise, comme `<x-layout.app>`, provoque au contraire l'erreur `Unable to locate a class or view for component [layout.app].`

Les données sont déclarées. En lisant `<x-layouts.app title="…">`, vous voyez quelles données la page transmet. En lisant `@props` dans le layout, vous voyez lesquelles il utilise. Avec `@yield`, il faut lire tout le layout pour trouver les noms des blocs.



Lisez `@extends` dans le code existant. Écrivez des composants dans le vôtre.

Blade et Vue.js côte à côte

Si vous suivez le cours Vue.js, ce tableau met les trois écritures côte à côte. Sinon, lisez seulement les trois premières colonnes.

Idée	Blade, ancienne façon	Blade, composants	Vue.js
Déclarer les données attendues	aucune déclaration	`@props(['concert'])`	`defineProps(['concert'])`
Passer une expression	le layout voit les variables de la page	`:concert="$concert"` (PHP)	`:concert="concert"` (JavaScript)
Zone de contenu principal	`@yield('content')`	`{{ $slot }}`	`<slot />`
Zone nommée	`@yield('title')` et `@section('title')`	`{{ $title }}` et `<x-slot:title>`	`<slot name="title" />` et `<template #title>`
Attributs non déclarés	aucun	`{{ $attributes }}`, à écrire soi-même	ajoutés automatiquement à la balise extérieure du composant

La comparaison a deux limites. Blade exécute un composant une seule fois, sur le serveur, et le navigateur reçoit du HTML qui ne change plus. Un composant Vue, lui, tourne dans le navigateur : il peut mettre la page à jour quand une donnée change, ou réagir à un clic avec `@click`. Ensuite, une prop Blade sans valeur par défaut que vous oubliez de transmettre arrête la page avec l'erreur `Undefined variable`. En Vue, la même prop vaut `undefined`, et la page s'affiche quand même.

9. Un formulaire et `@csrf`

Au chapitre Routing, une requête `POST` envoyée avec `curl` recevait l'erreur 419. Vous écrivez maintenant un vrai formulaire : un visiteur qui veut devenir bénévole envoie son prénom en `POST`. Une seule méthode suffit pour lui répondre, vous la placez donc dans un contrôleur à action unique, comme au chapitre Controllers :

```bash
php artisan make:controller VolunteerController --invokable
```

```
   INFO  Controller [app/Http/Controllers/VolunteerController.php] created successfully.
```

Remplacez le contenu du fichier. Au chapitre Controllers, `$request->query()` lisait un paramètre d'URL. La méthode `input()` de `Request` lit de la même façon un champ envoyé par un formulaire :

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class VolunteerController extends Controller
{
    public function __invoke(Request $request): string
    {
        $firstName = $request->input('first_name');

        return "Merci $firstName, votre candidature de bénévole est enregistrée.";
    }
}
```

Créez la vue avec `php artisan make:view volunteers.form` et écrivez le formulaire, pour l'instant sans `@csrf` :

```xml
<x-layouts.app title="Devenir bénévole">
    <h1>Devenir bénévole</h1>
    <form method="POST" action="{{ route('volunteers.store') }}">
        <label>Prénom <input type="text" name="first_name"></label>
        <button type="submit">Envoyer</button>
    </form>
</x-layouts.app>
```

Importez `VolunteerController` au-dessus de `WeatherController`, puis ajoutez les deux routes à la fin de `routes/web.php` :

```php
Route::view('/benevoles', 'volunteers.form')->name('volunteers.create');
Route::post('/benevoles', VolunteerController::class)->name('volunteers.store');
```

Ouvrez `http://premier-test-laravel.test/benevoles`, tapez `Lina` et cliquez sur « Envoyer ». Le navigateur affiche `Page Expired`, avec le code 419. C'est la protection contre la CSRF, vue au chapitre Routing : Laravel refuse une requête `POST` s'il ne peut pas vérifier qu'elle vient d'une page de votre site.

Au chapitre Routing, vous avez vu que Laravel regarde d'abord l'en-tête `Sec-Fetch-Site`, que les navigateurs n'envoient que sur une connexion HTTPS. Votre projet est servi en HTTP : Laravel cherche alors un jeton dans le formulaire, et il n'en trouve pas.

La directive `@csrf` ajoute ce jeton au formulaire. Écrivez-la juste sous la balise `<form>` :

```xml
    <form method="POST" action="{{ route('volunteers.store') }}">
        @csrf
        <label>Prénom <input type="text" name="first_name"></label>
```

```bash
curl -s http://premier-test-laravel.test/benevoles | grep _token
```

```
        <input type="hidden" name="_token" value="6HjTJozqH7XAcXVJCTyVptQjZ9lRXU2dZV2gkYri"
autocomplete="off">        <label>Prénom <input type="text" name="first_name"></label>
```

Chez vous, ce champ caché contient une autre suite de 40 caractères. Laravel garde une copie de ce jeton dans la **session** du visiteur, un espace de stockage que le serveur réserve à chaque visiteur. Quand le formulaire est envoyé, Laravel compare le jeton reçu avec celui de la session.

Pour savoir quelle session appartient à quel navigateur, Laravel envoie un **cookie**, une petite valeur que le navigateur conserve et renvoie ensuite à chaque requête vers votre site. Dans votre projet, ce cookie s'appelle `premier-test-session`, d'après le nom de l'application écrit dans `.env`.

Rechargez le formulaire, tapez de nouveau `Lina` et envoyez-le. Cette fois, Laravel accepte la requête, et le navigateur affiche la phrase renvoyée par le contrôleur :

```
Merci Lina, votre candidature de bénévole est enregistrée.
```

Une page piégée sur un autre site ne peut pas lire ce jeton. Le formulaire qu'elle envoie arrive sans lui, et Laravel le refuse avec le code 419. **Écrivez `@csrf` dans chaque formulaire `POST`.** Pour l'instant, le contrôleur ne vérifie pas le contenu des champs.

Ajoutez enfin le lien dans le menu, une ligne dans `components/layouts/app.blade.php`, et toutes les pages qui utilisent ce layout l'affichent :

```xml
<a href="{{ route('volunteers.create') }}">Devenir bénévole</a>
```

À retenir

- Un contrôleur passe les données à la vue avec un tableau, et `@if`, `@foreach`, `$loop` et `@forelse` les affichent sans balise `<?php` dans la vue.
- `{{ }}` échappe le texte avec la fonction `e()`, alors que `{!! !!}` l'affiche tel quel et ne sert qu'au HTML que vous avez écrit vous-même.
- Un composant anonyme est un fichier de `resources/views/components/` qui déclare ses props avec `@props` et reçoit son contenu dans `$slot`.
- Un attribut précédé de deux-points transmet une expression PHP, et un attribut sans deux-points transmet un texte fixe.
- Le layout d'un site est un composant, `<x-layouts.app>`, qui remplace `@extends`, `@section`, `@yield` et `@include` dans un projet neuf.

Exercices

Comptez 115 minutes pour les exercices, en séance ou à la maison.

Faites les exercices 1 à 8 dans `premier-test-laravel`, pour Octet, une entreprise d'informatique inventée. Ajoutez chaque route à la fin de `routes/web.php` et chaque import en haut du fichier. Supprimez la citation de chaque fichier créé par `make:view` ou `make:component`.

1. Une page d'accueil (10 minutes)

Créez la vue `home` avec `make:view`, avec un titre « Bienvenue chez Octet » et un paragraphe. Affichez-la à l'adresse `/accueil`, route nommée `home`, sans contrôleur. Inspirez-vous de la section 1.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/views

2. L'entreprise (10 minutes)

Créez `CompanyController`, dont la méthode `index()` passe ce tableau à la vue `company`, sous le nom `info`. La vue affiche une ligne par information, la clé puis la valeur. Reliez-la à l'adresse `GET /entreprise`, avec une route nommée `company`. Reprenez le `id =>concert` de la section 3.

```php
$info = [
    'Nom' => 'Octet',
    'Année de création' => 2012,
    'Mission' => 'Réparer plutôt que remplacer',
];
```

```
<h1>L'entreprise</h1>
<ul>
            <li>Nom : Octet</li>
            <li>Année de création : 2012</li>
            <li>Mission : Réparer plutôt que remplacer</li>
    </ul>
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/views

3. Les membres du club (15 minutes)

Créez `MemberController` avec cette propriété. Sa méthode `index()`, sur `GET /membres` nommée `members.index`, passe les membres à la vue `members`. Chaque ligne affiche le numéro donné par `$loop`, le prénom, puis l'âge si le membre a 18 ans ou plus, et sinon le mot « mineur ». Une méthode `empty()`, sur `GET /membres/vide`, passe un tableau vide à la même vue, qui affiche « Aucun membre ». Inspirez-vous de la section 3.

```php
private array $members = [
    ['first_name' => 'Alice', 'age' => 29],
    ['first_name' => 'Bob', 'age' => 34],
    ['first_name' => 'Chloé', 'age' => 17],
    ['first_name' => 'Damien', 'age' => 42],
];
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/views

4. Des cartes de services (10 minutes)

Créez le composant anonyme `card`, avec une prop `title` affichée dans un `<h2>` et le slot dans un `<p>`, le tout dans un `<div class="card">`. Affichez trois cartes, Design, Développement et Marketing, dans la vue `services`, reliée à `/services` par `Route::view`. La carte Développement reçoit en plus la classe

`featured`. Inspirez-vous des sections 5 et 6.

```
<div class="card featured">
    <h2>Développement</h2>
    <p>Un site Laravel sur mesure.</p>
</div>
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/views

5. Les produits et leur fiche (20 minutes)

Créez `ProductController` avec cette propriété, et deux routes : `GET /produits`, nommée `products.index`, et `GET /produits/{id}`, nommée `products.show`. `index()` passe les produits à la vue `products.index`, qui affiche pour chacun un lien vers sa fiche écrit avec `route()`, son prix, et « Promo ! » s'il est en promotion. `show()` affiche la vue `products.show` d'un produit, et renvoie une page 404 si le numéro n'existe pas. Inspirez-vous de `ConcertController`, à la section 3.

PHP

```php
private array $products = [
    1 => ['name' => 'PC portable', 'price' => 1200, 'on_sale' => true],
    2 => ['name' => 'Souris sans fil', 'price' => 80, 'on_sale' => false],
    3 => ['name' => 'Clavier mécanique', 'price' => 150, 'on_sale' => true],
    4 => ['name' => 'Casque audio', 'price' => 100, 'on_sale' => false],
];
```

BASH

```
curl -s http://premier-test-laravel.test/produits/3
curl -s -i http://premier-test-laravel.test/produits/9
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/views

6. Un composant produit (10 minutes)

Créez le composant anonyme `product`, qui reçoit un produit entier dans une prop `product` et affiche son nom, son prix et « Promo ! » s'il y a lieu. Utilisez-le dans la boucle de `products/index.blade.php`, en gardant le lien. Inspirez-vous de la carte du concert, à la fin de la section 5.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/views

7. Le layout d'Octet (15 minutes)

Créez `components/layouts/company.blade.php`, avec une prop `title` qui vaut `Octet` par défaut et qui forme seule le contenu de `<title>`, un menu de trois liens écrits avec `route()` vers l'accueil, l'entreprise et les produits, `{{ $slot }}` dans `<main>` et un pied de page. Appliquez-le aux pages des exercices 1, 2, 3 et 5, chacune avec son titre. Sur la fiche d'un produit, le titre est le nom du produit. Inspirez-vous de la section 7.

```bash
curl -s http://premier-test-laravel.test/produits/3 | grep "<title>"
```

```
    <title>Clavier mécanique</title>
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/views

8. Une newsletter (15 minutes)

Créez une page `/newsletter`, affichée par `Route::view` et nommée `newsletter.create`, avec un formulaire qui envoie un champ `email` en `POST` vers la route `newsletter.store`. Celle-ci appelle un contrôleur à action unique `NewsletterController`, qui renvoie « Inscription à la newsletter enregistrée pour » suivi de l'adresse. Envoyez le formulaire dans le navigateur sans `@csrf`, puis avec, et notez le code obtenu. Inspirez-vous de la section 9. Avec `@csrf`, la réponse est :

```
Inscription à la newsletter enregistrée pour lina@example.be.
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/views

9. Lire et diagnostiquer (sur papier, 10 minutes)

La vue reçoit `$text = '<b>Promo</b>'`, `$name = 'Design'` et `$empty = []`, et `card` est le composant de l'exercice 4. Écrivez le HTML produit par chaque extrait, le `<h2>` seulement pour 3 et 4.

```xml
{{-- 1 --}} <p>{{ text }}</p>
{{-- 2 --}} <p>{!! text !!}</p>
{{-- 3 --}} <x-card title="name">Texte</x-card>--4--<x-card:title="name">Texte</x-card>
{{-- 4 --}} <x-card :title="name">Texte</x-card>--4--<x-card:title="name">Texte</x-card>
{{-- 5 --}} {{-- Réduction --}} <!-- Promo --> <p>Prix</p>
{{-- 6 --}} @forelse (emptyasempty as emptyasx) <p>{{ $x }}</p> @empty <p>Rien à
afficher</p> @endforelse
```

Pour chaque message, écrivez la cause et le fichier à corriger.

```
View [products.list] not found.
Undefined variable $product (View: …/resources/views/products/show.blade.php)
Unable to locate a class or view for component [products].
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/views

10. Six cartes

À quoi sert un composant Blade ?

À écrire une seule fois un morceau de HTML utilisé sur plusieurs pages, comme une alerte ou un layout.

Que change le deux-points dans `:concert="$concert"` ?

L'attribut transmet la valeur de l'expression PHP, et pas le texte `$concert`.

Que reçoit `{{ $slot }}` ?

Ce qui est écrit entre les balises du composant, comme `Complet` dans `<x-alert>Complet</x-alert>`.

Pourquoi écrire `@props` ?

Pour déclarer les données du composant, leur donner une valeur par défaut et les retirer de `$attributes`.

`@extends` ou `<x-layouts.app>` ?

Lisez `@extends` dans le code existant, écrivez `<x-layouts.app>` dans un projet neuf.

Quand utiliser `{!! !!}` ?

Seulement pour du HTML que vous avez écrit vous-même, jamais pour une saisie de visiteur.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/views

3.10 Modèles : une table

Bloc Laravel · séance 4 · ~50 min

Slides du chapitre (PDF) PDF

Au chapitre Views, les trois concerts des Nuits du Parc sont écrits dans le tableau `$concerts` de `ConcertController`. Vous les rangez maintenant dans une table `concerts` de la base SQLite, lue par une classe `Concert`. À la fin, les pages `/concerts` et `/concerts/{id}` gardent leur mise en page, sans la liste des invités, mais leurs données viennent de la base.

À la fin de ce chapitre, vous serez capables de :

1. **écrire** une migration qui crée une table, puis l'appliquer et l'annuler avec Artisan,
2. **créer** un modèle et déclarer avec `#[Fillable]` les colonnes que `create()` peut remplir,
3. **remplir** la base avec un seeder, et la remettre à neuf avec `migrate:fresh --seed`,
4. **interroger** une table avec `all`, `find`, `findOrFail`, `where`, `orderBy`, `first` et `count`, et **inspecter** le résultat avec `dump()` et `dd()`,
5. **afficher** une page à partir de la base, puis **modifier** et **supprimer** une ligne.

1. Le tableau qui ne tient plus

Voici le début de la propriété écrite au chapitre Views :

```php
private array $concerts = [
    1 => [
        'artist' => 'Duo Mistral',
        'stage' => 'Chapiteau',
        'time' => '18 h',
        …
    ],
    …
];
```

Chaque concert occupe huit lignes de la classe. Avec trente concerts, la propriété compterait environ deux cent quarante lignes avant la première méthode. Une autre page ne peut pas lire cette propriété privée. Pour annoncer le prochain concert sur l'accueil, il faudrait donc recopier le tableau dans un second contrôleur.

Le code qui sent mauvais

En binôme, cinq minutes. L'organisatrice du festival ne programme pas en PHP. Elle doit avancer le concert de Duo Mistral à 17 h. Qui peut faire cette modification, dans quel fichier, et que devient le site si la personne qui la fait efface une virgule par erreur ?

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/modeles-une-table

2. La migration

Au chapitre MVC, une migration a créé la table `posts` des articles. Créez maintenant celle des concerts, sans modèle. Artisan reconnaît la forme `create_…_table` de l'argument et prépare la création de la table `concerts`.

```bash
php artisan make:migration create_concerts_table
```

```
   INFO  Migration [database/migrations/2026_09_28_213249_create_concerts_table.php] created
successfully.
```

Le nom du fichier commence par la date et l'heure de la commande, différentes chez vous. Sous Windows, Artisan affiche le chemin complet du fichier. Laravel exécute les migrations dans l'ordre de leurs noms. Voici le fichier créé :

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('concerts', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('concerts');
    }
};
```

`return new class extends Migration` crée l'objet d'une classe anonyme, lue au chapitre 11 de la POO. Vous connaissez le reste depuis le chapitre MVC : `up()` crée la table, `down()` la supprime, `id()` ajoute la clé primaire et `timestamps()` les colonnes `created_at` et `updated_at`. Dans `up()`, la fonction reçoit `$table`, un objet `Blueprint`, et chaque appel sur `$table` ajoute une colonne.

Ajoutez entre `id()` et `timestamps()` quatre des cinq colonnes d'un concert, toutes sauf la description :

```php
Schema::create('concerts', function (Blueprint $table) {
    $table->id();
    $table->string('artist');
    $table->string('stage');
    $table->dateTime('starts_at');
    $table->unsignedSmallInteger('duration');
    $table->timestamps();
});
```

La méthode donne le type de la colonne, et son argument le nom de la colonne. `starts_at` contiendra la date et l'heure sous la forme `2026-10-16 18:00:00`, que la base sait trier, et non plus le texte « 18 h ». `duration` compte des minutes. Voici les types utiles dans ce cours :

Méthode	La colonne contient	Exemple
`string('artist')`	un texte court, 255 caractères au plus	un nom
`text('description')`	un texte long	une description
`unsignedSmallInteger('duration')`	un entier de 0 à 65 535	des minutes
`date('released_on')`	une date sans heure	une sortie de film
`dateTime('starts_at')`	une date et une heure	un début de concert
`decimal('price', 5, 2)`	un nombre exact, 5 chiffres dont 2 décimales	un prix

Les limites du tableau sont celles de MySQL, qui refuse une valeur trop longue ou hors plage. PostgreSQL fait respecter la longueur des textes et les décimales. SQLite, la base de votre projet, n'impose ni longueur ni plage : `artist` accepterait un nom de 300 caractères, et `duration` la valeur 70 000.

Une colonne déclarée ainsi refuse une case vide, `null` en PHP. Pour l'accepter, ajoutez `->nullable()` après le type : `$table->text('synopsis')->nullable()`.

Appliquer la migration

`migrate` exécute la méthode `up()` de chaque migration pas encore appliquée :

```bash
php artisan migrate
```

```
   INFO  Running migrations.

  2026_09_28_213249_create_concerts_table ....................... 7.72ms DONE
```

`migrate:status` affiche l'état de chaque migration :

```bash
php artisan migrate:status
```

```
  Migration name ..................................................... Batch / Status
  0001_01_01_000000_create_users_table ............................... [1] Ran
  0001_01_01_000001_create_cache_table ............................... [1] Ran
  0001_01_01_000002_create_jobs_table ................................ [1] Ran
  2026_09_28_211118_create_posts_table ............................... [2] Ran
  2026_09_28_213249_create_concerts_table ............................ [3] Ran
```

Laravel inscrit chaque migration appliquée dans la table `migrations` de la base. Avant d'exécuter une migration, `migrate` cherche son nom dans cette table. Le numéro entre crochets est le **lot** : les migrations lancées par la même commande forment un lot. Relancez la commande :

```bash
php artisan migrate
```

```
   INFO  Nothing to migrate.
```

`db:table` décrit une table :

```bash
php artisan db:table concerts
```

```
  main.concerts ..............................................................
  Columns ..................................................................... 7

  Column ...................................................................... Type
  id integer, autoincrement ..................................... integer
  artist varchar ................................................ varchar
  stage varchar ................................................. varchar
  starts_at datetime ........................................... datetime
  duration integer .............................................. integer
  created_at datetime, nullable ................................. datetime
  updated_at datetime, nullable ................................. datetime

  Index ......................................................................
  primary id ................................................... primary
```

SQLite nomme `varchar` un texte court, sans longueur, et range `unsignedSmallInteger` parmi les `integer`. Dans VS Code, l'extension SQLite Viewer du chapitre Environnement de développement ouvre `database/database.sqlite` et montre la même table, encore vide.

Corriger une migration en cours de travail

Il manque la description. Or `migrate` ne rejoue pas une migration inscrite dans `migrations`. Annulez d'abord le dernier lot :

```bash
php artisan migrate:rollback
```

```
   INFO  Rolling back migrations.

  2026_09_28_213249_create_concerts_table ....................... 10.37ms DONE
```

Laravel a exécuté la méthode `down()` de la migration du lot 3, qui a supprimé la table `concerts`, puis il a effacé cette migration de la table `migrations`. Ajoutez la colonne sous `duration` :

```php
table->unsignedSmallInteger('duration');table→unsignedSmallInteger('duration');
table->unsignedSmallInteger('duration');table→text('description');
```

Relancez la migration, puis décrivez de nouveau la table (sortie raccourcie) :

```bash
php artisan migrate
php artisan db:table concerts
```

```
   INFO  Running migrations.

  2026_09_28_213249_create_concerts_table ....................... 7.43ms DONE

  main.concerts ..............................................................
  Columns ................................................................ 8
…
  duration integer ................................................ integer
  description text ................................................... text
  created_at datetime, nullable ................................... datetime
…
```

Chaque ligne de la méthode up() devient une colonne de la table concerts

Ce retour en arrière ne vaut que sur votre machine. Chez un collègue ou sur un serveur, la migration déjà appliquée reste inscrite dans `migrations`, et `migrate` ne la rejoue pas. **Une migration appliquée par d'autres ne se modifie plus : écrivez une nouvelle migration qui change la table.**

3. Le modèle

Un modèle est une classe liée à une table : la classe interroge la table, et chaque objet de la classe représente une ligne. Créez celui des concerts :

```bash
php artisan make:model Concert
```

```
   INFO  Model [app/Models/Concert.php] created successfully.
```

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Concert extends Model
{
    //
}
```

Comme `Post` au chapitre MVC, la classe est vide. Par héritage, elle reçoit de `Model` les méthodes qui lisent et écrivent dans la base.

Vous n'avez écrit nulle part le nom de la table. Eloquent, la partie de Laravel qui relie les classes aux tables, le déduit du nom de la classe, mis au pluriel anglais et en minuscules, avec un `_` entre les mots :

Classe	Table
`Concert`	`concerts`
`ParkingRate`	`parking_rates`
`Person`	`people`
`Cheval`	`chevals`

Avec la classe `Cheval`, Eloquent interroge la table `chevals`. Une migration qui crée la table `chevaux` laisse donc le modèle chercher une table qui n'existe pas. C'est pourquoi tout le code du cours porte des noms anglais.

Un premier essai dans une route

Pour essayer le modèle sans écrire de page, vous utilisez une **route bac à sable**, une route réservée à vos essais. Dans `routes/web.php`, importez le modèle au-dessus de `use Illuminate\Support\Facades\Route;` :

```php
use App\Models\Concert;
```

Ajoutez cette route à la fin du fichier. La méthode `create()` reçoit un tableau dont chaque clé porte le nom d'une colonne. Elle crée un objet `Concert` avec ces valeurs et l'enregistre dans la table.

```php
Route::get('/bac-a-sable', function () {
    Concert::create([
        'artist' => 'Duo Mistral',
        'stage' => 'Chapiteau',
        'starts_at' => '2026-10-16 18:00:00',
        'duration' => 45,
        'description' => 'Guitare et violon, <strong>entrée libre</strong>.',
    ]);

    return 'Concert enregistré';
});
```

Ouvrez `http://premier-test-laravel.test/bac-a-sable` : Laravel répond par une erreur 500.

```
Illuminate\Database\Eloquent\MassAssignmentException
Add [artist] to fillable property to allow mass assignment on [App\Models\Concert].
```

Remplir plusieurs colonnes d'un coup avec un tableau s'appelle une **affectation de masse**. Laravel la refuse tant que le modèle ne donne pas la liste des colonnes qu'un tableau peut remplir. Le message cite `artist`, la première clé du tableau.

Écrivez la liste au-dessus de la classe `Concert`, avec l'attribut `#[Fillable]` lu dans `User.php` au chapitre Structure :

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Attributes\Fillable;
use Illuminate\Database\Eloquent\Model;

#[Fillable(['artist', 'stage', 'starts_at', 'duration', 'description'])]
class Concert extends Model
{
    //
}
```

Rechargez la page, ou demandez-la avec `curl` :

```bash
curl -s http://premier-test-laravel.test/bac-a-sable
```

```
Concert enregistré
```

Avec la liste, `create()` ignore les clés absentes. Sans cette protection, un contrôleur d'inscription qui passerait à `create()` tout un formulaire laisserait un visiteur ajouter un champ `is_admin` et se rendre administrateur. Au chapitre MVC, `forceCreate()` ne consultait pas la liste.

> **ATTENTION**
>
> Une colonne oubliée dans la liste est ignorée sans message. Sans `'description'` dans `#[Fillable]`, la base refuse la ligne incomplète : `NOT NULL constraint failed: concerts.description`. Devant ce message, relisez d'abord la liste.

Chaque visite ajoute un concert : rechargez la page, et la table contient deux lignes « Duo Mistral ».

4. Le seeder

Un seeder, vu au chapitre Structure, est une classe qui ajoute des données de départ dans la base.

```bash
php artisan make:seeder ConcertSeeder
```

```
   INFO  Seeder [database/seeders/ConcertSeeder.php] created successfully.
```

Remplissez sa méthode `run()`, vide, avec les trois concerts du chapitre Views, le vendredi 16 octobre 2026, sans les invités :

```php
<?php

namespace Database\Seeders;

use App\Models\Concert;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class ConcertSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        Concert::create([
            'artist' => 'Duo Mistral',
            'stage' => 'Chapiteau',
            'starts_at' => '2026-10-16 18:00:00',
            'duration' => 45,
            'description' => 'Guitare et violon, <strong>entrée libre</strong>.',
        ]);

        Concert::create([
            'artist' => "L'Orchestre du Lac",
            'stage' => 'Grande scène',
            'starts_at' => '2026-10-16 20:00:00',
            'duration' => 90,
            'description' => 'Un orchestre de trente musiciens, <strong>gratuit</strong>
pour les moins de 12 ans.',
        ]);

        Concert::create([
            'artist' => 'Orage Mécanique',
            'stage' => 'Grande scène',
            'starts_at' => '2026-10-16 22:00:00',
            'duration' => 75,
            'description' => 'Rock et percussions pour la <strong>clôture</strong>.',
        ]);
    }
}
```

La commande `db:seed` exécute `DatabaseSeeder`, créé avec le projet. Sa méthode `run()` crée déjà un utilisateur de test, `test@example.com`. Ajoutez à la fin l'appel de votre seeder :

```php
public function run(): void
{
    // User::factory(10)->create();

    User::factory()->create([
        'name' => 'Test User',
        'email' => 'test@example.com',
    ]);

    $this->call(ConcertSeeder::class);
}
```

```bash
php artisan db:seed
```

```
   INFO  Seeding database.

  Database\Seeders\ConcertSeeder ..................................... RUNNING
  Database\Seeders\ConcertSeeder ..................................... 12 ms DONE
```

La table `concerts` contient cinq lignes : les deux « Duo Mistral » de la route bac à sable, puis les trois du seeder. Relancez la commande :

```bash
php artisan db:seed
```

```
   INFO  Seeding database.


   Illuminate\Database\UniqueConstraintViolationException

  SQLSTATE[23000]: Integrity constraint violation: 19 UNIQUE constraint failed: users.email
(Connection: sqlite, …
```

`DatabaseSeeder` a voulu créer une seconde fois l'utilisateur `test@example.com`, et la table `users` interdit deux lignes avec la même adresse. Un seeder ajoute des lignes à celles qui existent. Pour tout reprendre à zéro, lancez `migrate:fresh` avec l'option `--seed` (sortie raccourcie) :

```bash
php artisan migrate:fresh --seed
```

```
  Dropping all tables ..................................... 4.77ms DONE

   INFO  Preparing database.
…
   INFO  Running migrations.

  0001_01_01_000000_create_users_table ..................... 23.02ms DONE
…
  2026_09_28_213249_create_concerts_table ..................... 3.76ms DONE


   INFO  Seeding database.

  Database\Seeders\ConcertSeeder ................................... RUNNING
  Database\Seeders\ConcertSeeder ............................... 12 ms DONE
```

La commande supprime toutes les tables, sans passer par les méthodes `down()`. Elle applique ensuite toutes les migrations, puis lance `DatabaseSeeder`. La base contient un utilisateur et les trois concerts, numérotés 1, 2 et 3.

> **ATTENTION**
>
> `migrate:fresh` a aussi vidé la table `posts` : les trois articles du chapitre MVC ne figuraient dans aucun seeder, et `/posts` affiche maintenant une liste vide. Ne lancez jamais cette commande sur une base dont vous gardez les données, comme celle d'un site en ligne.

5. Lire la base sans page

Deux fonctions montrent ce que renvoie une requête. **`dump()`** affiche dans la page le contenu d'une variable, puis le code continue. **`dd()`** affiche le contenu, puis arrête le programme : Laravel n'envoie aucune vue.

`Concert::all()` demande à la base toutes les lignes de la table `concerts`. Remplacez le contenu de la route bac à sable :

PHP

```php
Route::get('/bac-a-sable', function () {
    dd(Concert::all());
});
```

Ouvrez `/bac-a-sable`. La page affiche un bloc sombre :

```
Illuminate\Database\Eloquent\Collection {#1522 ▼ // routes/web.php:182
  #items: array:3 [▶]
  #escapeWhenCastingToString: false
}
```

La première ligne donne la classe du résultat, une collection comme au chapitre MVC, puis le fichier et la ligne du `dd()`. Le numéro après `#` peut différer chez vous. Avec `▶`, dépliez `#items`, puis le premier élément, puis `#attributes` (sortie raccourcie) :

```
  #items: array:3 [▼
    0 ⇒ App\Models\Concert {#1534 ▼
      #connection: "sqlite"
      #table: "concerts"
…
      #attributes: array:8 [▼
        "id" ⇒ 1
        "artist" ⇒ "Duo Mistral"
        "stage" ⇒ "Chapiteau"
        "starts_at" ⇒ "2026-10-16 18:00:00"
        "duration" ⇒ 45
        "description" ⇒ "Guitare et violon, <strong>entrée libre</strong>."
        "created_at" ⇒ "2026-09-28 21:35:32"
        "updated_at" ⇒ "2026-09-28 21:35:32"
      ]
…
```

Chaque élément est un objet `Concert`, qui représente une ligne de la table `concerts`. `#attributes` contient une valeur par colonne. Eloquent a rempli `created_at` et `updated_at` lui-même, pendant `create()`. Vous lisez ces valeurs avec une flèche, comme `$post->title` au chapitre MVC.

Un objet, `null` ou un nombre

Plusieurs `dump()` peuvent précéder le `dd()` final :

```php
Route::get('/bac-a-sable', function () {
    dump(Concert::count());
    dump(Concert::find(2)->artist);
    dd(Concert::find(99));
});
```

```
3 // routes/web.php:182

"L'Orchestre du Lac" // routes/web.php:183

null // routes/web.php:184
```

`find(2)` cherche la ligne dont l'`id` vaut 2 et renvoie un seul objet `Concert`, pas une collection. `find(99)` ne trouve aucune ligne et renvoie `null`.

Dans une vue, `$concert->artist` sur `null` arrêterait la page avec une erreur 500. **`findOrFail()`** cherche de la même façon, mais lance une exception quand la ligne n'existe pas, et Laravel la transforme en réponse 404. Remplacez le contenu de la route par `dd(Concert::findOrFail(99));` :

```bash
curl -s -i http://premier-test-laravel.test/bac-a-sable
```

```
HTTP/1.1 404 Not Found
…
```

Le `dd()` n'affiche rien : l'exception a arrêté le traitement avant lui.

Filtrer et trier

`where()` ajoute une condition à la requête, et `orderBy()` un tri, sans encore l'envoyer à la base. `get()`, `first()` ou `count()` l'envoient. Une collection se parcourt avec `foreach`, comme un tableau :

```php
Route::get('/bac-a-sable', function () {
    foreach (Concert::where('stage', 'Grande scène')->get() as $concert) {
        dump($concert->artist);
    }

    dump(Concert::orderBy('starts_at', 'desc')->first()->artist);
    dd(Concert::where('stage', 'Grande scène')->orderBy('starts_at')->toRawSql());
});
```

```
"L'Orchestre du Lac" // routes/web.php:183

"Orage Mécanique" // routes/web.php:183

"Orage Mécanique" // routes/web.php:186

"select * from "concerts" where "stage" = 'Grande scène' order by "starts_at" asc" //
routes/web.php:187
```

`orderBy('starts_at', 'desc')` trie de l'heure la plus tardive à la plus tôt, et `first()` prend donc le dernier concert de la soirée. Sans second argument, le tri est croissant, `asc`. `toRawSql()`, vu au chapitre MVC, montre le SQL de la requête.

Vous écrivez	Vous recevez
`Concert::all()`, `…->get()`	une collection d'objets `Concert`, peut-être vide
`Concert::find(2)`, `…->first()`	un objet `Concert`, ou `null`
`Concert::findOrFail(2)`	un objet `Concert`, ou une réponse 404
`Concert::count()`, `…->count()`	un entier

Ce que renvoient get(), find(), first() et count() sur la table des trois concerts

Écrivez `findOrFail()` dans un contrôleur quand l'`id` vient de l'adresse, `find()` quand `null` vous convient, et `where()` pour chercher selon une autre colonne.

6. La page lit la base

Remplacez tout le contenu de `ConcertController.php`. La propriété `$concerts` disparaît : `index()` lit les concerts triés par heure, et `show()` lit celui dont l'`id` est dans l'adresse.

```php
<?php

namespace App\Http\Controllers;

use App\Models\Concert;
use Illuminate\View\View;

class ConcertController extends Controller
{
    public function index(): View
    {
        concerts=Concert::orderBy(’startsat’)->get();returnview(’concerts.index’,
[’concerts’⇒concerts = Concert::orderBy('starts_at')→get();

        return view('concerts.index', ['concerts' ⇒ concerts=Concert::orderBy(’startsat’)
->get();returnview(’concerts.index’,[’concerts’⇒concerts]);
    }

    public function show(int $id): View
    {
        $concert = Concert::findOrFail($id);

        return view('concerts.show', ['concert' ⇒ $concert]);
    }
}
```

Au chapitre Views, `show()` testait la clé avec `isset()` et appelait `abort(404)`. `findOrFail()` remplace ce test. Sans toucher aux vues, demandez le programme :

```bash
curl -s http://premier-test-laravel.test/concerts
```

```
…
            <article class="concert-card">
    <h2>Duo Mistral</h2>
    <p>Chapiteau, </p>
    </article>
        <a href="http://premier-test-laravel.test/concerts/0">Voir la fiche</a>
…
```

Les noms s'affichent encore, car un modèle accepte aussi l'écriture `$concert['artist']`. L'heure reste vide, car la table n'a pas de colonne `time`. Le lien vise `/concerts/0`, car `$id` reçoit maintenant la position du concert dans la collection, qui commence à 0. Enfin, la fiche `/concerts/2` répond 500, car `$concert['guests']` vaut `null` :

```
foreach() argument must be of type array|object, null given (View:
…/resources/views/concerts/show.blade.php)
```

Les vues lisent un objet

Un concert est maintenant un objet, comme `$post` au chapitre MVC. Dans les trois vues, `$concert['artist']` devient `$concert->artist`, et de même pour les autres clés. Remplacez le contenu de `components/concert-card.blade.php` :

```xml
@props(['concert'])

<article class="concert-card">
    <h2>{{ concert->artist }}</h2>
    <p>{{ concert->stage }}, {{ concert->starts_at->format('G \h') }}</p>
    @isset(note)
        <p class="note">{{ $note }}</p>
    @endisset
</article>
```

Dans `concerts/index.blade.php`, la boucle ne lit plus de clé, et le lien utilise la colonne `id` :

```xml
<x-layouts.app title="Programme">
    <h1>Programme</h1>
    @foreach (concertsasconcerts as concertsasconcert)
        <x-concert-card :concert="concert">@if(concert">
            @if (concert">@if(concert->duration ⩾ 90)
                <x-slot:note>Entracte de 15 minutes</x-slot:note>
            @endif
        </x-concert-card>
        <a href="{{ route('concerts.show', $concert->id) }}">Voir la fiche</a>
    @endforeach
</x-layouts.app>
```

Dans `concerts/show.blade.php`, faites les mêmes remplacements, et supprimez le titre « Invités » avec sa liste :

```xml
<x-layouts.app>
    <x-slot:title>Concert de {{ concert->artist }}</x-slot:title>

    {{-- Ce commentaire Blade n'arrive pas dans la page --}}
    <!-- Ce commentaire HTML arrive dans la page -->
    <h1>{{ concert->artist }}</h1>
    <p>{{ concert->stage }}, {{ concert->starts_at->format('G \h') }}</p>

    @if (concert->duration >= 90)
        <p>Long concert : {{ $concert->duration }} minutes, avec entracte.</p>
    @elseif (concert->duration >= 60)
        <p>Concert de {{ concert->duration }} minutes, sans entracte.</p>
    @else
        <p>Concert court : {{ concert->duration }} minutes.</p>
    @endif

    <p>{{ concert->description }}</p>
    <p>{!! concert->description !!}</p>
</x-layouts.app>
```

Rechargez le programme :

```
Call to a member function format() on string (View: …/resources/views/components/concert-card.blade.php) (View: …/resources/views/components/concert-card.blade.php)
```

Eloquent donne à chaque colonne la valeur que la base lui envoie. `starts_at` arrive donc sous forme de texte, `"2026-10-16 18:00:00"`, et une chaîne n'a pas de méthode `format()`. Une **conversion**, *cast* en anglais, transforme la valeur d'une colonne à chaque lecture. Déclarez-la dans la méthode `casts()` de `Concert` :

```php
#[Fillable(['artist', 'stage', 'starts_at', 'duration', 'description'])]
class Concert extends Model
{
    protected function casts(): array
    {
        return [
            'starts_at' => 'datetime',
        ];
    }
}
```

Avec la conversion `datetime`, `$concert->starts_at` renvoie un objet Carbon, la bibliothèque de dates installée au chapitre Framework et librairie.

La méthode `format()` de Carbon écrit la date selon des lettres. `G` donne l'heure sans zéro devant, et `\h` écrit la lettre h telle quelle. `format('G \h')` renvoie donc `18 h`, comme l'ancienne clé `time`.

```bash
curl -s http://premier-test-laravel.test/concerts
```

```
…
    <main>
        <h1>Programme</h1>
            <article class="concert-card">
    <h2>Duo Mistral</h2>
    <p>Chapiteau, 18 h</p>
    </article>
        <a href="http://premier-test-laravel.test/concerts/1">Voir la fiche</a>
            <article class="concert-card">
    <h2>L&#039;Orchestre du Lac</h2>
    <p>Grande scène, 20 h</p>
            <p class="note">Entracte de 15 minutes</p>
    </article>
        <a href="http://premier-test-laravel.test/concerts/2">Voir la fiche</a>
…
```

BASH

```
curl -s http://premier-test-laravel.test/concerts/2
```

```
…
    <main>
        <!-- Ce commentaire HTML arrive dans la page -->
    <h1>L&#039;Orchestre du Lac</h1>
    <p>Grande scène, 20 h</p>

            <p>Long concert : 90 minutes, avec entracte.</p>
…
    <p>Un orchestre de trente musiciens, <strong>gratuit</strong> pour les moins de 12 ans.
</p>
    </main>
…
```

Le HTML est celui du chapitre Views, sans la liste des invités, et `/concerts/9` répond toujours 404. Les routes n'ont pas changé :

BASH

```
php artisan route:list --path=concerts
```

```
  GET|HEAD       concerts ............................... concerts.index ›
ConcertController@index
  GET|HEAD       concerts/{id} ........................... concerts.show ›
ConcertController@show

                                                                    Showing [2]
routes
```

7. Modifier et supprimer

Pour modifier un concert, lisez-le, changez une propriété, puis appelez `save()` :

```php
Route::get('/bac-a-sable', function () {
    concert=Concert::find(1);concert = Concert::find(1);
    concert=Concert::find(1);concert→duration = 60;
    concert->save();dd(concert→save();

    dd(concert->save();dd(concert);
});
```

Dépliez `#attributes` (sortie raccourcie) :

```
  #attributes: array:8 [▼
    "id" ⇒ 1
…
    "duration" ⇒ 60
…
    "created_at" ⇒ "2026-09-28 21:35:32"
    "updated_at" ⇒ "2026-09-28 21:38:29"
  ]
```

`save()` envoie à la base une requête `UPDATE` avec la nouvelle durée. Eloquent y ajoute `updated_at`, mis à l'heure de l'enregistrement, et `created_at` garde l'heure de la création. La fiche `/concerts/1` affiche maintenant `Concert de 60 minutes, sans entracte.` Au chapitre 16, un formulaire appellera cette écriture à votre place : la personne qui gère un site changera une valeur sans toucher au PHP.

La méthode `update()` modifie plusieurs colonnes avec un tableau, comme `create()`, et passe donc elle aussi par `#[Fillable]`. La méthode `delete()` supprime la ligne de l'objet :

```php
Route::get('/bac-a-sable', function () {
    concert=Concert::find(1);concert = Concert::find(1);
    concert=Concert::find(1);concert→update(['stage' ⇒ 'Kiosque', 'duration' ⇒ 30]);

    Concert::find(3)→delete();

    dd(Concert::count());
});
```

```
2 // routes/web.php:187
```

Le programme n'affiche plus que deux concerts, et `/concerts/3` répond 404. Rechargez la route bac à sable :

```
Call to a member function delete() on null
```

Le concert 3 n'existe plus : `find(3)` a renvoyé `null`, qui n'a pas de méthode `delete()`. Remettez les trois concerts d'origine :

```bash
php artisan migrate:fresh --seed
```

La sortie est celle de la section 4. Supprimez enfin la route `/bac-a-sable` et l'import de `Concert` dans `routes/web.php`. Cette adresse modifie la base à chaque visite, et un site ne garde pas une telle route.

À retenir

- Une migration décrit une table avec `up()` et `down()`, et `migrate` n'applique que celles que la table `migrations` ne contient pas encore.
- Un modèle comme `Concert` lit la table `concerts`, dont Eloquent déduit le nom avec les règles du pluriel anglais.
- `create()` et `update()` ne remplissent que les colonnes que `#[Fillable]` déclare.
- `get()` et `all()` renvoient une collection, `find()` et `first()` un objet ou `null`, et `findOrFail()` répond 404 à la place de `null`.
- `migrate:fresh --seed` supprime toutes les tables, rejoue les migrations et relance les seeders, et seules les données d'un seeder reviennent.

Exercices

Comptez 85 minutes pour les exercices, en séance ou à la maison, et 10 minutes de plus la première fois pour installer le dépôt.

Les katas 10.1 à 10.6 se font dans le dépôt `eloquent-26`, un projet Laravel 13 qui ne contient que des tests, sur le cinéma de l'exercice 1 du chapitre Controllers.

Installer le dépôt (10 minutes, une seule fois)

Sur github.com/opmvpc/eloquent-26, créez votre copie avec le bouton Fork. Ouvrez ensuite un terminal dans le dossier de Herd ou de Laragon, celui qui contient `premier-test-laravel`, et clonez votre copie :

```bash
git clone https://github.com/VOTRE-COMPTE/eloquent-26.git
cd eloquent-26
```

Le dépôt ne contient ni `vendor/`, ni `.env`, ni `database.sqlite` : ces fichiers sont exclus de Git, comme vous l'avez vu au chapitre Installation. Chaque projet cloné se prépare avec les mêmes étapes, que `laravel new` avait lancées pour vous :

```bash
composer install
cp .env.example .env
php artisan key:generate
php artisan migrate
```

- `composer install` recrée `vendor/` à partir de `composer.lock`. Sans ce dossier, `php artisan` s'arrête sur `Failed opening required '…/vendor/autoload.php'`.
- `cp .env.example .env` copie le modèle de configuration versionné vers votre `.env`, qui reste hors Git. La commande fonctionne dans PowerShell comme dans le Terminal de macOS.
- `php artisan key:generate` remplit la ligne `APP_KEY=` de `.env`. Sans clé, chaque page répond 500 avec `MissingAppKeyException`.
- `php artisan migrate` remarque que la base n'existe pas encore et propose de la créer :

```
   WARN  The SQLite database configured for this application does not exist: …/eloquent-
26/database/database.sqlite.

 Would you like to create it? (yes/no) [yes]
›
   INFO  Preparing database.

  Creating migration table ..................................... 12.38ms DONE

   INFO  Running migrations.

  0001_01_01_000000_create_users_table ........................... 24.88ms DONE
  0001_01_01_000001_create_cache_table ........................... 18.41ms DONE
  0001_01_01_000002_create_jobs_table ............................ 27.93ms DONE
```

Appuyez sur Entrée pour répondre `yes`. Laravel crée `database/database.sqlite` et y applique les trois migrations livrées avec le projet. Les tables du cinéma viendront de vos migrations. Avec Herd, le site répond aussitôt à l'adresse `http://eloquent-26.test`. Avec Laragon, rechargez-le depuis son menu pour qu'il découvre le nouveau dossier.

Lancer les tests

Les tests du chapitre forment le groupe `chapitre10` :

```bash
php artisan test --group=chapitre10
```

Au départ, les 18 tests sont rouges. Ils tournent sur leur propre base, en mémoire, et ne touchent pas à `database.sqlite`. Pour voir vos pages à l'adresse `http://eloquent-26.test/films`, lancez `php artisan migrate:fresh --seed`. Le README du dépôt explique comment lire un test rouge et rappelle les noms que les tests imposent.

10.1 La table des films (10 minutes)

Créez la migration `create_movies_table`. En plus de `id()` et `timestamps()`, elle ajoute quatre colonnes :

- `title`, un texte court,
- `duration`, une durée en minutes,
- `released_on`, une date sans heure,
- `synopsis`, un texte long qui peut rester vide.

Appliquez-la. Inspirez-vous de la section 2.

```bash
php artisan make:migration create_movies_table
```

Test	Ce que vous écrivez
la table movies existe avec les colonnes title, duration, released_on, synopsis et les timestamps	les quatre colonnes dans `up()`
duration est une colonne d'entiers et released_on une colonne de dates	`unsignedSmallInteger()` et `date()`
un film peut être enregistré sans synopsis (synopsis accepte null)	`text('synopsis')->nullable()`

10.2 Le modèle (10 minutes)

Créez le modèle `Movie` et déclarez ses quatre colonnes avec `#[Fillable]`. Inspirez-vous de la section 3.

```bash
php artisan make:model Movie
```

Test	Ce que vous écrivez
Movie::create() écrit une ligne dans la table movies	`#[Fillable([...])]` au-dessus de la classe
un film créé avec Movie::create() se relit avec ses valeurs	les quatre noms de colonnes, sans faute
sans synopsis, Movie::create() laisse synopsis à null	rien de plus

10.3 Les cinq films (15 minutes)

Créez `MovieSeeder` avec ces cinq films, et appelez-le depuis `DatabaseSeeder`. Pour « Les Jours sans », n'écrivez pas la clé `synopsis`. Inspirez-vous de la section 4.

```bash
php artisan make:seeder MovieSeeder
```

title	duration	released_on	synopsis
Terminus Nord	112	2024-02-14	Une contrôleuse de train découvre une lettre oubliée dans un wagon vide.
La Nuit des Lucioles	98	2024-09-04	Trois amis d'enfance se retrouvent pour une dernière nuit au bord du lac.
Marée basse	134	2019-11-20	Sur la côte, un gardien de phare refuse de partir à la retraite.
Les Jours sans	87	2011-03-09	aucun
Sous le pont	105	2003-06-25	Un vieux musicien apprend le violon à une fillette du quartier.

Test	Ce que vous écrivez
MovieSeeder crée exactement les cinq films de l'énoncé	cinq `Movie::create([...])` dans `run()`
MovieSeeder enregistre les durées, les dates de sortie et les synopsis de l'énoncé	les valeurs du tableau
DatabaseSeeder appelle MovieSeeder : après db:seed, les cinq films sont en base	`$this->call(MovieSeeder::class);`

10.4 La liste (15 minutes)

Créez `MovieController` avec une méthode `index()`, qui passe les films triés par titre à la vue `movies.index`. La vue les affiche dans un `<table>`, une ligne `<tr>` par film. Reliez `index()` à `GET /films`, route nommée `movies.index`. Inspirez-vous de la section 6.

Le dépôt ne contient aucun layout. Écrivez dans la vue une page HTML complète, ou créez d'abord votre layout comme au chapitre Views, avec `php artisan make:component layouts.app --view`.

```bash
php artisan make:controller MovieController
php artisan make:view movies.index
```

Test	Ce que vous écrivez
la page /films répond 200 et affiche les cinq titres dans l'ordre alphabétique	`Movie::orderBy('title')->get()` et un `@foreach`
la page /films présente les films dans un tableau HTML	`<table>`, puis `<tr>` et `<td>` dans la boucle
la route /films s'appelle movies.index	`->name('movies.index')`

10.5 La fiche (15 minutes)

Ajoutez `show(int $id)` et la route `GET /films/{id}`, avec `whereNumber('id')`, nommée `movies.show`. La vue `movies.show` affiche le titre, la durée en heures et minutes, et le synopsis.

```bash
php artisan make:view movies.show
```

La fonction PHP `intdiv()` donne le quotient d'une division entière, et `%` son reste :

```xml
<p>{{ intdiv(movie->duration, 60) }} h {{ movie->duration % 60 }}</p>
```

Test	Ce que vous écrivez
la fiche d'un film affiche son titre, sa durée en « 1 h 52 » et son synopsis	`Movie::findOrFail($id)` et la vue
la durée d'un autre film est aussi écrite en heures et minutes : 98 minutes donnent « 1 h 38 »	la ligne ci-dessus
un numéro de film qui n'existe pas répond 404 (findOrFail)	`findOrFail()`, pas `find()`
la route /films/{id} s'appelle movies.show	`->name('movies.show')`

10.6 Le filtre par année (10 minutes)

`/films?annee=2024` n'affiche que les films sortis en 2024, et « Aucun film » si aucun ne correspond. Sans `?annee=`, la liste reste complète. Dans `index(Request $request)`, lisez l'année avec `$request->query('annee')`, comme au chapitre Controllers, et importez `Illuminate\Http\Request`. `whereYear()` fonctionne comme `where()`, mais ne compare que l'année d'une date. Complétez la requête avant `get()` :

```php
year=year = year=request->query('annee');
query=Movie::orderBy('title');if(query = Movie::orderBy('title');

if (query=Movie::orderBy('title');if(year !== null) {
    query->whereYear('releasedon',query->whereYear('released_on',
query->whereYear('releasedon',year);
}
```

Test	Ce que vous écrivez
/films?annee=2024 ne garde que les deux films sortis en 2024	les lignes ci-dessus, puis `$query->get()`
/films?annee=1999 n'affiche aucun titre, seulement le message « Aucun film »	`@forelse` et `@empty` dans la vue

10.7 Prédire cinq requêtes (sur papier, 10 minutes)

La table `concerts` contient les trois concerts du seeder. Écrivez ce que renvoie chaque ligne : un nombre, un texte, `null`, un objet ou une collection, avec son contenu.

```php
Concert::count();
Concert::find(3)->stage;
Concert::where('stage', 'Grande scène')->get();
Concert::orderBy('duration')->first()->artist;
Concert::find(4);
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/modeles-une-table

10.8 Six cartes

Que fait `php artisan migrate` ?

Il exécute la méthode `up()` de chaque migration absente de la table `migrations`, puis il l'y inscrit.

Quelle table lit le modèle `ParkingRate` ?

`parking_rates` : Eloquent met le nom de la classe au pluriel anglais, en minuscules, avec un `_` entre les mots.

Pourquoi écrire `#[Fillable]` ?

Pour qu'un visiteur ne puisse pas remplir une colonne que le formulaire ne prévoit pas. `create()` et `update()` ignorent les clés absentes de la liste. Sans liste, ils lèvent une `MassAssignmentException`.

`find()` ou `findOrFail()` dans un contrôleur ?

`findOrFail()` : il répond 404 quand la ligne n'existe pas. `find()` renverrait `null`, et la vue s'arrêterait sur une erreur 500.

Que renvoient `get()` et `first()` ?

`get()` renvoie une collection d'objets, peut-être vide. `first()` renvoie un seul objet, ou `null`.

Que fait `php artisan migrate:fresh --seed` ?

Il supprime toutes les tables, rejoue toutes les migrations, puis lance `DatabaseSeeder`. Les lignes qui ne viennent pas d'un seeder sont perdues.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/modeles-une-table

3.11 Modèles : deux tables

Bloc Laravel · séance 4 · ~50 min

Slides du chapitre (PDF) PDF

Au chapitre 10, chaque concert recopie le nom de sa scène, en toutes lettres, dans la colonne `stage`. Dans ce chapitre, les scènes reçoivent leur propre table, `stages`, et chaque concert ne garde que le numéro de sa scène.

À la fin de ce chapitre, vous serez capables de :

1. **écrire** une migration qui ajoute une clé étrangère à une table existante,
2. **déclarer** une relation avec `hasMany` et `belongsTo`, et la **lire** dans un diagramme de classes,
3. **remplir** la base avec des fabriques et Faker,
4. **lire** les données liées avec `$concert->stage` et `$stage->concerts`,
5. **afficher** une page qui liste les concerts d'une scène.

1. Le nom de la scène, recopié

Recréez la route bac à sable du chapitre 10, avec l'import de `Concert` :

```php
use App\Models\Concert;
```

```php
Route::get('/bac-a-sable', function () {
    dd(Concert::where('stage', 'Grande scène')->count());
});
```

```
2 // routes/web.php:182
```

Deux concerts se jouent sur la Grande scène : L'Orchestre du Lac et Orage Mécanique. Ajoutez maintenant un concert dont le nom de scène porte une majuscule de trop :

```php
Route::get('/bac-a-sable', function () {
    Concert::create([
        'artist' => 'Les Cordes Sauvages',
        'stage' => 'Grande Scène',
        'starts_at' => '2026-10-16 21:00:00',
        'duration' => 60,
        'description' => 'Violons et contrebasse.',
    ]);

    dump(Concert::count());
    dd(Concert::where('stage', 'Grande scène')->count());
});
```

```
4 // routes/web.php:190

2 // routes/web.php:191
```

Chaque rechargement ajoute un concert : ouvrez la page une seule fois. La table contient quatre concerts, mais `where()` n'en trouve toujours que deux sur la Grande scène. SQLite compare les textes caractère par caractère, et `S` n'est pas `s`. La page `/concerts` affiche maintenant trois noms de scène : Chapiteau, Grande scène et Grande Scène.

Le code qui sent mauvais

En binôme, cinq minutes. L'organisatrice renomme la Grande scène en « Scène du Lac ». Elle veut aussi afficher sur la fiche de chaque concert la capacité de sa scène, 3 000 places pour celle-ci. Avec la table `concerts` du chapitre 10, que devez-vous modifier pour satisfaire ces deux demandes ?

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/modeles-deux-tables

Aux chapitres 4 et 5 de la POO, vous avez relié deux classes par un trait, avec un rôle et une multiplicité à chaque bout. Ici, une scène accueille plusieurs concerts, et chaque concert se joue sur une seule scène. Vous écrivez maintenant ce lien dans la base, puis dans les modèles.

2. La table des scènes et la clé étrangère

Créez d'abord la table des scènes :

```bash
php artisan make:migration create_stages_table
```

```
   INFO  Migration [database/migrations/2026_09_28_220036_create_stages_table.php] created
successfully.
```

```php
Schema::create('stages', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->unsignedSmallInteger('capacity');
    $table->string('location');
    $table->timestamps();
});
```

`capacity` compte des places, et `location` situe la scène dans le parc.

Une migration qui modifie `concerts`

La table `concerts` existe déjà : pour la changer, écrivez une nouvelle migration, comme au chapitre 10. Artisan reconnaît la fin `_to_concerts_table` du nom, et l'option `--table=concerts` le dit en toutes lettres :

```bash
php artisan make:migration add_stage_id_to_concerts_table --table=concerts
```

```
   INFO  Migration
[database/migrations/2026_09_28_220038_add_stage_id_to_concerts_table.php] created
successfully.
```

Le fichier créé appelle `Schema::table('concerts', …)` dans `up()` et dans `down()`. Cet appel modifie une table existante, là où `Schema::create()` en crée une. Remplacez le commentaire vide des deux méthodes :

```php
public function up(): void
{
    Schema::table('concerts', function (Blueprint table) {
        $table->foreignId('stage_id')->constrained()->cascadeOnDelete();
        $table->dropColumn('stage');
    });
}

public function down(): void
{
    Schema::table('concerts', function (Blueprint table) {
        table->dropConstrainedForeignId('stageid');table
->dropConstrainedForeignId('stage_id');
        table->dropConstrainedForeignId('stageid');table->string('stage');
    });
}
```

- **`foreignId('stage_id')`** ajoute une colonne d'entiers, `stage_id`, qui contiendra l'`id` d'une ligne de `stages`. Si la Grande scène porte l'`id` 2, ses concerts auront un `stage_id` égal à 2. Une colonne qui contient l'identifiant d'une ligne d'une autre table s'appelle une **clé étrangère**.
- **`constrained()`** ajoute une contrainte : la base refusera un `stage_id` qui ne correspond à aucune ligne de `stages`. Laravel trouve le nom de cette table en retirant `_id` du nom de la colonne, puis en mettant le reste au pluriel.
- **`cascadeOnDelete()`** demande à la base de supprimer aussi les concerts d'une scène supprimée.
- **`dropColumn('stage')`** supprime la colonne de texte, remplacée par la clé.
- **`down()`** fait l'inverse : `dropConstrainedForeignId('stage_id')` retire la clé et sa colonne, puis `string('stage')` remet la colonne de texte.

Pour le festival, un concert n'existe pas sans sa scène : si le Chapiteau n'est pas monté, ses concerts n'ont pas lieu. `cascadeOnDelete()` écrit cette règle dans la base.

Une table déjà remplie

Appliquez les deux migrations :

```bash
php artisan migrate
```

```
   INFO  Running migrations.

  2026_09_28_220036_create_stages_table ......................... 7.71ms DONE
  2026_09_28_220038_add_stage_id_to_concerts_table ............... 4.15ms FAIL

   Illuminate\Database\QueryException

  SQLSTATE[HY000]: General error: 1 Cannot add a NOT NULL column with default value NULL
(Connection: sqlite, …, SQL: alter table "concerts" add column "stage_id" integer not null)
…
```

La table `stages` existe, mais la seconde migration a échoué. Les quatre concerts de la table recevraient `null` dans la nouvelle colonne. Or `stage_id`, comme toute colonne sans `nullable()`, refuse `null` : SQLite refuse donc de l'ajouter. `migrate:status` montre l'état exact (sortie raccourcie) :

```bash
php artisan migrate:status
```

```
  Migration name ..................................... Batch / Status
…
  2026_09_28_213249_create_concerts_table ........................ [1] Ran
  2026_09_28_220036_create_stages_table .......................... [2] Ran
  2026_09_28_220038_add_stage_id_to_concerts_table ................ Pending
```

Videz la base et appliquez toutes les migrations avec `migrate:fresh`, cette fois sans `--seed` : `ConcertSeeder` écrit encore dans la colonne `stage`, qui va disparaître. Le seeder, réécrit plus loin, recréera les trois concerts du festival avec leur scène (sortie raccourcie).

```bash
php artisan migrate:fresh
```

```
  Dropping all tables ..................................... 3.50ms DONE
…
  2026_09_28_213249_create_concerts_table ................... 3.84ms DONE
  2026_09_28_220036_create_stages_table ..................... 3.62ms DONE
  2026_09_28_220038_add_stage_id_to_concerts_table ........... 24.25ms DONE
```

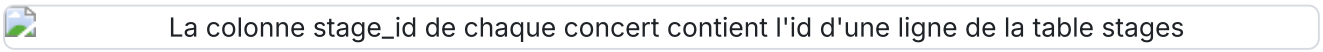
Sur une table vide, aucune ligne ne doit recevoir de `stage_id`, et l'ajout réussit. Laravel trie les migrations par leur nom, qui commence par leur date de création : `create_stages_table` passe en premier, et `stages` existe avant la clé.

> **ATTENTION**
>
> Sur une table remplie, `migrate:rollback` échoue aussi : `down()` recrée la colonne `stage`, qui refuse `null` comme `stage_id`, et SQLite affiche la même erreur. Dans ce chapitre, revenez en arrière avec `migrate:fresh`, jamais sur une base dont vous gardez les données.

```bash
php artisan db:table concerts
```

```
  main.concerts ..............................................................
  Columns ..................................................................... 8

  Column ...................................................................... Type
  id integer, autoincrement ................................................... integer
  artist varchar .............................................................. varchar
  starts_at datetime .......................................................... datetime
…
  stage_id integer ............................................................ integer
…
  Foreign Key ................................................. On Update / On Delete
  stage_id references id on stages ........................... no action / cascade
```

La colonne `stage` a disparu, et `stage_id` s'est ajoutée en dernière position. La ligne `Foreign Key` se lit ainsi : `stage_id` désigne la colonne `id` de `stages`, et la suppression d'une scène se propage à ses concerts, `cascade`.


La colonne stage_id de chaque concert contient l'id d'une ligne de la table stages

3. Deux modèles, une relation

Créez le modèle des scènes :

```bash
php artisan make:model Stage
```

```
   INFO  Model [app/Models/Stage.php] created successfully.
```

Donnez à `Stage` sa liste `#[Fillable]` et une méthode `concerts()` :

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Attributes\Fillable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

#[Fillable(['name', 'capacity', 'location'])]
class Stage extends Model
{
    public function concerts(): HasMany
    {
        return $this->hasMany(Concert::class);
    }
}
```

La méthode `concerts()` est une **relation** Eloquent : une méthode de modèle qui décrit comment retrouver, dans une autre table, les lignes liées à l'objet.

Dans `Concert`, remplacez `'stage'` par `'stage_id'` dans `#[Fillable]`. Ajoutez l'import de `BelongsTo` sous les imports existants, puis la méthode `stage()` sous `casts()` :

```php
use Illuminate\Database\Eloquent\Relations\BelongsTo;

#[Fillable(['artist', 'stage_id', 'starts_at', 'duration', 'description'])]
class Concert extends Model
{
    protected function casts(): array
    {
        return [
            'starts_at' => 'datetime',
        ];
    }

    public function stage(): BelongsTo
    {
        return $this->belongsTo(Stage::class);
    }
}
```

Chaque méthode renvoie un objet qui prépare la recherche des lignes liées. Sa classe, `HasMany` ou `BelongsTo`, sert de type de retour.

- **`hasMany(Concert::class)`** se lit « une scène a plusieurs concerts ». Pour trouver les concerts d'une scène, Eloquent cherche dans la table `concerts` les lignes dont `stage_id` vaut l'`id` de la scène. Il déduit le nom `stage_id` du nom de la classe `Stage`, écrit comme le nom de sa table au singulier, `stage`, et suivi de `_id`. `ParkingRate` donnerait `parking_rate_id`.

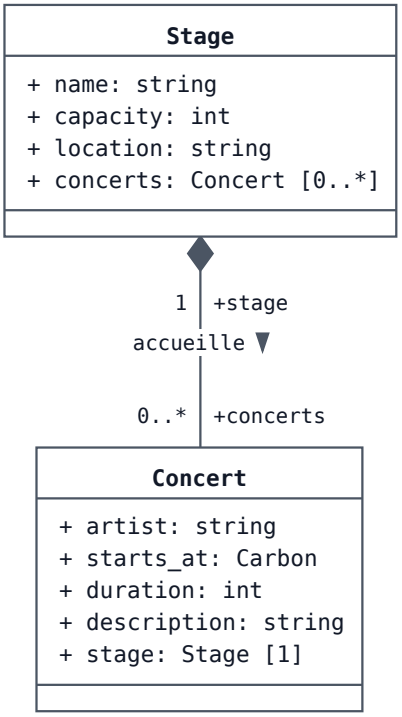
- `belongsTo(Stage::class)` se lit « un concert appartient à une scène ». Eloquent lit le `stage_id` du concert, puis cherche dans la table `stages` la ligne dont l'`id` vaut ce nombre. Il déduit le nom `stage_id` du nom de la méthode, `stage`, suivi de `_id`.

Les deux déductions donnent `stage_id`, le nom choisi dans la migration : vous n'écrivez aucun nom de colonne. Une méthode appelée `scene()` chercherait une colonne `scene_id`, qui n'existe pas.

La classe `Concert` n'a ni propriété ni colonne `stage`. Quand vous lisez `$concert->stage`, sans parenthèses, la méthode magique `__get()`, vue au chapitre 3 de la POO, appelle donc `stage()` et renvoie la scène trouvée. `$stage->concerts` fonctionne de la même façon.

hasMany cherche dans concerts, belongsTo cherche dans stages, et les deux passent par stage_id

Le diagramme de classes



Le trait se lit « une scène accueille zéro, un ou plusieurs concerts », et chaque concert a exactement une scène. Chaque bout porte le rôle que le code utilise : `concerts` est le nom que `Stage` donne à ses concerts, et `stage` celui que `Concert` donne à sa scène.

Comme au chapitre 5 de la POO, chaque rôle revient dans la boîte de la classe qui le porte. `stage_id` n'apparaît pas : dans un diagramme de classes, c'est le trait qui relie les deux classes.

Le losange plein ◆———, collé à `Stage`, marque une composition : un concert n'existe pas sans sa scène et disparaît avec elle, comme le sac du héros. `cascadeOnDelete()` est la traduction, en base, de la composition dessinée au chapitre 5 de la POO.

La base vérifie la clé

La table `stages` est encore vide. Dans la route bac à sable, essayez d'enregistrer un concert sur la scène 1 :

```php
Route::get('/bac-a-sable', function () {
    Concert::create([
        'stage_id' => 1,
        'artist' => 'Les Cordes Sauvages',
        'starts_at' => '2026-10-16 21:00:00',
        'duration' => 60,
        'description' => 'Violons et contrebasse.',
    ]);

    return 'Concert enregistré';
});
```

```
Illuminate\Database\QueryException
SQLSTATE[23000]: Integrity constraint violation: 19 FOREIGN KEY constraint failed
(Connection: sqlite, …, SQL: insert into "concerts" ("stage_id", "artist", "starts_at",
"duration", "description", "updated_at", "created_at") values (1, Les Cordes Sauvages, …))
```

La base a refusé la ligne, car aucune scène ne porte l'`id` 1. C'est `constrained()` qui a posé cette règle : **une clé étrangère désigne toujours une ligne qui existe.**

4. Les fabriques et Faker

Le programme d'un vrai festival compte une vingtaine de concerts, soit vingt appels à `Concert::create()`. Une fabrique, vue au chapitre Structure, est une classe qui produit des objets d'un modèle avec des valeurs inventées. `DatabaseSeeder` utilise déjà `UserFactory` quand il appelle `User::factory()` pour créer `Test User`. Créez une fabrique par modèle :

```bash
php artisan make:factory StageFactory --model=Stage
php artisan make:factory ConcertFactory --model=Concert
```

```
   INFO  Factory [database/factories/StageFactory.php] created successfully.

   INFO  Factory [database/factories/ConcertFactory.php] created successfully.
```

```php
<?php

namespace Database\Factories;

use App\Models\Stage;
use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * @extends Factory<Stage>
 */
class StageFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
        return [
            //
        ];
    }
}
```

`StageFactory` hérite de `Factory`, et le commentaire `@extends Factory<Stage>` indique à l'éditeur le modèle produit.

La méthode **`definition()`** renvoie un tableau avec une clé par colonne, comme celui que vous passez à `create()`. La fabrique l'appelle une fois pour chaque objet à produire. Remplissez-la :

```php
public function definition(): array
{
    return [
        'name' => fake()->randomElement(['Kiosque', 'Clairière', 'Verger', 'Serre']),
        'capacity' => fake()->numberBetween(100, 2000),
        'location' => fake()->streetName(),
    ];
}
```

La fonction `fake()` renvoie un objet de **Faker**, une bibliothèque installée pour le développement dans chaque projet Laravel, qui invente des données. Chaque appel donne une nouvelle valeur :

Vous écrivez	Faker renvoie
`fake()->randomElement([...])`	un élément du tableau, pris au hasard
`fake()->numberBetween(100, 2000)`	un entier entre les deux bornes comprises
`fake()->streetName()`	un nom de rue
`fake()->name()`	un prénom et un nom

Vous écrivez	Faker renvoie
`fake()->dateTimeBetween('…', '…')`	une date et une heure entre deux moments
`fake()->sentence()`	une phrase de faux latin

Dans `routes/web.php`, importez `Stage` sous `Concert`, et demandez trois scènes à la fabrique :

```php
Route::get('/bac-a-sable', function () {
    dd(Stage::factory()->count(3)->create());
});
```

```
BadMethodCallException
Call to undefined method App\Models\Stage::factory()
```

La méthode statique `factory()` vient du trait `HasFactory`, et les traits sont vus au chapitre 7 de la POO. `User` utilise déjà ce trait, mais `make:model` ne l'ajoute pas. Collez-le dans `Stage`, et de la même façon dans `Concert` :

```php
use Illuminate\Database\Eloquent\Factories\HasFactory;

#[Fillable(['name', 'capacity', 'location'])]
class Stage extends Model
{
    use HasFactory;

    // concerts() ne change pas
}
```

Rechargez la page, puis dépliez le premier élément et ses `#attributes` (sortie raccourcie) :

```
Illuminate\Database\Eloquent\Collection {#1565 ▼ // routes/web.php:183
  #items: array:3 [▼
    0 => App\Models\Stage {#1548 ▼
…
      #attributes: array:6 [▼
        "name" => "Verger"
        "capacity" => 472
        "location" => "Lane Harbors"
        "updated_at" => "2026-09-28 22:42:15"
        "created_at" => "2026-09-28 22:42:15"
        "id" => 1
      ]
…
```

`count(3)` demande trois objets, et `create()` les enregistre puis les renvoie dans une collection. `factory()` trouve la classe `StageFactory` d'après le nom du modèle, suivi de `Factory`. Chez vous, les valeurs sont différentes, et chaque rechargement ajoute trois scènes.

La langue des données

« Lane Harbors » est une rue américaine. La ligne `APP_FAKER_LOCALE` du fichier `.env` fixe la langue et le pays des données de Faker, et vaut `en_US` dans un projet neuf. Passez au français de Belgique :

```
APP_FAKER_LOCALE=fr_BE
```

Rechargez la page. Les trois nouvelles scènes, numérotées de 4 à 6, ont des noms de rues belges (extraits) :

```
        "location" => "chemin Lecomte"
…
        "location" => "impasse Claeys"
…
        "location" => "place Dierckx"
```

Les noms de personnes suivent aussi ce réglage, mais `sentence()` reste en faux latin.

Une fabrique qui crée sa scène

Remplissez la fabrique des concerts, et importez `Stage` sous `Concert` :

```php
public function definition(): array
{
    return [
        'stage_id' => Stage::factory(),
        'artist' => fake()->name(),
        'starts_at' => fake()->dateTimeBetween('2026-10-16 17:00', '2026-10-16 23:00')
->format('Y-m-d H:00:00'),
        'duration' => fake()->randomElement([45, 60, 75, 90]),
        'description' => fake()->sentence(),
    ];
}
```

- `'stage_id' => Stage::factory()` donne une fabrique au lieu d'un nombre. Au moment d'enregistrer le concert, Laravel lance la fabrique des scènes, enregistre une nouvelle scène, puis place son `id` dans `stage_id`.
- `starts_at` reçoit une heure du vendredi soir, entre 17 h et 23 h. `dateTimeBetween()` renvoie un objet date, dont la méthode `format()` fonctionne comme celle de Carbon au chapitre 10. `Y-m-d` écrit la date, `H` l'heure sur deux chiffres, et `00:00` est recopié tel quel.

```php
Route::get('/bac-a-sable', function () {
    dump(Stage::count());
    concert=Concert::factory()->create();dump(Stage::count());dd(concert =
Concert::factory()→create();
    dump(Stage::count());
    dd(concert=Concert::factory()->create();dump(Stage::count());dd(concert);
});
```

Dépliez `#attributes` (sortie raccourcie) :

```
6 // routes/web.php:183

7 // routes/web.php:185

App\Models\Concert {#1464 ▼ // routes/web.php:186
…
  #attributes: array:8 [▼
    "stage_id" ⇒ 7
    "artist" ⇒ "Antoine Renard"
    "starts_at" ⇒ "2026-10-16 22:00:00"
    "duration" ⇒ 90
    "description" ⇒ "Tempore repellat non sed reprehenderit."
…
```

La table comptait six scènes, celles des deux essais précédents. Pour un seul concert, la fabrique a créé une septième scène, et le concert a reçu son `id`, 7.

5. Le seeder avec les fabriques

Créez `StageSeeder`, qui enregistre les deux vraies scènes du festival :

```bash
php artisan make:seeder StageSeeder
```

```
   INFO  Seeder [database/seeders/StageSeeder.php] created successfully.
```

Importez `App\Models\Stage` en haut du fichier, puis remplissez `run()` :

```php
public function run(): void
{
    Stage::create([
        'name' => 'Chapiteau',
        'capacity' => 400,
        'location' => 'Pelouse est',
    ]);

    Stage::create([
        'name' => 'Grande scène',
        'capacity' => 3000,
        'location' => 'Bord du lac',
    ]);
}
```

Dans `ConcertSeeder`, chaque concert reçoit le numéro de sa scène, retrouvée par son nom avec `where()` et `first()`. Importez `Stage` sous `Concert`, et remplacez la méthode `run()` :

```php
public function run(): void
{
    tent=Stage::where('name','Chapiteau')->first();tent = Stage::where('name', 'Chapiteau')
->first();
    tent=Stage::where('name','Chapiteau')->first();mainStage = Stage::where('name', 'Grande
scène')->first();

    Concert::create([
        'stage_id' => tent->id,'artist'⇒'DuoMistral','startsat'⇒'2026-10-1618:00:00',
'duration'⇒45,'description'⇒'Guitareetviolon,<strong>entre´elibre</strong>.
',]);Concert::create(['stageid'⇒tent→id,
        'artist' => 'Duo Mistral',
        'starts_at' => '2026-10-16 18:00:00',
        'duration' => 45,
        'description' => 'Guitare et violon, <strong>entrée libre</strong>.',
    ]);

    Concert::create([
        'stage_id' => tent->id,'artist'⇒'DuoMistral','startsat'⇒'2026-10-1618:00:00',
'duration'⇒45,'description'⇒'Guitareetviolon,<strong>entre´elibre</strong>.
',]);Concert::create(['stageid'⇒mainStage→id,
        'artist' => "L'Orchestre du Lac",
        // … starts_at, duration et description du chapitre 10
    ]);

    Concert::create([
        'stage_id' => mainStage->id,'artist'⇒'OrageMe´canique',//…
startsat,durationetdescriptionduchapitre10]);foreach(Stage::all()asmainStage→id,
        'artist' => 'Orage Mécanique',
        // … starts_at, duration et description du chapitre 10
    ]);

    foreach (Stage::all() as mainStage->id,'artist'⇒'OrageMe´canique',//…startsa
t,durationetdescriptionduchapitre10]);foreach(Stage::all()asstage) {
        Concert::factory()→count(10)→create(['stage_id' => $stage→id]);
    }
}
```

La boucle demande dix concerts de fabrique pour chaque scène. Le tableau passé à `create()` remplace les valeurs de `definition()` : chaque concert reçoit l'`id` de la scène en cours, sans lancer `Stage::factory()`. Dans `DatabaseSeeder`, appelez `StageSeeder` avant `ConcertSeeder` :

```php
this->call(StageSeeder::class);this→call(StageSeeder::class);
this->call(StageSeeder::class);this→call(ConcertSeeder::class);
```

Dans l'ordre inverse, la table `stages` serait vide au moment où `ConcertSeeder` la lit. `first()` renverrait `null`, et le seeder s'arrêterait sur `Attempt to read property "id" on null`. Remettez la base à neuf (sortie raccourcie) :

```bash
php artisan migrate:fresh --seed
```

```
…
  2026_09_28_220038_add_stage_id_to_concerts_table ..... 24.55ms DONE


   INFO  Seeding database.

  Database\Seeders\StageSeeder ..................................... RUNNING
  Database\Seeders\StageSeeder ..................................... 8 ms DONE

  Database\Seeders\ConcertSeeder ................................... RUNNING
  Database\Seeders\ConcertSeeder ................................... 92 ms DONE
```

Comptez les lignes dans la route bac à sable :

```php
Route::get('/bac-a-sable', function () {
    dump(Stage::count());
    dd(Concert::count());
});
```

```
2 // routes/web.php:183

23 // routes/web.php:184
```

La base contient deux scènes et vingt-trois concerts : les trois du chapitre 10, puis dix concerts de fabrique pour chacune des deux scènes. **Écrivez à la main les concerts que les pages doivent toujours montrer, et laissez une fabrique inventer les lignes dont les valeurs importent peu.**

6. Lire à travers la relation

Lisez la scène du concert 1, Duo Mistral :

```php
Route::get('/bac-a-sable', function () {
    concert=Concert::find(1);dump(concert = Concert::find(1);

    dump(concert=Concert::find(1);dump(concert→stage_id);
    dump(concert->stage->name);dump(concert→stage→name);
    dump(concert->stage->name);dump(concert→stage);
    dump(concert->stage());dd(concert→stage());
    dd(concert->stage());dd(concert→stage()→toRawSql());
});
```

Dépliez `#attributes` dans le premier objet (sortie raccourcie) :

```
1 // routes/web.php:185

"Chapiteau" // routes/web.php:186

App\Models\Stage {#1095 ▼ // routes/web.php:187
…
  #attributes: array:6 [▼
    "id" ⇒ 1
    "name" ⇒ "Chapiteau"
    "capacity" ⇒ 400
…
Illuminate\Database\Eloquent\Relations\BelongsTo {#1526 ▼ // routes/web.php:188
…
  #foreignKey: "stage_id"
  #ownerKey: "id"
…
"select * from "stages" where "stages"."id" = 1" // routes/web.php:189
```

- **`$concert->stage`**, sans parenthèses, passe par `__get()`. À la première lecture, Eloquent envoie la requête à la base et renvoie un objet `Stage`. Il garde ensuite cet objet : une seconde lecture de `$concert->stage` ne relance pas la requête.
- **`$concert->stage()`**, avec parenthèses, appelle votre méthode. Elle renvoie la relation, un objet `BelongsTo` qui décrit la recherche sans l'envoyer : la clé étrangère `stage_id` et la colonne visée, `id`.

`toRawSql()` montre la requête de la relation : Eloquent a lu le `stage_id` du concert, 1, et cherche dans `stages` la ligne dont l'`id` vaut 1. Lisez la propriété pour afficher une donnée, et appelez la méthode pour compléter la requête.

De l'autre côté, `$stage->concerts` renvoie une collection. Remplacez le contenu de la route (sortie raccourcie) :

PHP

```php
Route::get('/bac-a-sable', function () {
    stage=Stage::find(1);dump(stage = Stage::find(1);

    dump(stage=Stage::find(1);dump(stage->concerts);
    dump(stage->concerts->count());dump(stage->concerts->count());
    dump(stage->concerts->count());dump(stage->concerts()->toRawSql());

    foreach (stage->concerts()->orderBy('startsat')->get()asstage->concerts()
->orderBy('starts_at')->get() as stage->concerts()->orderBy('startsat')->get()asconcert) {
        dump(concert->starts_at->format('G \h') . ' ' . concert->artist);
    }
});
```

```
Illuminate\Database\Eloquent\Collection {#1118 ▼ // routes/web.php:185
  #items: array:11 [▶]
  #escapeWhenCastingToString: false
}

11 // routes/web.php:186

"select * from "concerts" where "concerts"."stage_id" = 1 and "concerts"."stage_id" is not
null" // routes/web.php:187

"17 h Léa Bodart" // routes/web.php:190

"17 h Lars Urbain" // routes/web.php:190

"18 h Duo Mistral" // routes/web.php:190
…
"22 h Anna Poncelet" // routes/web.php:190
```

Le Chapiteau a onze concerts : Duo Mistral et dix concerts de fabrique. La requête de `hasMany` garde les lignes de `concerts` dont `stage_id` vaut 1, l'`id` du Chapiteau. La condition `is not null` est toujours vraie ici, car la colonne refuse `null`.

`concerts()` renvoie une relation, que vous complétez comme une requête du chapitre 10 : `orderBy('starts_at')` trie les concerts, et `get()` envoie la requête.

Vous écrivez	Vous recevez
`$concert->stage`	un objet `Stage`
`$concert->stage()`	une relation `BelongsTo`, sans requête envoyée
`$stage->concerts`	une collection d'objets `Concert`, peut-être vide
`$stage->concerts()`	une relation `HasMany`, à compléter avec `orderBy()`, `get()` ou `count()`

Supprimer une scène

Vérifiez maintenant la cascade choisie dans la migration :

```php
Route::get('/bac-a-sable', function () {
    dump(Concert::count());

    Stage::where('name', 'Grande scène')->first()->delete();

    dd(Concert::count());
});
```

```
23 // routes/web.php:183

11 // routes/web.php:187
```

Aucune ligne de PHP ne supprime de concert. La base a supprimé elle-même les douze concerts de la Grande scène, à cause de `cascadeOnDelete()`. Remettez le festival en état avec `php artisan migrate:fresh --seed`. Les concerts de fabrique reçoivent de nouveaux noms.

7. Les pages

Ouvrez le programme :

```bash
curl -s http://premier-test-laravel.test/concerts
```

```
…
            <article class="concert-card">
    <h2>Duo Mistral</h2>
    <p>
{&amp;quot;id&amp;quot;:1,&amp;quot;name&amp;quot;:&amp;quot;Chapiteau&amp;quot;,&amp;quot;capacity&amp;quot;:400,&amp;quot;location&amp;quot;:&amp;quot;Pelouse est&amp;quot;,&amp;quot;created_at&amp;quot;:&amp;quot;2026-09-28T22:43:26.000000Z&amp;quot;,&amp;quot;updated_at&amp;quot;:&amp;quot;2026-09-28T22:43:26.000000Z&amp;quot;},
18 h</p>
    </article>
…
```

Le composant `concert-card` écrit toujours `{{ $concert->stage }}`, qui renvoie maintenant un objet `Stage`. Pour l'écrire dans la page, `{{ }}` appelle sa méthode `__toString()`, vue au chapitre 1 de la POO. Celle d'un modèle renvoie ses colonnes au format JSON, vu au chapitre Controllers. Dans `components/concert-card.blade.php`, affichez le nom de la scène :

```xml
<p>{{ concert->stage->name }}, {{ concert->starts_at->format('G \h') }}</p>
```

```
…
            <article class="concert-card">
    <h2>Duo Mistral</h2>
    <p>Chapiteau, 18 h</p>
    </article>
        <a href="http://premier-test-laravel.test/concerts/1">Voir la fiche</a>
…
            <article class="concert-card">
    <h2>Kobe Hermans</h2>
    <p>Grande scène, 18 h</p>
            <p class="note">Entracte de 15 minutes</p>
    </article>
…
```

Les vingt-trois cartes restent triées par heure, et `ConcertController` n'a pas changé. Pour chaque carte, la lecture de `$concert->stage` envoie une requête à la table `stages`, avec le `stage_id` du concert.

Une page par scène

Créez le contrôleur et la vue de la page d'une scène :

BASH

```bash
php artisan make:controller StageController
php artisan make:view stages.show
```

```
   INFO  Controller [app/Http/Controllers/StageController.php] created successfully.

   INFO  View [resources/views/stages/show.blade.php] created successfully.
```

Remplacez le contenu du contrôleur :

PHP

```php
<?php

namespace App\Http\Controllers;

use App\Models\Stage;
use Illuminate\View\View;

class StageController extends Controller
{
    public function show(int $id): View
    {
        $stage = Stage::findOrFail($id);
        $concerts = $stage->concerts()->orderBy('starts_at')->get();

        return view('stages.show', ['stage' => $stage, 'concerts' => $concerts]);
    }
}
```

`findOrFail()` répond 404 pour un numéro inconnu, comme au chapitre 10, et la méthode `concerts()` permet de trier les concerts par heure. Écrivez la vue :

```xml
<x-layouts.app :title="stage->name">
    <h1>{{ $stage->name }}</h1>
    <p>{{ $stage->location }}, {{ $stage->capacity }} places</p>

    <h2>Concerts</h2>
    <ul>
        @forelse (concerts as $concert)
            <li>
                {{ $concert->starts_at->format('G \h') }}
                <a href="{{ route('concerts.show', $concert->id) }}">{{ $concert->artist }}
</a>
            </li>
        @empty
            <li>Aucun concert sur cette scène</li>
        @endforelse
    </ul>
</x-layouts.app>
```

Dans `routes/web.php`, importez `StageController` sous `ConcertController`, puis ajoutez la route au-dessus de la route bac à sable :

```php
Route::get('/scenes/{id}', [StageController::class, 'show'])->whereNumber('id')
->name('stages.show');
```

```bash
curl -s http://premier-test-laravel.test/scenes/1
```

```
…
    <title>Chapiteau · Les Nuits du Parc</title>
…
        <h1>Chapiteau</h1>
    <p>Pelouse est, 400 places</p>

    <h2>Concerts</h2>
    <ul>
                    <li>
                18 h
                <a href="http://premier-test-laravel.test/concerts/1">Duo Mistral</a>
            </li>
                    <li>
                18 h
                <a href="http://premier-test-laravel.test/concerts/7">Antoine Verhaeghe</a>
            </li>
…
```

`/scenes/9` répond 404. La commande `route:list` confirme le nom de la route :

```bash
php artisan route:list --path=scenes
```

```
  GET|HEAD       scenes/{id} ..................................... stages.show › StageController@show

                                                                                Showing [1] routes
```

Sur la fiche d'un concert, faites du nom de la scène un lien. Dans `concerts/show.blade.php`, remplacez le paragraphe sous le `<h1>` :

```xml
<p>
    <a href="{{ route('stages.show', concert->stage_id) }}">{{ concert->stage->name }}</a>,
    {{ $concert->starts_at->format('G \h') }}
</p>
```

```bash
curl -s http://premier-test-laravel.test/concerts/2
```

```
…
    <h1>L&#039;Orchestre du Lac</h1>
    <p>
        <a href="http://premier-test-laravel.test/scenes/2">Grande scène</a>,
        20 h
    </p>
…
```

Le lien utilise `$concert->stage_id`, le numéro de scène que le concert contient déjà. Pour voir le bloc `@empty`, créez une scène sans concert dans la route bac à sable, puis envoyez le navigateur vers sa page avec `redirect()`, vu au chapitre Routing :

```php
Route::get('/bac-a-sable', function () {
    stage=Stage::create(['name'⇒'Kiosque',
'capacity'⇒80,'location'⇒'Entre´esud',]);returnredirect()->route('stages.show',stage =
Stage::create([
        'name' ⇒ 'Kiosque',
        'capacity' ⇒ 80,
        'location' ⇒ 'Entrée sud',
    ]);

    return redirect()->route('stages.show', stage=Stage::create(['name'⇒'Kiosque',
'capacity'⇒80,'location'⇒'Entre´esud',]);returnredirect()->route('stages.show',stage->id);
});
```

Ouvrez `/bac-a-sable`. Le navigateur arrive sur `/scenes/3` :

```
    <main>
        <h1>Kiosque</h1>
    <p>Entrée sud, 80 places</p>

    <h2>Concerts</h2>
    <ul>
                    <li>Aucun concert sur cette scène</li>
            </ul>
    </main>
```

Pour finir, supprimez la route `/bac-a-sable` et les imports de `Concert` et de `Stage` dans `routes/web.php`, comme au chapitre 10. Relancez `php artisan migrate:fresh --seed` pour retirer le Kiosque.

À retenir

- Une clé étrangère comme `stage_id` contient l'`id` d'une ligne d'une autre table, et `constrained()` interdit une valeur qui ne désigne aucune ligne.
- `cascadeOnDelete()` fait supprimer par la base les concerts d'une scène supprimée.
- `hasMany()` cherche les concerts dont `stage_id` vaut l'`id` de la scène, et `belongsTo()` cherche la scène dont l'`id` vaut le `stage_id` du concert.
- `$stage->concerts` renvoie les concerts de la scène, alors que `$stage->concerts()` renvoie la relation, que vous complétez avec `orderBy()`, `count()` ou `get()`.
- Une fabrique invente les valeurs d'un modèle avec `fake()`, et le tableau passé à `create()` remplace celles que vous voulez fixer.

Exercices

Comptez 90 minutes pour les exercices, en séance ou à la maison.

Les katas 11.1 à 11.6 se font dans votre dépôt `eloquent-26`, à la suite des katas du chapitre 10. Les films y reçoivent leurs séances : une table `showtimes` reliée à `movies`. Lancez les tests du chapitre :

```bash
php artisan test --group=chapitre11
```

Au départ, les 16 tests sont rouges. Les tests du chapitre 10 doivent rester verts : relancez `php artisan test --group=chapitre10` après chaque kata. Faites les katas dans l'ordre.

11.1 La table des séances (10 minutes)

Créez la migration `create_showtimes_table`. En plus de `id()` et `timestamps()`, elle ajoute trois colonnes :

- `movie_id`, une clé étrangère vers `movies`, qui supprime les séances d'un film supprimé,
- `starts_at`, une date avec l'heure,

- `price`, un prix avec deux décimales.

Appliquez-la. Inspirez-vous de la section 2.

```bash
php artisan make:migration create_showtimes_table
```

Test	Ce que vous écrivez
la table showtimes existe avec les colonnes movie_id, starts_at, price et les timestamps	les trois colonnes dans `up()`
starts_at est une date avec heure et price un nombre décimal	`dateTime()` et `decimal('price', 5, 2)`
une séance ne peut pas pointer vers un film qui n'existe pas (clé étrangère)	`foreignId('movie_id')->constrained()`
supprimer un film supprime aussi ses séances (cascadeOnDelete)	`->cascadeOnDelete()`

11.2 Les deux relations (10 minutes)

Créez le modèle `Showtime` et déclarez ses trois colonnes, `movie_id`, `starts_at` et `price`, avec `#[Fillable]`. Donnez-lui la méthode `casts()` du chapitre 10, avec `'starts_at' => 'datetime'`. Ajoutez ensuite `movie()` dans `Showtime` et `showtimes()` dans `Movie`. Inspirez-vous de la section 3.

```bash
php artisan make:model Showtime
```

Test	Ce que vous écrivez
une séance connaît son film : $showtime->movie->title	`movie()`, qui renvoie `$this->belongsTo(Movie::class)`
un film connaît ses séances : $movie->showtimes en compte 2 après deux créations	`showtimes()`, qui renvoie `$this->hasMany(Showtime::class)`

11.3 Les fabriques (15 minutes)

Créez `MovieFactory` et `ShowtimeFactory`, puis ajoutez le trait `HasFactory` aux deux modèles. Un film de fabrique a un titre court, dure entre 70 et 180 minutes, et est sorti dans les vingt dernières années. Une séance de fabrique crée son film et coûte entre 6 et 12 euros. Elle a lieu dans les deux prochaines semaines, avec `dateTimeBetween('now', '+2 weeks')`. Inspirez-vous de la section 4.

```bash
php artisan make:factory MovieFactory --model=Movie
php artisan make:factory ShowtimeFactory --model=Showtime
```

Test	Ce que vous écrivez
Movie::factory()->count(3)->create() crée trois films qui durent entre 70 et 180 minutes	`fake()->sentence(3)` et `fake()->numberBetween(70, 180)`
les films de fabrique sont sortis dans les vingt dernières années	`fake()->dateTimeBetween('-20 years', 'now')->format('Y-m-d')`
Showtime::factory() crée une séance pour le film donné, à un prix entre 6 et 12	`'movie_id' => Movie::factory()` et `fake()->numberBetween(6, 12)`

11.4 Le seeder avec fabriques (10 minutes)

Dans `DatabaseSeeder`, importez `Movie` et `Showtime`. Après l'appel de `MovieSeeder`, créez dix films de fabrique, puis trois séances pour chacun. `Movie::factory()->count(10)->create()` renvoie une collection : parcourez-la avec `foreach`, comme la boucle de `ConcertSeeder`. Inspirez-vous de la section 5, puis lancez `php artisan migrate:fresh --seed`.

Test	Ce que vous écrivez
DatabaseSeeder crée 15 films et 30 séances	la boucle, avec `Showtime::factory()->count(3)->create([...])`
après DatabaseSeeder, chaque film de fabrique a trois séances	`'movie_id' => $movie->id` dans le tableau de `create()`

11.5 Le programme des séances (20 minutes)

Créez `ShowtimeController`, avec une méthode `index()`, et la vue `showtimes.index`. Reliez-les à `GET /seances`, route nommée `showtimes.index`. La vue affiche toutes les séances, triées par heure de début, avec pour chacune le titre du film et le prix. Inspirez-vous de la section 7.

```bash
php artisan make:controller ShowtimeController
php artisan make:view showtimes.index
```

Test	Ce que vous écrivez
la page /seances affiche les séances triées par heure de début, avec le titre du film	`Showtime::orderBy('starts_at')->get()` et `$showtime->movie->title`
la page /seances affiche le prix de chaque séance	`$showtime->price`
la route /seances s'appelle showtimes.index	`->name('showtimes.index')`

11.6 La fiche complétée (15 minutes)

La fiche `/films/{id}` du chapitre 10 liste maintenant les séances du film, triées par heure, et affiche « Aucune séance » quand il n'y en a pas. Affichez l'heure avec ses minutes : `format('G \h i')` écrit `14 h 15`, car `i` donne les minutes sur deux chiffres. Inspirez-vous de `StageController`, à la section 7.

Test	Ce que vous écrivez
la fiche d'un film liste ses séances	`$movie->showtimes()->orderBy('starts_at')->get()`, passé à la vue
la fiche d'un film sans séance affiche « Aucune séance »	`@forelse` et `@empty`

11.7 Prédire cinq lectures (sur papier, 10 minutes)

La base contient ce que le seeder de la section 5 y a mis : deux scènes et vingt-trois concerts. Écrivez ce que renvoie chaque ligne : un nombre, un texte, `null`, un objet, une relation ou une requête SQL.

```php
Concert::find(2)->stage->name;
Concert::find(3)->stage_id;
Concert::find(1)->stage();
Stage::find(1)->concerts->count();
Concert::find(3)->stage()->toRawSql();
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/modeles-deux-tables

11.8 Six cartes

Que contient la colonne `stage_id` d'un concert ?

L'`id` de la ligne de `stages` qui représente sa scène. C'est une clé étrangère.

Que fait `constrained()` ?

Il demande à la base de refuser un `stage_id` qui ne correspond à aucune ligne de la table `stages`.

Que fait `cascadeOnDelete()` ?

Quand une scène est supprimée, la base supprime aussi tous les concerts dont le `stage_id` la désigne.

`$stage->concerts` ou `$stage->concerts()` ?

Sans parenthèses, vous recevez une collection de concerts. Avec parenthèses, vous recevez la relation, à compléter avec `orderBy()`, `count()` ou `get()`.

Comment `belongsTo(Stage::class)` trouve-t-il la colonne `stage_id` ?

Il prend le nom de la méthode, `stage`, et ajoute `_id`. Une méthode `scene()` chercherait `scene_id`.

Que fait `'stage_id' => Stage::factory()` dans une fabrique ?

Il crée une nouvelle scène pour chaque concert fabriqué, sauf si le tableau passé à `create()` donne déjà un `stage_id`.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/modeles-deux-tables

3.12 Modèles : plusieurs à plusieurs

Bloc Laravel · séance 5 · ~50 min

Slides du chapitre (PDF) PDF

Au chapitre Views, la fiche d'un concert affichait ses invités. Cette liste a disparu au chapitre 10, quand les concerts sont passés en base. Dans ce chapitre, les invités reviennent avec leur propre table, `artists`, et une troisième table qui relie les artistes aux concerts. Vous comptez ensuite les requêtes qu'une boucle envoie à la base, puis vous chargez les relations avec `with()` pour éviter une requête de plus par concert.

À la fin de ce chapitre, vous serez capables de :

1. **expliquer** pourquoi une relation plusieurs à plusieurs demande une table pivot, et **écrire** sa migration,

2. **déclarer** une relation `belongsToMany` des deux côtés, et la **dessiner** dans un diagramme de classes,
3. **relier** et **délier** des lignes avec `attach`, `detach` et `sync`,
4. **compter** les requêtes d'une boucle, et les **réduire** avec `with()`,
5. **prévoir** ce que la base supprime quand une ligne liée disparaît.

1. Deux listes qui se croisent

Voici les invités écrits dans le tableau `$concerts` du chapitre Views :

```php
1 => [
    'artist' => 'Duo Mistral',
    …
    'guests' => [],
],
2 => [
    'artist' => "L'Orchestre du Lac",
    …
    'guests' => ['Ana Lopes', 'Karim Benali'],
],
3 => [
    'artist' => 'Orage Mécanique',
    …
    'guests' => ['Les Tambours du Parc'],
],
```

Le concert 1, celui de Duo Mistral, n'avait pas d'invité. Ana Lopes y est maintenant invitée aussi : elle joue à deux concerts, et L'Orchestre du Lac reçoit deux invités.

Première idée : une colonne de texte `guests` dans la table `concerts`, qui contient les noms séparés par des virgules.

```
concerts
id | artist             | guests
1  | Duo Mistral        |
2  | L'Orchestre du Lac | Ana Lopes, Karim Benali
3  | Orage Mécanique    | Les Tambours du Parc
```

Seconde idée : une table `artists`, avec une clé étrangère `concert_id`, comme la colonne `stage_id` du chapitre 11.

```
artists
id | name                | concert_id
1  | Ana Lopes           | 2
2  | Karim Benali        | 2
3  | Les Tambours du Parc | 3
```

Le code qui sent mauvais

En binôme, cinq minutes. Dans chacune des deux tables, comment enregistrez-vous qu'Ana Lopes joue aussi au concert de Duo Mistral ?

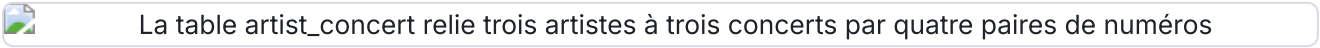
Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/modeles-plusieurs-a-plusieurs

Au chapitre 11, entre une scène et ses concerts, la multiplicité vaut `1` d'un côté et `0..*` de l'autre : la clé étrangère se range du côté `0..*`, dans `concerts`. Entre un concert et ses invités, elle vaut `0..*` des deux côtés : c'est la relation plusieurs à plusieurs du chapitre 4 de la POO, comme entre `Hero` et `Quest`. Une colonne `artist_id` dans `concerts` ne garderait qu'un invité par concert, et une colonne `concert_id` dans `artists` qu'un concert par artiste.

La solution est une troisième table, qui ne contient que des paires de numéros :

```
artists                    artist_concert          concerts
id | name                  artist_id | concert_id   id | artist
1  | Ana Lopes             1         | 1            1  | Duo Mistral
2  | Karim Benali          1         | 2            2  | L'Orchestre du Lac
3  | Les Tambours du Parc  2         | 2            3  | Orage Mécanique
                           3         | 3
```

Chaque ligne de la table du milieu se lit « cet artiste est invité à ce concert ». Le numéro 1 d'Ana Lopes y apparaît deux fois, mais son nom n'est écrit qu'une fois, dans `artists`. Cette table est la **table pivot** annoncée au chapitre 4 de la POO : elle relie deux tables par des paires de clés étrangères.


La table artist_concert relie trois artistes à trois concerts par quatre paires de numéros

2. La table des artistes et la table pivot

Créez d'abord la table des artistes :

```bash
php artisan make:migration create_artists_table
```

```
   INFO  Migration [database/migrations/2026_09_28_222952_create_artists_table.php] created successfully.
```

```php
Schema::create('artists', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('instrument');
    $table->timestamps();
});
```

`instrument` contient l'instrument principal de l'artiste. Créez ensuite la table pivot :

```bash
php artisan make:migration create_artist_concert_table
```

```
   INFO  Migration [database/migrations/2026_09_28_222954_create_artist_concert_table.php]
created successfully.
```

Dans le fichier créé, remplacez les lignes `id()` et `timestamps()` :

```php
Schema::create('artist_concert', function (Blueprint $table) {
    $table->foreignId('artist_id')->constrained()->cascadeOnDelete();
    $table->foreignId('concert_id')->constrained()->cascadeOnDelete();
    $table->primary(['artist_id', 'concert_id']);
});
```

- **Les deux `foreignId()`** fonctionnent comme `stage_id` au chapitre 11 : la base refuse un numéro qui ne désigne aucune ligne, et supprime la paire quand l'artiste ou le concert disparaît.
- **`primary(['artist_id', 'concert_id'])`** fait de la paire la clé primaire de la table. Une clé primaire faite de plusieurs colonnes s'appelle une **clé primaire composée**. La base refuse deux lignes avec la même paire : Ana Lopes ne peut pas être invitée deux fois au même concert.
- **Pas d'`id()`**, car une ligne se reconnaît à sa paire. **Pas de `timestamps()`**, car aucune page n'affiche la date d'une invitation.

Le nom `artist_concert` suit la convention de Laravel pour une table pivot : les noms des deux modèles, au singulier et en minuscules, dans l'ordre alphabétique, séparés par `_`. Le `a` d'`artist` vient avant le `c` de `concert` : la table s'appelle donc `artist_concert`, et non `concert_artist`.

```bash
php artisan migrate
php artisan migrate:status
```

```
   INFO  Running migrations.

  2026_09_28_222952_create_artists_table ......................... 4.91ms DONE
  2026_09_28_222954_create_artist_concert_table .................. 4.23ms DONE

  Migration name ............................................ Batch / Status
…
  2026_09_28_220038_add_stage_id_to_concerts_table ................... [1] Ran
  2026_09_28_222952_create_artists_table ............................. [2] Ran
  2026_09_28_222954_create_artist_concert_table ...................... [2] Ran
```

BASH

```
php artisan db:table artist_concert
```

```
  main.artist_concert ........................................................
  Columns ............................................................... 2

  Column ................................................................ Type
  artist_id integer ................................................ integer
  concert_id integer ............................................... integer

  Index ..................................................................
  sqlite_autoindex_artist_concert_1 artist_id, concert_id .. compound, primary

  Foreign Key ........................................... On Update / On Delete
  concert_id references id on concerts .................... no action / cascade
  artist_id references id on artists ...................... no action / cascade
```

La ligne `Index` montre la clé primaire, `primary`, faite des deux colonnes, `compound`. Les deux lignes `Foreign Key` montrent les suppressions en cascade.

3. Les deux modèles et `belongsToMany`

BASH

```
php artisan make:model Artist
```

```
   INFO  Model [app/Models/Artist.php] created successfully.
```

Donnez à `Artist` sa liste `#[Fillable]` et une relation vers les concerts :

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Attributes\Fillable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

#[Fillable(['name', 'instrument'])]
class Artist extends Model
{
    public function concerts(): BelongsToMany
    {
        return $this->belongsToMany(Concert::class);
    }
}
```

Dans `Concert`, importez `BelongsToMany` sous `BelongsTo`, et ajoutez la méthode `guests()` sous `stage()` :

```php
public function guests(): BelongsToMany
{
    return $this->belongsToMany(Artist::class);
}
```

`belongsToMany(Concert::class)` se lit « un artiste est lié à plusieurs concerts ». Pour trouver ces concerts, Eloquent a besoin de trois noms, qu'il déduit des deux classes :

- **la table pivot**, `artist_concert`, faite des noms `Artist` et `Concert` rangés dans l'ordre alphabétique,
- **la colonne de la classe qui déclare la relation**, `artist_id`, le nom `Artist` en minuscules suivi de `_id`,
- **la colonne de la classe visée**, `concert_id`, de la même façon.

Le nom de la méthode ne sert à aucune de ces déductions. Dans `Concert`, elle s'appelle `guests()`, et non `artists()`, parce qu'elle renvoie les invités du concert. Au chapitre 11, `belongsTo()` lisait au contraire le nom de la méthode `stage()` pour trouver `stage_id`.

Ce que la relation demande à la base

Recréez la route bac à sable à la fin de `routes/web.php`, et importez `Artist` et `Concert` au-dessus de `use Illuminate\Support\Facades\Route;` :

```php
use App\Models\Artist;
use App\Models\Concert;
```

```php
Route::get('/bac-a-sable', function () {
    concert=Concert::find(2);dump(concert = Concert::find(2);

    dump(concert=Concert::find(2);dump(concert->guests);
    dd($concert->guests()->toRawSql());
});
```

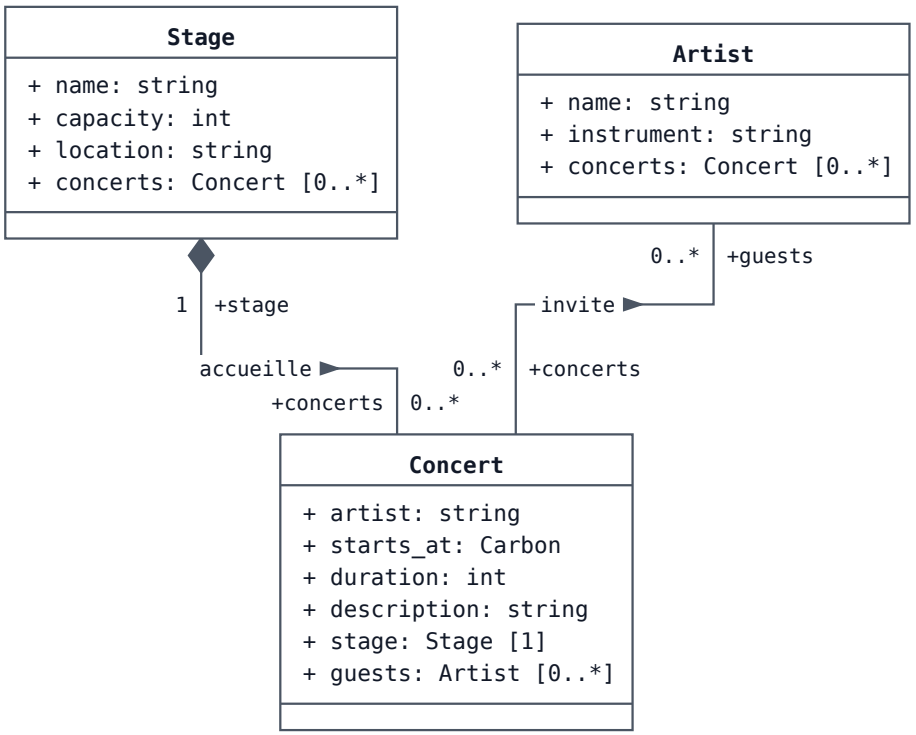
```
Illuminate\Database\Eloquent\Collection {#1102 ▼ // routes/web.php:188
  #items: []
  #escapeWhenCastingToString: false
}

"select * from "artists" inner join "artist_concert" on "artists"."id" =
"artist_concert"."artist_id" where "artist_concert"."concert_id" = 2" // routes/web.php:189
```

La collection est vide, car la table pivot ne contient encore aucune paire. La requête se lit en trois morceaux :

- `where "artist_concert"."concert_id" = 2` garde les paires du concert 2,
- `inner join "artist_concert" on "artists"."id" = "artist_concert"."artist_id"` associe à chaque paire la ligne d'`artists` dont l'`id` vaut `artist_id`,
- `select * from "artists"` renvoie ces lignes d'artistes.

Une requête qui assemble ainsi les lignes de deux tables à partir d'une valeur commune s'appelle une **jointure**, `join` en SQL. En une seule requête, elle renvoie les artistes dont l'`id` figure dans les paires du concert 2.

Le diagramme de classes



Le trait entre `Concert` et `Artist` porte `0..*` à chaque bout : un concert invite zéro, un ou plusieurs artistes, et un artiste est invité à zéro, un ou plusieurs concerts. Chaque rôle est le nom que le code utilise, `guests` dans `Concert` et `concerts` dans `Artist`, et revient dans la boîte de la classe qui le porte.

Le losange plein ◆——— entre `Stage` et `Concert` est celui du chapitre 11 : un concert disparaît avec sa scène. Entre `Concert` et `Artist`, le trait reste une association simple ———, car un artiste existe sans le concert. La table `artist_concert` n'apparaît pas : c'est une table de la base, pas une classe, et le trait la représente.

Dans `Concert`, `artist` reste un texte : la tête d'affiche garde sa colonne dans `concerts`, que toutes les pages lisent déjà, et seuls les invités passent dans `artists`.

4. Relier : `attach`, `detach` et `sync`

Enregistrez Ana Lopes et Karim Benali, puis invitez-les :

```php
Route::get('/bac-a-sable', function () {
    ana=Artist::create(['name'⇒'AnaLopes','instrument'⇒'chant']);ana =
Artist::create(['name' ⇒ 'Ana Lopes', 'instrument' ⇒ 'chant']);
    ana=Artist::create(['name'⇒'AnaLopes','instrument'⇒'chant']);karim =
Artist::create(['name' ⇒ 'Karim Benali', 'instrument' ⇒ 'oud']);

    Concert::find(1)→guests()→attach(ana->id);Concert::find(2)->guests()->attach([ana
→id);
    Concert::find(2)→guests()→attach([ana->id);Concert::find(2)->guests()->attach([ana
→id, $karim→id]);

    return 'Invités enregistrés';
});
```

```
Invités enregistrés
```

La méthode `guests()`, avec parenthèses, renvoie la relation, comme `stage()` au chapitre 11. Sa méthode **`attach()`** insère une paire dans `artist_concert` pour chaque numéro d'artiste reçu, seul ou dans un tableau.

Chaque visite de cette page crée deux artistes de plus : ouvrez-la une seule fois. Dans SQLite Viewer, la table `artist_concert` contient alors trois lignes :

```
artist_id | concert_id
1         | 1
1         | 2
2         | 2
```

Lisez maintenant la relation dans les deux sens :

```php
Route::get('/bac-a-sable', function () {
    foreach (Concert::find(2)→guests as guest) {
        dump($guest→name);
    }

    ana = Artist::where('name', 'Ana Lopes')→first();

    foreach (ana->concertsasana→concerts as ana->concertsasconcert) {
        dump($concert→artist);
    }
});
```

```
"Ana Lopes" // routes/web.php:187

"Karim Benali" // routes/web.php:187

"Duo Mistral" // routes/web.php:193

"L'Orchestre du Lac" // routes/web.php:193
```

`$concert→guests` cherche dans `artist_concert` les paires du concert, et `$ana→concerts` celles de l'artiste.

La même paire deux fois

Invitez une seconde fois Ana Lopes au concert 2 :

```php
Route::get('/bac-a-sable', function () {
    ana=Artist::where('name','AnaLopes')->first();Concert::find(2)->guests()->attach(ana =
Artist::where('name', 'Ana Lopes')→first();

    Concert::find(2)→guests()→attach(ana=Artist::where('name','AnaLopes')
->first();Concert::find(2)->guests()->attach(ana→id);

    return 'Ana Lopes invitée';
});
```

```
Illuminate\Database\UniqueConstraintViolationException
SQLSTATE[23000]: Integrity constraint violation: 19 UNIQUE constraint failed:
artist_concert.artist_id, artist_concert.concert_id (Connection: sqlite, …, SQL: insert into
"artist_concert" ("artist_id", "concert_id") values (1, 2))
```

`attach()` envoie directement l'insertion de la paire `(1, 2)`, sans requête pour vérifier qu'elle existe déjà. La base la refuse : `UNIQUE constraint failed` cite les deux colonnes de la clé primaire composée. Sans cette clé, la fiche du concert afficherait Ana Lopes deux fois.

Délier

`detach()` retire des paires du concert sur lequel vous l'appelez. `detach($karim→id)` retire la paire de Karim Benali, et `detach()` sans argument retire toutes les paires du concert.

```php
Route::get('/bac-a-sable', function () {
    concert=Concert::find(2);concert = Concert::find(2);
    concert=Concert::find(2);karim = Artist::where('name', 'Karim Benali')->first();

    concert->guests()->detach(concert->guests()->detach(concert->guests()->detach(karim
->id);
    dump(concert->guests()->count());concert->guests()->count());

    concert->guests()->count());concert->guests()->detach();
    dd($concert->guests()->count());
});
```

```
1 // routes/web.php:190

0 // routes/web.php:193
```

Le concert 2 n'a plus d'invité, et Karim Benali existe toujours dans `artists`. La table pivot ne contient plus que la paire `(1, 1)`, qui relie Ana Lopes au concert de Duo Mistral.

Remplacer toute la liste

`sync()` reçoit la liste complète des invités voulus pour le concert. Il compare cette liste avec les paires du concert, ajoute celles qui manquent et retire celles qui n'y figurent pas.

```php
Route::get('/bac-a-sable', function () {
    concert=Concert::find(1);concert = Concert::find(1);
    concert=Concert::find(1);karim = Artist::where('name', 'Karim Benali')->first();

    foreach (concert->guests()->get()asconcert->guests()->get() as concert->guests()
->get()asguest) {
        dump('avant : ' . guest->name);
    }

    concert->guests()->sync([karim->id]);foreach(karim->id]);

    foreach (karim->id]);foreach(concert->guests()->get() as $guest) {
        dump('après : ' . $guest->name);
    }
});
```

```
"avant : Ana Lopes" // routes/web.php:190

"après : Karim Benali" // routes/web.php:196
```

Les boucles lisent `guests()->get()`, qui envoie une nouvelle requête à chaque appel. La propriété `guests` resterait en mémoire après la première boucle, comme au chapitre 11, et la seconde afficherait encore Ana Lopes. `sync()` renvoie aussi un tableau qui détaille ses changements :

```php
Route::get('/bac-a-sable', function () {
    concert=Concert::find(1);concert = Concert::find(1);
    concert=Concert::find(1);ana = Artist::where('name', 'Ana Lopes')->first();
    karim=Artist::where('name','KarimBenali')->first();dd(karim = Artist::where('name',
'Karim Benali')->first();

    dd(karim=Artist::where('name','KarimBenali')->first();dd(concert->guests()
->sync([ana->id,ana->id, ana->id,karim->id]));
});
```

```
array:3 [▼ // routes/web.php:190
  "attached" => array:1 [▼
    0 => 1
  ]
  "detached" => []
  "updated" => []
]
```

Karim Benali était déjà invité : `sync()` n'a ajouté qu'Ana Lopes, l'artiste 1, sans erreur. La clé `updated` concerne des colonnes supplémentaires d'une table pivot, que `artist_concert` n'a pas. Rechargez la page : les tableaux `attached`, `detached` et `updated` sont vides, car les deux paires existent déjà.

Écrivez `attach()` pour ajouter un invité, et `sync()` quand vous connaissez la liste complète des invités.

Le seeder et la fabrique

Créez une fabrique et un seeder pour les artistes :

```bash
php artisan make:factory ArtistFactory --model=Artist
php artisan make:seeder ArtistSeeder
```

```
   INFO  Factory [database/factories/ArtistFactory.php] created successfully.

   INFO  Seeder [database/seeders/ArtistSeeder.php] created successfully.
```

Comme au chapitre 11, `make:model` n'a pas ajouté le trait `HasFactory` : collez-le dans `Artist`, avec son import, comme dans `Stage`. Remplissez ensuite la fabrique :

```php
public function definition(): array
{
    return [
        'name' => fake()->name(),
        'instrument' => fake()->randomElement(['guitare', 'violon', 'piano', 'batterie',
'saxophone']),
    ];
}
```

Dans `ArtistSeeder`, importez `Artist` et `Concert`. Écrivez les trois invités du chapitre Views, ajoutez Ana Lopes au concert 1, puis créez dix artistes de fabrique :

```php
public function run(): void
{
    ana=Artist::create(['name'=>'AnaLopes','instrument'=>'chant']);ana =
Artist::create(['name' => 'Ana Lopes', 'instrument' => 'chant']);
    ana=Artist::create(['name'=>'AnaLopes','instrument'=>'chant']);karim =
Artist::create(['name' => 'Karim Benali', 'instrument' => 'oud']);
    drums=Artist::create(['name'=>'LesTamboursduParc',
'instrument'=>'percussions']);Concert::where('artist','DuoMistral')->first()->guests()
->attach(drums = Artist::create(['name' => 'Les Tambours du Parc', 'instrument' =>
'percussions']);

    Concert::where('artist', 'Duo Mistral')->first()->guests()
->attach(drums=Artist::create(['name'=>'LesTamboursduParc',
'instrument'=>'percussions']);Concert::where('artist','DuoMistral')->first()->guests()
->attach(ana->id);
    Concert::where('artist', "L'Orchestre du Lac")->first()->guests()->attach([ana->id,ana
->id, ana->id,karim->id]);
    Concert::where('artist', 'Orage Mécanique')->first()->guests()->attach(drums->id);drums
->id);

    drums->id);artists = Artist::factory()->count(10)->create();

    foreach (Concert::where('id', '>', 3)->get() as $concert) {
        $concert->guests()->attach($artists->random(2));
    }
}
```

- `where('id', '>', 3)` place un opérateur de comparaison entre la colonne et la valeur. La boucle parcourt donc les concerts 4 à 23, ceux de la fabrique.
- `$artists->random(2)` prend au hasard deux artistes différents dans la collection. Une même paire n'est donc jamais insérée deux fois. `attach()` accepte une collection d'objets, dont il lit les `id`.

Dans `DatabaseSeeder`, appelez `ArtistSeeder` après `ConcertSeeder`, car il cherche des concerts qui doivent déjà exister :

```php
this->call(StageSeeder::class);this→call(StageSeeder::class);
this->call(StageSeeder::class);this→call(ConcertSeeder::class);
$this→call(ArtistSeeder::class);
```

```bash
php artisan migrate:fresh --seed
```

```
…
   INFO  Seeding database.

  Database\Seeders\StageSeeder ..................................... RUNNING
  Database\Seeders\StageSeeder ..................................... 8 ms DONE

  Database\Seeders\ConcertSeeder ................................... RUNNING
  Database\Seeders\ConcertSeeder ................................... 90 ms DONE

  Database\Seeders\ArtistSeeder .................................... RUNNING
  Database\Seeders\ArtistSeeder .................................... 134 ms DONE
```

La base contient treize artistes et quarante-quatre paires : les quatre paires écrites à la main, puis deux invités pour chacun des vingt concerts de fabrique. Les noms inventés par Faker changent à chaque lancement.

5. La fiche du concert retrouve ses invités

Dans `concerts/show.blade.php`, ajoutez sous les deux descriptions la liste du chapitre Views, qui lit maintenant la relation :

```xml
<h2>Invités</h2>
<ul>
    @forelse (concert->guestsasconcert→guests as concert->guestsasguest)
        <li>{{ guest→name }}, {{ guest→instrument }}</li>
    @empty
        <li>Aucun invité annoncé</li>
    @endforelse
</ul>
```

```bash
curl -s http://premier-test-laravel.test/concerts/1
curl -s http://premier-test-laravel.test/concerts/2
```

```
…
    <h2>Invités</h2>
    <ul>
                    <li>Ana Lopes, chant</li>
            </ul>
…
    <h2>Invités</h2>
    <ul>
                    <li>Ana Lopes, chant</li>
                    <li>Karim Benali, oud</li>
            </ul>
…
```

Pour voir le bloc `@empty`, retirez les invités du concert 3 dans la route bac à sable, puis envoyez le navigateur vers sa fiche :

PHP

```php
Route::get('/bac-a-sable', function () {
    Concert::find(3)->guests()->detach();

    return redirect()->route('concerts.show', 3);
});
```

```
    <h2>Invités</h2>
    <ul>
                    <li>Aucun invité annoncé</li>
            </ul>
```

Une page par artiste

BASH

```bash
php artisan make:controller ArtistController
php artisan make:view artists.show
```

```
   INFO  Controller [app/Http/Controllers/ArtistController.php] created successfully.

   INFO  View [resources/views/artists/show.blade.php] created successfully.
```

Le contrôleur suit le modèle de `StageController` :

```php
<?php

namespace App\Http\Controllers;

use App\Models\Artist;
use Illuminate\View\View;

class ArtistController extends Controller
{
    public function show(int $id): View
    {
        $artist = Artist::findOrFail($id);
        $concerts = $artist->concerts()->orderBy('starts_at')->get();

        return view('artists.show', ['artist' => $artist, 'concerts' => $concerts]);
    }
}
```

```xml
<x-layouts.app :title="artist->name">
    <h1>{{ $artist->name }}</h1>
    <p>{{ $artist->instrument }}</p>

    <h2>Concerts</h2>
    <ul>
        @foreach (concerts as $concert)
            <li>
                {{ $concert->starts_at->format('G \h') }}
                <a href="{{ route('concerts.show', $concert->id) }}">{{ $concert->artist }}
</a>
            </li>
        @endforeach
    </ul>
</x-layouts.app>
```

Dans `routes/web.php`, importez `ArtistController` sous `StageController`, puis ajoutez la route au-dessus de la route bac à sable :

```php
Route::get('/artistes/{id}', [ArtistController::class, 'show'])->whereNumber('id')
->name('artists.show');
```

```bash
curl -s http://premier-test-laravel.test/artistes/1
```

```
…
    <title>Ana Lopes · Les Nuits du Parc</title>
…
        <h1>Ana Lopes</h1>
    <p>chant</p>

    <h2>Concerts</h2>
    <ul>
                    <li>
                18 h
                <a href="http://premier-test-laravel.test/concerts/1">Duo Mistral</a>
            </li>
                    <li>
                20 h
                <a href="http://premier-test-laravel.test/concerts/2">L&#039;Orchestre du
Lac</a>
            </li>
            </ul>
…
```

`/artistes/99` répond 404. Sur la fiche du concert, faites de chaque nom d'invité un lien vers sa page :

XML

```
<li><a href="{{ route('artists.show', guest->id) }}">{{ guest->name }}</a>, {{ $guest
->instrument }}</li>
```

```
    <h2>Invités</h2>
    <ul>
                    <li><a href="http://premier-test-laravel.test/artistes/1">Ana Lopes</a>,
chant</li>
                    <li><a href="http://premier-test-laravel.test/artistes/2">Karim
Benali</a>, oud</li>
            </ul>
```

Avant de mesurer, remettez la base à neuf. La commande rend son invité au concert 3, et sa sortie est celle de la section 4 :

BASH

```
php artisan migrate:fresh --seed
```

6. Compter les requêtes : le piège N+1

`DB::enableQueryLog()` demande à Laravel de garder en mémoire chaque requête envoyée ensuite à la base. `DB::getQueryLog()` renvoie le tableau de ces requêtes. Importez la classe `DB` au-dessus de `Route` :

PHP

```
use Illuminate\Support\Facades\DB;
```

Comptez les requêtes d'une boucle qui lit la scène de chaque concert, comme le fait le composant `concert-card` :

```php
Route::get('/bac-a-sable', function () {
    DB::enableQueryLog();

    foreach (Concert::all() as $concert) {
        $stageName = $concert->stage->name;
    }

    dd(count(DB::getQueryLog()));
});
```

```
24 // routes/web.php:196
```

Remplacez `count(DB::getQueryLog())` par `DB::getQueryLog()`, puis dépliez les trois premières entrées (sortie raccourcie) :

```
array:24 [▼ // routes/web.php:196
  0 => array:4 [▼
    "query" => "select * from "concerts""
    "bindings" => []
…
  1 => array:4 [▼
    "query" => "select * from "stages" where "stages"."id" = ? limit 1"
    "bindings" => array:1 [▼
      0 => 1
    ]
…
  2 => array:4 [▼
    "query" => "select * from "stages" where "stages"."id" = ? limit 1"
    "bindings" => array:1 [▼
      0 => 2
    ]
…
```

La première requête lit les vingt-trois concerts. Ensuite, chaque lecture de `$concert->stage` envoie sa propre requête, et le `?` y reçoit la valeur de `bindings`, le `stage_id` du concert. Les scènes 1 et 2 sont donc demandées vingt-trois fois en tout.

Une requête pour la liste, puis une requête de plus par élément de la liste : c'est le **piège N+1**. Avec cinq cents concerts, la boucle enverrait cinq cent une requêtes. La page `/concerts` fait la même boucle dans le composant `concert-card`, à chaque visite.

Charger la relation avec `with()`

```php
Route::get('/bac-a-sable', function () {
    DB::enableQueryLog();

    foreach (Concert::with('stage')->get() as $concert) {
        $stageName = $concert->stage->name;
    }

    dd(DB::getQueryLog());
});
```

```
array:2 [▼ // routes/web.php:196
  0 => array:4 [▼
    "query" => "select * from "concerts""
    "bindings" => []
…
  1 => array:4 [▼
    "query" => "select * from "stages" where "stages"."id" in (1, 2)"
    "bindings" => []
…
```

`with('stage')` demande à Eloquent de charger la relation en même temps que la liste. Eloquent envoie la première requête, puis relève les `stage_id` des vingt-trois concerts, 1 et 2. Il envoie ensuite une seule requête avec `in (1, 2)`, qui lit les deux scènes.

Enfin, Eloquent range chaque scène dans la propriété `stage` des concerts qui la désignent. Dans la boucle, `$concert->stage` lit cette propriété gardée en mémoire, sans requête.

À gauche, la boucle sans with() envoie 24 requêtes. À droite, with('stage') en envoie 2

`with()` accepte aussi un tableau de relations :

```php
Route::get('/bac-a-sable', function () {
    DB::enableQueryLog();

    foreach (Concert::with(['stage', 'guests'])->get() as $concert) {
        $stageName = $concert->stage->name;

        foreach ($concert->guests as $guest) {
            $guestName = $guest->name;
        }
    }

    dump(count(DB::getQueryLog()));
    dd(DB::getQueryLog()[2]['query']);
});
```

```
3 // routes/web.php:200

"select "artists".*, "artist_concert"."concert_id" as "pivot_concert_id",
"artist_concert"."artist_id" as "pivot_artist_id" from "artists" inner join "artist_concert"
on "artists"."id" = "artist_concert"."artist_id" where "artist_concert"."concert_id" in (1,
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23)" //
routes/web.php:201
```

Trois requêtes, là où la même boucle sans `with()` en envoie quarante-sept : une pour les concerts, vingt-trois pour les scènes et vingt-trois pour les invités. La troisième requête est la jointure de la section 3, avec `in (1, …, 23)` à la place de `= 2`. Les deux colonnes `pivot_…` indiquent à Eloquent à quel concert revient chaque artiste.

Appliquez la règle dans `ConcertController`. La méthode `index()` charge les scènes avec les concerts :

```php
$concerts = Concert::with('stage')->orderBy('starts_at')->get();
```

La page `/concerts` affiche le même HTML. Sa boucle, dans `concert-card`, n'envoie plus que les deux requêtes de la route bac à sable, au lieu de vingt-quatre. La méthode `show()` ne change pas : la fiche `/concerts/2` envoie trois requêtes, pour le concert, sa scène et ses invités, avec ou sans `with()`.

Quand une boucle affiche une relation, chargez cette relation avec `with()`.

7. Modifier et supprimer avec des relations

Importez `Stage` sous `Concert` dans `routes/web.php`. Ana Lopes se produit désormais sous le nom « Ana Lopes Ferreira ». Son nom n'est écrit qu'une fois, dans `artists` :

```php
Route::get('/bac-a-sable', function () {
    Artist::find(1)->update(['name' => 'Ana Lopes Ferreira']);

    return redirect()->route('concerts.show', 2);
});
```

```
    <h2>Invités</h2>
    <ul>
                    <li><a href="http://premier-test-laravel.test/artistes/1">Ana Lopes
Ferreira</a>, chant</li>
                    <li><a href="http://premier-test-laravel.test/artistes/2">Karim
Benali</a>, oud</li>
            </ul>
```

Une seule ligne a changé, et la fiche du concert 1 affiche aussi le nouveau nom. Vérifiez maintenant les cascades. La table pivot n'a pas de modèle : `DB::table('artist_concert')` l'interroge directement, et `count()` compte ses lignes.

```php
Route::get('/bac-a-sable', function () {
    dump(DB::table('artist_concert')->count());

    Artist::find(1)->delete();

    dump(DB::table('artist_concert')->count());
    dd(Concert::count());
});
```

```
44 // routes/web.php:191

42 // routes/web.php:195

23 // routes/web.php:196
```

Ana Lopes était invitée à deux concerts. La base a supprimé ses deux paires, et les vingt-trois concerts restent. Supprimez maintenant un concert :

```php
Route::get('/bac-a-sable', function () {
    dump(DB::table('artist_concert')->count());

    Concert::find(2)->delete();

    dump(DB::table('artist_concert')->count());
    dd(Artist::count());
});
```

```
42 // routes/web.php:191

41 // routes/web.php:195

12 // routes/web.php:196
```

La paire de Karim Benali au concert 2 a disparu, et les douze artistes restent. Supprimez enfin la Grande scène, comme au chapitre 11 :

```php
Route::get('/bac-a-sable', function () {
    dump(Concert::count());
    dump(DB::table('artist_concert')->count());

    Stage::where('name', 'Grande scène')->first()->delete();

    dump(Concert::count());
    dd(DB::table('artist_concert')->count());
});
```

```
22 // routes/web.php:191

41 // routes/web.php:192

11 // routes/web.php:196

20 // routes/web.php:197
```

La clé étrangère `stage_id` de `concerts` fait supprimer les onze concerts de la scène. La clé `concert_id` d'`artist_concert` fait ensuite supprimer les paires de chacun de ces concerts. Il reste les onze concerts du Chapiteau et leurs vingt paires, et aucun artiste n'est supprimé.

Vous supprimez	La base supprime aussi
un artiste	ses paires dans `artist_concert`
un concert	ses paires dans `artist_concert`
une scène	ses concerts, puis leurs paires

Remettez le festival en état :

```bash
php artisan migrate:fresh --seed
```

La sortie est celle de la section 4. Supprimez enfin la route `/bac-a-sable` et les imports d'`Artist`, `Concert`, `Stage` et `DB` dans `routes/web.php`, comme aux chapitres 10 et 11.

À retenir

- Une relation plusieurs à plusieurs se range dans une table pivot, dont la clé primaire composée interdit d'enregistrer deux fois la même paire.
- `belongsToMany()` déduit tout des noms des deux classes : la table pivot les réunit dans l'ordre alphabétique, et chaque colonne ajoute `_id` à l'un d'eux.
- `attach()` ajoute des paires, `detach()` en retire, et `sync()` rend les paires du concert identiques à la liste reçue.
- Une boucle qui lit une relation envoie une requête par élément, et `with()` remplace ces requêtes par une seule.
- Supprimer un artiste ou un concert supprime ses paires dans la table pivot, jamais l'artiste ou le concert d'en face.

Exercices

Comptez 85 minutes pour les exercices, en séance ou à la maison, et 20 minutes de plus pour le bonus.

Les katas 12.1 à 12.5 se font dans votre dépôt `eloquent-26`, à la suite des katas du chapitre 11. Les films y reçoivent des genres : une table `genres` et une table pivot `genre_movie`. Lancez les tests du chapitre :

```bash
php artisan test --group=chapitre12
```

Au départ, les 14 tests sont rouges. Les groupes `chapitre10` et `chapitre11` doivent rester verts : après chaque kata, lancez `php artisan test --group=chapitre10`, puis `php artisan test --group=chapitre11`.

12.1 La table pivot (15 minutes)

Créez la migration `create_genres_table`, avec une colonne `name` en plus de `id()` et `timestamps()`. Créez ensuite `create_genre_movie_table`, avec deux clés étrangères qui suppriment leurs lignes en cascade, et la paire comme clé primaire. Appliquez-les. Inspirez-vous de la section 2.

```bash
php artisan make:migration create_genres_table
php artisan make:migration create_genre_movie_table
```

Test	Ce que vous écrivez
les tables genres (name) et genre_movie (genre_id, movie_id) existent	les deux `Schema::create()`
la même paire genre-film ne peut pas être enregistrée deux fois (clé primaire composée)	`$table->primary(['genre_id', 'movie_id'])`
genre_movie refuse un film ou un genre qui n'existe pas (clés étrangères)	`foreignId(…)->constrained()` pour les deux colonnes
supprimer un film retire ses lignes de genre_movie (cascadeOnDelete)	`->cascadeOnDelete()`

12.2 `belongsToMany` (10 minutes)

Créez le modèle `Genre` avec `#[Fillable]`. Ajoutez `movies()` dans `Genre` et `genres()` dans `Movie`. Inspirez-vous de la section 3.

```bash
php artisan make:model Genre
```

Test	Ce que vous écrivez
attach() relie un film à deux genres : $movie->genres en compte 2	`genres()`, qui renvoie `$this->belongsToMany(Genre::class)`
un genre connaît ses films : $genre->movies contient le film	`movies()`, qui renvoie `$this->belongsToMany(Movie::class)`
sync([]) détache tous les genres d'un film	rien de plus

12.3 Le seeder (15 minutes)

Créez `GenreSeeder`, importez-y `Genre`, et écrivez les six genres : Animation, Comédie, Documentaire, Drame, Science-fiction, Thriller.

```bash
php artisan make:seeder GenreSeeder
```

Dans `DatabaseSeeder`, importez `Genre` et appelez `GenreSeeder` après la création des films. Donnez ensuite à chaque film un à trois genres au hasard. La fonction PHP `rand(1, 3)` renvoie un entier entre les deux bornes. Inspirez-vous de `ArtistSeeder`, à la section 4 :

```php
genres=Genre::all();foreach(Movie::all()asgenres = Genre::all();

foreach (Movie::all() as genres=Genre::all();foreach(Movie::all()asmovie) {
    movie->genres()->attach(movie→genres()→attach(movie->genres()->attach(genres
→random(rand(1, 3)));
}
```

Test	Ce que vous écrivez
GenreSeeder crée les six genres de l'énoncé	six `Genre::create([...])`, ou une boucle sur un tableau de noms
après DatabaseSeeder, il y a six genres et chaque film en a entre un et trois	`$this→call(GenreSeeder::class);` et la boucle ci-dessus

12.4 La page genre (20 minutes)

Créez `GenreController` avec une méthode `show()`, et la vue `genres.show`. La page affiche le nom du genre et ses films. Dans `routes/web.php`, reliez `GET /genres/{id}` à `GenreController::show()`, ajoutez `whereNumber('id')` et nommez la route `genres.show`. Inspirez-vous d'`ArtistController`, à la section 5.

```bash
php artisan make:controller GenreController
php artisan make:view genres.show
```

Test	Ce que vous écrivez
la page /genres/{id} affiche le nom du genre et ses films, et seulement eux	`$genre→movies()→orderBy('title')→get()` et un `@foreach`
un numéro de genre qui n'existe pas répond 404	`Genre::findOrFail($id)`
la route /genres/{id} s'appelle genres.show	`→name('genres.show')`

12.5 Les genres dans la liste (10 minutes)

Sur la page `/films`, chaque ligne du tableau affiche les genres du film. Chargez-les dans `MovieController::index()` avec `with()`. Inspirez-vous de la section 6.

Test	Ce que vous écrivez
la page /films affiche les genres de chaque film	un `@foreach ($movie->genres as $genre)` dans la ligne du film
la page /films charge les genres avec with() : au plus 3 requêtes SQL pour toute la page	`Movie::with('genres')->orderBy('title')`

12.6 Bonus : les salles (20 minutes)

Chaque séance a lieu dans une salle. Créez la table `rooms`, avec `name` et `seats`, le nombre de places, et le modèle `Room`.

Ajoutez à `showtimes` une clé étrangère `room_id` avec `nullable()`, car les séances déjà enregistrées, et celles des tests du chapitre 11, n'ont pas de salle. Ajoutez la relation `room()` dans `Showtime`, sur le modèle du chapitre 11.

Sur la page `/seances`, affichez la salle avec `$showtime->room?->name`, l'opérateur `?->` du chapitre 5 de la POO. Dans `ShowtimeController::index()`, chargez le film et la salle avec `with(['movie', 'room'])`. Ces tests forment un groupe à part :

```bash
php artisan make:migration create_rooms_table
php artisan make:migration add_room_id_to_showtimes_table --table=showtimes
php artisan make:model Room
php artisan test --group=bonus
```

Test	Ce que vous écrivez
bonus : la table rooms (name, seats) existe et showtimes a une colonne room_id	`string('name')`, `unsignedSmallInteger('seats')`, `foreignId('room_id')->nullable()->constrained()`
bonus : une séance connaît sa salle : $showtime->room->name	`room()`, qui renvoie `$this->belongsTo(Room::class)`
bonus : la page /seances affiche la salle de chaque séance	`{{ $showtime->room?->name }}` dans la ligne de la séance

12.7 Des tables et des requêtes (sur papier, 15 minutes)

1. L'application d'une bibliothèque enregistre des livres et leurs auteurs. Un livre a un ou plusieurs auteurs, et un auteur a écrit un ou plusieurs livres. Dessinez les trois tables avec leurs colonnes et deux lignes d'exemple, en donnant à la table pivot le nom attendu par Laravel. Dessinez ensuite le diagramme de classes de `Book` et `Author`.

2. La base contient ce que le seeder de la section 4 y a mis : deux scènes, vingt-trois concerts et treize artistes. Combien de requêtes chaque extrait envoie-t-il, et quelles relations `with()` devrait-il charger ?

```php
// Extrait A
stage=Stage::find(1);foreach(stage = Stage::find(1);
foreach (stage=Stage::find(1);foreach(stage→concerts as concert) {
    echo $concert→artist;
}

// Extrait B
foreach (Artist::all() as artist) {
    echo artist→concerts→count();
}

// Extrait C
foreach (Concert::with('guests')→get() as concert) {
    echo concert->stage->name.':'.concert→stage→name . ' : ' . concert->stage->name.':
'.concert→guests→count();
}
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/modeles-plusieurs-a-plusieurs

12.8 Six cartes

Pourquoi une table pivot ?

Parce qu'un concert a plusieurs invités et qu'un invité vient à plusieurs concerts. Une case de `concerts` ou d' `artists` ne peut contenir qu'une clé étrangère, alors que chaque ligne de la table pivot en contient une paire.

Comment Laravel nomme-t-il la table pivot de `Artist` et `Concert` ?

`artist_concert` : les deux noms de modèles au singulier, en minuscules, dans l'ordre alphabétique, reliés par `_` .

Que refuse la clé primaire composée de la table pivot ?

Une seconde ligne avec la même paire `(artist_id, concert_id)` . La base répond `UNIQUE constraint failed` .

`attach()` ou `sync()` ?

`attach()` ajoute des paires, et échoue si l'une existe déjà. `sync()` reçoit la liste complète, ajoute ce qui manque et retire le reste.

Qu'est-ce que le piège N+1 ?

Une requête pour lire une liste, puis une requête de plus par élément quand la boucle lit une relation. Avec 23 concerts, 24 requêtes.

Que fait `Concert::with('stage')->get()` ?

Il lit les concerts, puis toutes leurs scènes en une seule requête avec `where "id" in (…)` . Dans la boucle, `$concert->stage` n'envoie plus de requête.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/modeles-plusieurs-a-plusieurs

3.13 Tailwind CSS

Bloc Laravel · séance 5 · ~60 min

Slides du chapitre (PDF) PDF

Depuis le chapitre 9, vos pages s'affichent avec le style par défaut du navigateur : texte noir sur fond blanc, titres en gras, liens bleus soulignés. Dans ce chapitre, vous donnez à une application une interface propre et moderne, sans écrire de feuille de style. Les exemples viennent de la Ludothèque des Tilleuls,

qui prête des jeux de société à ses membres. Ce chapitre se lit : vous pratiquerez au chapitre 14, dans le dépôt `tailwind-26`, qui contient les pages de la ludothèque encore sans style.

À la fin de ce chapitre, vous serez capables de :

1. **expliquer** ce qu'est une classe utilitaire, et **comparer** une page stylée avec Tailwind à une page stylée par une feuille de style écrite à la main,
2. **charger** le CSS et le JavaScript de Vite dans un layout, et **choisir** entre `npm run dev` et `npm run build`,
3. **composer** une page avec les classes d'espacement, de typographie, de couleur, de bordure, de Flex et de Grid, adaptée à la largeur de l'écran et aux états d'un bouton,
4. **regrouper** ces classes dans des composants Blade à variantes, sans construire de nom de classe à partir d'une variable,
5. **trouver** une classe dans la documentation, et **écrire** un court script dans `resources/js/app.js`.

1. Une carte, deux façons d'écrire le style

Le catalogue affiche chaque jeu dans une carte : un cadre blanc, un titre, une ligne grise. Voici cette carte écrite comme dans un cours de CSS, avec une classe inventée et une feuille de style :

```css
.game-card {
    padding: 1.5rem;
    background-color: white;
    border: 1px solid #e4e4e7;
    border-radius: 0.5rem;
    box-shadow: 0 1px 2px 0 rgb(0 0 0 / 0.05);
}

.game-card h2 {
    font-size: 1.125rem;
    font-weight: 600;
}

.game-card p {
    margin-top: 0.5rem;
    color: #52525b;
}
```

```xml
<article class="game-card">
    <h2>Les Comptoirs de Safran</h2>
    <p>2 à 4 joueurs · 90 min</p>
</article>
```

Avec Tailwind CSS, la même carte s'écrit sans feuille de style. Chaque balise reçoit une liste de classes déjà prêtes :

```xml
<article class="rounded-lg border border-zinc-200 bg-white p-6 shadow-xs">
    <h2 class="text-lg font-semibold">Les Comptoirs de Safran</h2>
    <p class="mt-2 text-zinc-600">2 à 4 joueurs · 90 min</p>
</article>
```

Chaque classe applique une seule propriété CSS. `p-6` règle la marge intérieure, `mt-2` la marge au-dessus du paragraphe, `text-lg` la taille du texte. Une classe de ce genre s'appelle une **classe utilitaire**. Tailwind lit vos vues, y trouve ces noms, et écrit pour chacun une règle dans le fichier CSS qu'il produit. Voici trois de ces règles :

```
.p-6{padding:calc(var(--spacing) * 6)}
.mt-2{margin-top:calc(var(--spacing) * 2)}
.text-lg{font-size:var(--text-lg);line-height:var(--tw-leading,var(--text-lg--line-height))}
```

La variable `--spacing` vaut `0.25rem`. `p-6` donne donc 6 × 0,25 rem, soit le `1.5rem` de la feuille de style. Tailwind propose ainsi une échelle de valeurs, et vous choisissez dans cette échelle au lieu d'écrire des nombres.

Cette façon de travailler s'appelle **utility-first** : vous partez des classes utilitaires, et vous n'écrivez du CSS que pour ce qu'elles ne couvrent pas. La liste de classes est longue, mais vous la lisez à l'endroit où elle s'applique, sans ouvrir un second fichier.

La longueur ne se répète pas d'une page à l'autre si la carte devient un composant Blade, comme au chapitre 9. Vous écrivez la liste une fois, dans `components/card.blade.php`, et chaque page écrit `<x-card>`.

Le code qui sent mauvais

En binôme, cinq minutes. Une ludothèque a stylé son site avec une feuille de style de 900 lignes, écrite à la main. Elle contient ces règles :

```css
.loan-card {
    padding: 1rem;
    border-left: 4px solid #b91c1c;
}

.loan-card .late {
    color: #b91c1c;
    font-weight: 700;
}
```

La page des prêts a été réécrite le mois dernier. Pouvez-vous supprimer ces deux règles ? Comment en êtes-vous sûrs ?

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/tailwindcss

2. Brancher Tailwind dans le layout

Ce que `laravel new` a déjà installé

Tailwind est déjà installé dans tout projet créé avec `laravel new`, comme `premier-test-laravel`. Le fichier `package.json` contient les paquets `tailwindcss` et `@tailwindcss/vite`. Le fichier `vite.config.js`, vu au chapitre 6, charge l'extension de Tailwind pour Vite. Ouvrez maintenant `resources/css/app.css` :

```css
@import 'tailwindcss';

@source
'../../vendor/laravel/framework/src/Illuminate/Pagination/resources/views/*.blade.php';
@source '../../storage/framework/views/*.php';

@theme {
    --font-sans: 'Instrument Sans', ui-sans-serif, system-ui, sans-serif, 'Apple Color
Emoji', 'Segoe UI Emoji',
        'Segoe UI Symbol', 'Noto Color Emoji';
}
```

- **`@import 'tailwindcss';`** charge Tailwind. Cette ligne apporte une remise à zéro des styles du navigateur, les variables de l'échelle, comme `--spacing`, et la capacité de produire les classes utilitaires.
- **Les deux `@source`** ajoutent des dossiers à lire. Tailwind lit déjà tous les fichiers du projet, sauf ceux que Git ignore, comme `vendor/` et `storage/framework/views/`. La première ligne ajoute les templates de pagination de Laravel, rangés dans `vendor/`. La seconde ajoute les vues compilées, que Blade écrit dans `storage/framework/views/`.
- **`@theme`** déclare des **variables de thème**, des variables CSS qui créent des classes. Ici, `--font-sans` choisit la police Instrument Sans pour la classe `font-sans`, que Tailwind applique à toute la page.

Tailwind lit vos fichiers comme du texte. Il n'exécute ni PHP ni JavaScript : il cherche dans chaque fichier des mots qui ressemblent à des noms de classes. Vos vues de `resources/views/` et votre JavaScript de `resources/js/` sont donc lus sans aucune ligne à ajouter.

La seule ligne à ajouter : `@vite` dans le `<head>`

Le fichier `app.css` est une source. Le navigateur ne le charge pas tel quel : il faut lui donner le fichier que Vite produit à partir de lui. La directive Blade `@vite` écrit les balises qui chargent ce fichier. Dans le layout de la ludothèque, `components/layouts/app.blade.php`, ajoutez-la sous la balise `<title>`, avec la directive `@fonts` :

```xml
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>{{ $title }} · Ludothèque des Tilleuls</title>
    @fonts
    @vite(['resources/css/app.css', 'resources/js/app.js'])
</head>
```

- `@vite([...])` reçoit les deux sources que `vite.config.js` déclare, le CSS et le JavaScript, et écrit une balise pour chacune.
- `@fonts` charge la police Instrument Sans. `vite.config.js` la télécharge, et `@theme` la choisit. Sans `@fonts`, le navigateur utilise la police suivante de la liste, celle du système. La page d'accueil de Laravel, `welcome.blade.php`, contient les mêmes deux directives.
- **La balise `<meta name="viewport">`**, déjà présente, demande au téléphone d'afficher la page à sa vraie largeur. Sans elle, il dessine la page sur une largeur d'environ 1 000 pixels, puis la réduit. Le texte devient minuscule, et la page prend la mise en page prévue pour un écran large.

Ouvrez le catalogue, puis affichez son code source :

```
    <title>Catalogue · Ludothèque des Tilleuls</title>
…
    <link rel="preload" as="style" href="http://tailwind-26.test/build/assets/app-D32E-
bvF.css" /><link rel="modulepreload" as="script" href="http://tailwind-
26.test/build/assets/app-C5TGVq_p.js" /><link rel="stylesheet" href="http://tailwind-
26.test/build/assets/app-D32E-bvF.css" /><script type="module" src="http://tailwind-
26.test/build/assets/app-C5TGVq_p.js"></script></head>
```

`@fonts` a écrit les lignes masquées par `…`, qui chargent la police. `@vite` a écrit une balise `<link>` pour le CSS et une balise `<script>` pour le JavaScript. Les deux fichiers viennent de `public/build/assets/`. La suite de caractères dans leur nom, comme `D32E-bvF`, change à chaque fois que leur contenu change. Le navigateur ne peut donc pas garder en mémoire une ancienne version.

`npm run build` et `npm run dev`

Au chapitre 4, `laravel new` a lancé `npm run build`. Cette commande lit les sources une fois, écrit le résultat dans `public/build/`, puis s'arrête :

```bash
npm run build
```

```
> build
> vite build

vite v8.3.2 building client environment for production...
…
public/build/manifest.json                                  1.47 kB │ gzip: 0.33 kB
…
public/build/assets/app-D32E-bvF.css                       29.46 kB │ gzip: 6.28 kB
public/build/assets/app-C5TGVq_p.js                         0.24 kB │ gzip: 0.17 kB

✓ built in 801ms
```

Le fichier CSS ne contient que les classes trouvées dans vos fichiers au moment du build. Si vous écrivez ensuite dans une vue une classe qu'aucun fichier n'utilisait, comme `bg-red-50`, elle n'a aucun effet : aucune règle ne lui correspond dans `public/build/`. **Pendant que vous travaillez, laissez tourner `npm run dev` dans un terminal :**

```bash
npm run dev
```

```
> dev
> vite

  VITE v8.3.2  ready in 319 ms
…
  LARAVEL v13.34.0  plugin v3.2.0
```

La commande ne rend pas la main. Vite surveille vos fichiers, ajoute les règles des nouvelles classes et recharge la page ouverte dans le navigateur. Le code source de la page montre alors d'autres balises :

```
    <script type="module" src="http://…/@vite/client"></script><link rel="stylesheet"
href="http://…/resources/css/app.css" /><script type="module"
src="http://…/resources/js/app.js"></script></head>
```

Pendant que `npm run dev` tourne, Vite écrit son adresse dans le fichier `public/hot`. `@vite` y lit cette adresse, et les balises désignent Vite au lieu de `public/build/`. Quand vous arrêtez la commande avec Ctrl+C, le fichier `public/hot` disparaît, et `@vite` revient aux fichiers de `public/build/`.

Vous avez vu au chapitre 4 que `composer run dev` lance plusieurs programmes, dont `vite` : quatre sous macOS et Linux, trois sous Windows. Ce programme est `npm run dev` : si `composer run dev` tourne déjà, vous n'avez rien d'autre à lancer.

Lancez `npm run build` avant de livrer un projet, ou quand vous arrêtez de travailler. La page reste alors stylée sans Vite.

Deux messages à reconnaître

Dans un projet cloné avec Git, comme `eloquent-26` au chapitre 10, le dossier `public/build/` n'existe pas : il est exclu de Git. Une page qui utilise `@vite` renvoie alors une erreur 500 :

```
Vite manifest not found at: …/public/build/manifest.json
```

Le **manifeste** est le fichier qui associe chaque source à son fichier produit, comme `app.css` à `app-D32E-bvF.css`. Lancez `npm install`, puis `npm run build` ou `npm run dev`.

> **ASTUCE**
>
> Les deux commandes affichent aussi un avertissement jaune sur le paquet `fontaine`. Il concerne l'affichage de la police pendant son chargement, et vous pouvez l'ignorer. Si une page perd tous ses styles après l'arrêt de `npm run dev`, vérifiez si le fichier `public/hot` existe encore, et supprimez-le.

3. Les familles de classes

Les tableaux de cette section rangent par famille les classes que vous utiliserez au chapitre 14. Ils ne sont pas à apprendre par cœur : la documentation de Tailwind et l'extension Tailwind CSS IntelliSense de VS Code vous les rappellent au moment d'écrire (section 7).

Espacement et tailles

Chaque page de la ludothèque place son contenu dans un bloc centré, de largeur limitée :

```xml
<main class="mx-auto max-w-6xl px-4 py-8">
    <h1 class="text-3xl font-semibold tracking-tight">Catalogue</h1>
    <div class="mt-6 space-y-4">
        …
    </div>
</main>
```

`max-w-6xl` limite la largeur de `<main>` à 72 rem, et `mx-auto` partage l'espace restant à gauche et à droite : le bloc est centré. `px-4` évite que le texte touche les bords sur un téléphone. `space-y-4` écarte chaque enfant du `<div>` de son voisin du dessus.

La **marge intérieure**, `padding` en CSS, est l'espace entre le bord d'un élément et son contenu. La **marge extérieure**, `margin` en CSS, est l'espace entre l'élément et ses voisins. Le nombre d'une classe d'espacement est un multiple de `0.25rem`, soit 4 pixels avec la taille de texte par défaut du navigateur.

Classe	Ce qu'elle fait	Exemple
`p-{n}`	marge intérieure sur les quatre côtés	`p-6`, 1,5 rem

Classe	Ce qu'elle fait	Exemple
`px-{n}`, `py-{n}`, `pt-{n}`	marge intérieure à gauche et à droite (`x`), en haut et en bas (`y`), en haut seulement (`t`)	`px-4 py-2`
`mt-{n}`, `mb-{n}`	marge extérieure en haut, en bas	`mt-6`
`mx-auto`	centre un bloc de largeur limitée	`mx-auto max-w-6xl`
`space-y-{n}`	espace vertical entre les enfants	`space-y-4`
`w-full`, `max-w-{taille}`	largeur de 100 %, largeur maximale	`max-w-md`, `max-w-6xl`
`h-full`, `min-h-screen`	hauteur du parent, au moins la hauteur de la fenêtre	`min-h-screen`
`size-{n}`	largeur et hauteur ensemble	`size-8`

Typographie

Le titre du catalogue, puis une ligne plus petite et grise qui compte les jeux :

```xml
<h1 class="text-3xl font-semibold tracking-tight">Catalogue</h1>
<p class="mt-2 text-zinc-600">{{ $games->count() }} jeux à emprunter pour trois semaines.
</p>
```

Classe	Ce qu'elle fait	Exemple
`text-{taille}`	taille du texte, avec une hauteur de ligne adaptée	`text-xs`, `text-sm`, `text-base`, `text-lg`, `text-3xl`
`font-{graisse}`	épaisseur des lettres	`font-normal`, `font-medium`, `font-semibold`
`leading-{valeur}`	hauteur de ligne	`leading-tight`, `leading-relaxed`
`tracking-{valeur}`	espace entre les lettres	`tracking-tight`, `tracking-wide`
`uppercase`	tout en majuscules	`uppercase`
`text-center`	texte centré	`text-left`, `text-center`

La ludothèque télécharge sa police en trois graisses : 400, 500 et 600. Ses titres utilisent donc `font-semibold`, la graisse 600. Avec `font-bold`, la graisse 700, le navigateur ne trouve aucun fichier : il se rabat sur la graisse 600 ou imite un gras plus épais.

Couleurs

Un message d'alerte en haut de la page des prêts :

```xml
<p class="rounded-md bg-red-50 px-4 py-3 text-sm text-red-800">
    Deux prêts sont en retard.
</p>
```

Une classe de couleur réunit trois parties : ce qui est coloré, une teinte et une nuance. `bg-red-50` colore le fond, `text-red-800` le texte. Tailwind propose une vingtaine de teintes, dont `zinc`, `gray`, `red`, `amber`, `green`, `teal`, `sky` et `indigo`. Chacune existe en onze nuances, de `50`, presque blanc, à `950`, presque noir. S'y ajoutent `black` et `white`. La ludothèque utilise `zinc`, un gris neutre, pour le texte, les fonds et les bordures.

Classe	Ce qu'elle colore	Exemple
`bg-{couleur}`	le fond	`bg-white`, `bg-zinc-50`
`text-{couleur}`	le texte	`text-zinc-600`
`border-{couleur}`	la bordure	`border-zinc-200`
`{classe}/{opacité}`	la même couleur, transparente	`bg-black/50`, à moitié transparent
`bg-linear-to-{direction}`, `from-`, `to-`	un dégradé	`bg-linear-to-r from-zinc-50 to-zinc-200`

Pour un texte, gardez un contraste suffisant avec son fond : au moins 4,5 pour 1 pour un texte courant. Sur un fond blanc, `text-zinc-500` atteint 4,8 pour 1, et `text-zinc-600` 7,7 pour 1. `text-zinc-400` n'atteint que 2,6 pour 1 : il paraît lisible sur un écran neuf, mais plus sur un vieil écran ni en plein soleil.

Une couleur de marque avec `@theme`

La ludothèque a sa couleur, un bleu canard, qui ne fait pas partie des teintes de Tailwind. Elle la déclare dans le bloc `@theme` de `resources/css/app.css`, sous `--font-sans`, en onze nuances :

```css
@theme {
    --font-sans: 'Instrument Sans', ui-sans-serif, system-ui, sans-serif, 'Apple Color Emoji', 'Segoe UI Emoji',
        'Segoe UI Symbol', 'Noto Color Emoji';

    /* La couleur de la ludothèque : un bleu canard, de 50 (très clair) à 950 (très foncé). */
    --color-brand-50: oklch(0.98 0.016 195);
    --color-brand-100: oklch(0.95 0.04 195);
    --color-brand-200: oklch(0.9 0.07 195);
    --color-brand-300: oklch(0.83 0.1 195);
    --color-brand-400: oklch(0.74 0.12 195);
    --color-brand-500: oklch(0.64 0.12 196);
    --color-brand-600: oklch(0.54 0.105 198);
    --color-brand-700: oklch(0.46 0.09 200);
    --color-brand-800: oklch(0.4 0.075 201);
    --color-brand-900: oklch(0.35 0.06 202);
    --color-brand-950: oklch(0.25 0.045 204);
}
```

Une variable dont le nom commence par `--color-` crée une couleur, avec toutes ses classes : `bg-brand-600`, `text-brand-700`, `border-brand-600`, et les autres. Une valeur `oklch(…)` donne trois nombres : la luminosité, de 0 à 1, l'intensité de la couleur, puis sa teinte sur un cercle de 360 degrés. Les nuances gardent ici la même teinte, autour de 195, et deviennent de plus en plus sombres.

```
.bg-brand-600{background-color:var(--color-brand-600)}
```

Les vues écrivent `brand`, le rôle de la couleur, et non `teal` ou `sky`. Si la ludothèque change de couleur, vous modifiez ces onze lignes, et tout le site suit. Le texte blanc sur `brand-600` atteint 4,7 pour 1, au-dessus du minimum de 4,5 pour 1.

Bordures et ombres

Un champ de recherche du catalogue :

```xml
<input type="search" name="q" placeholder="Chercher un jeu"
    class="w-full rounded-md border border-zinc-300 bg-white px-3 py-2 text-sm shadow-xs">
```

Classe	Ce qu'elle fait	Exemple
`border`, `border-b`	bordure de 1 pixel, sur les quatre côtés ou en bas	`border border-zinc-200`
`rounded-{taille}`	coins arrondis	`rounded-md`, `rounded-lg`, `rounded-xl`, `rounded-full`
`shadow-{taille}`	ombre portée	`shadow-xs`, `shadow-sm`, `shadow-md`

Classe	Ce qu'elle fait	Exemple
`ring-{n}`	contour dessiné par une ombre, qui ne décale pas les voisins	`ring-1 ring-zinc-300`
`divide-y`	trait horizontal entre les enfants	`divide-y divide-zinc-200`

Écrivez toujours la couleur d'une bordure, comme `border border-zinc-200`. Les exemples écrits pour Tailwind 3 l'omettent souvent. Dans ces exemples, `border`, `shadow-sm` et `ring` n'ont pas le même effet qu'en Tailwind 4 : la section 7 compare les deux versions.

Flex

Une ligne de prêt aligne le jeu et le membre à gauche, et le statut à droite :

```xml
<div class="flex items-center justify-between gap-4 border-b border-zinc-200 py-3">
    <div>
        <p class="font-medium">Marée d'étoiles</p>
        <p class="text-sm text-zinc-600">Emprunté par Amina Benali, à rendre le 12
octobre</p>
    </div>
    <span class="shrink-0 text-sm font-medium text-red-700">En retard</span>
</div>
```

`flex` place les deux enfants du `<div>` côte à côte. `justify-between` pousse le premier à gauche et le dernier à droite. `items-center` les centre en hauteur. `shrink-0` empêche le statut de rétrécir quand la phrase de gauche est longue : c'est elle qui passe à la ligne.

Classe	Ce qu'elle fait	Exemple
`flex`, `flex-col`	aligne les enfants en ligne, ou en colonne	`flex flex-col`
`flex-wrap`	passe à la ligne quand la place manque	`flex flex-wrap`
`items-{position}`	position des enfants en hauteur, quand ils sont en ligne	`items-center`, `items-start`
`justify-{position}`	répartition des enfants dans le sens de la ligne	`justify-between`, `justify-end`
`gap-{n}`	espace entre les enfants	`gap-4`
`flex-1`	l'enfant prend la place restante	`flex-1`

Grid

Le catalogue range les cartes des jeux sur trois colonnes de même largeur :

```xml
<ul class="grid grid-cols-3 gap-6">
    <li><article class="rounded-lg border border-zinc-200 bg-white p-6 shadow-xs">…
</article></li>
    <li><article class="rounded-lg border border-zinc-200 bg-white p-6 shadow-xs">…
</article></li>
    <li><article class="rounded-lg border border-zinc-200 bg-white p-6 shadow-xs">…
</article></li>
</ul>
```

Une grille découpe la largeur en colonnes, et range les enfants de gauche à droite, puis ligne par ligne. Avec Flex, la largeur d'un enfant dépend de son contenu. Avec Grid, elle dépend de la colonne : les cartes sont alignées même si leurs titres n'ont pas la même longueur. `grid-cols-{n}` fixe le nombre de colonnes, et `gap-{n}` écarte les lignes et les colonnes.

4. Écrans et états

D'abord le téléphone

Trois colonnes sur un téléphone de 375 pixels de large donnent des cartes d'environ 100 pixels, où « Les Comptoirs de Safran » ne tient pas. Une classe précédée d'un préfixe de largeur, comme `sm:`, ne s'applique qu'**à partir de** cette largeur. Voici la grille du catalogue de la ludothèque :

```xml
<ul class="grid gap-6 sm:grid-cols-2 lg:grid-cols-3 xl:grid-cols-4">
```

`grid` seul range les cartes sur une colonne, à toutes les largeurs. À partir de 640 pixels, `sm:grid-cols-2` passe à deux colonnes. À partir de 1 024 pixels, `lg:grid-cols-3` passe à trois, et `xl:grid-cols-4` à quatre dès 1 280 pixels. Tailwind écrit ces classes dans une **requête média**, un bloc de règles que le navigateur n'applique que si la fenêtre remplit une condition. Ici, la fenêtre doit mesurer au moins 40 rem de large :

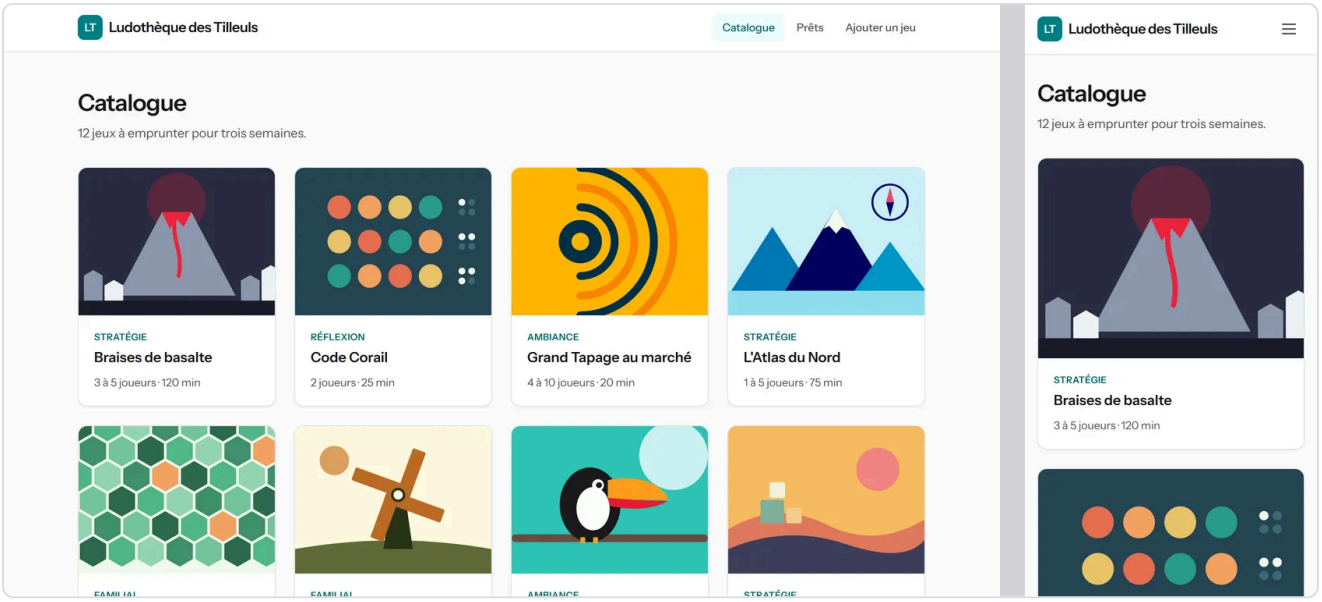
```
@media (width ⩾ 40rem){… .sm\:grid-cols-2{grid-template-columns:repeat(2,minmax(0,1fr))} …}
```

Dans le nom de la règle, `\:` protège le deux-points du préfixe, qui a un autre sens en CSS.

Vous écrivez donc d'abord la page du téléphone, sans préfixe, puis vous ajoutez ce qui change sur un écran plus large. Cette façon de faire s'appelle **mobile first**. `sm:` ne veut pas dire « sur un petit écran », mais « à partir de 640 pixels ».

Préfixe	À partir de	Écran typique
aucun	0 pixel	téléphone
`sm:`	640 pixels (40 rem)	grand téléphone couché
`md:`	768 pixels (48 rem)	tablette

Préfixe	À partir de	Écran typique
`lg:`	1 024 pixels (64 rem)	ordinateur portable
`xl:`	1 280 pixels (80 rem)	écran de bureau
`2xl:`	1 536 pixels (96 rem)	grand écran



Pour tester, ouvrez les outils de développement du navigateur avec F12, puis le mode d'affichage mobile, Ctrl+Maj+M dans Chrome et Firefox. Vous choisissez alors la largeur de la page.

Les états d'un bouton

Un bouton change d'aspect quand la souris le survole, quand il reçoit le focus au clavier, et quand il est désactivé. Un préfixe d'état ne s'applique que dans cet état :

```XML
<button type="submit"
    class="rounded-lg bg-brand-600 px-4 py-2 text-sm font-semibold text-white hover:bg-brand-700 focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-brand-600 disabled:cursor-not-allowed disabled:opacity-50">
    Ajouter le jeu
</button>
```

Tailwind place lui aussi `hover:` dans une requête média. La règle ne vaut que sur un appareil qui a une souris, et non sur un écran tactile :

```
@media (hover:hover){.hover\:bg-brand-700:hover{background-color:var(--color-brand-700)} …}
```

Le **focus** est l'état de l'élément qui reçoit les touches du clavier. La touche Tab le fait passer d'un lien ou d'un bouton au suivant. Sur un bouton, `focus-visible:` ne s'applique qu'au focus reçu par le clavier, pas après un clic de souris.

Préfixe	S'applique	Exemple
`hover:`	au survol de la souris	`hover:bg-brand-700`
`focus-visible:`	au focus au clavier	`focus-visible:outline-2`
`active:`	pendant le clic	`active:bg-brand-800`
`disabled:`	quand l'élément a l'attribut `disabled`	`disabled:opacity-50`

ATTENTION

Ne retirez jamais le contour de focus sans le remplacer. Beaucoup d'exemples écrivent `outline-none` sur les boutons et les champs, parce que le contour du navigateur ne leur plaît pas. Une personne qui navigue au clavier ne sait alors plus où elle se trouve dans la page. Si vous changez ce contour, écrivez-en un autre avec `focus-visible:`, puis parcourez la page avec la touche Tab.

Combiner les préfixes

Les préfixes s'additionnent. Écrivez d'abord la largeur, puis l'état :

```xml
<a href="{{ route('games.index') }}" class="block py-2 md:py-0
md:hover:underline">Catalogue</a>
```

`md:hover:underline` ne souligne le lien qu'au survol, et seulement à partir de 768 pixels.

Le mode sombre

Le préfixe `dark:` applique une classe quand le système de la personne est réglé en mode sombre. Voici le `<body>` du layout de la ludothèque :

```xml
<body class="flex min-h-screen flex-col bg-zinc-50 font-sans text-zinc-900 antialiased
dark:bg-zinc-950 dark:text-zinc-100">
```

En mode clair, la page a un fond `zinc-50` et un texte `zinc-900`. En mode sombre, les deux s'inversent : fond `zinc-950`, texte `zinc-100`. Chaque classe de couleur demande ainsi sa version sombre, dans chaque composant et chaque vue. Un mode sombre à moitié fait affiche du texte gris foncé sur fond noir : ajoutez les classes `dark:` partout, ou nulle part.

5. Des classes aux composants Blade

Le bouton de la section 4 compte treize classes. La ludothèque en affiche partout : ajouter un jeu, annuler, supprimer. Recopier ces treize classes dans chaque vue reviendrait à la situation du chapitre 9, avant le layout : un changement de couleur demanderait d'ouvrir toutes les vues. Le composant Blade règle ce problème ici aussi.

La carte et son image

Voici le composant `card` de la ludothèque, `components/card.blade.php` :

```XML
<article {{ attributes->class([
    'overflow-hidden rounded-xl border border-zinc-200 bg-white shadow-xs',
    'dark:border-zinc-800 dark:bg-zinc-900',
]) }}>
    @isset(image)
        {{ image }}
    @endisset

    <div class="p-5">
        {{ slot }}
    </div>
</article>
```

- **`$attributes->class([...])`** écrit les classes du tableau, puis ajoute celles de la balise `<x-card>`, comme `merge()` au chapitre 9.
- **`@isset($image)`** affiche le slot nommé `image` seulement si la page en donne un, comme `$note` au chapitre 9. L'image touche les bords de la carte, et `overflow-hidden` coupe ses coins pour suivre l'arrondi `rounded-xl`.
- **`<div class="p-5">`** écarte le texte des bords, mais pas l'image.

Le catalogue n'écrit plus que le contenu de chaque carte :

```XML
<ul class="grid gap-6 sm:grid-cols-2 lg:grid-cols-3 xl:grid-cols-4">
    @foreach (gamesasgames as gamesasgame)
        <li>
            <x-card class="h-full transition-shadow hover:shadow-md">
                <x-slot:image>
                    <img src="{{ asset(game->coverPath()) }}" alt="" width="400"
height="300" class="aspect-4/3 w-full object-cover">
                </x-slot:image>

                <p class="text-xs font-semibold tracking-wide text-brand-700 uppercase
dark:text-brand-300">{{ game->category }}</p>
                <h2 class="mt-1 text-lg font-semibold">
                    <a href="{{ route('games.show', game->id) }}" class="rounded-sm
hover:underline hover:underline-offset-4 focus-visible:outline-2 focus-visible:outline-
offset-2 focus-visible:outline-brand-600">{{ game->title }}</a>
                </h2>
                <p class="mt-2 text-sm text-zinc-600 dark:text-zinc-400">{{ game
->playersLabel() }} · {{ game->duration }} min</p>
            </x-card>
        </li>
    @endforeach
</ul>
```

`class="h-full …"` s'ajoute aux classes du composant : toutes les cartes d'une ligne de la grille prennent la même hauteur, et leur ombre grandit au survol. `aspect-4/3` donne à l'image un rapport largeur sur hauteur de 4 pour 3, et `object-cover` la recadre sans la déformer.

Des classes selon une condition : `@class`

Le statut d'un prêt s'affiche dans une pastille dont la couleur dépend du statut : bleu pour un prêt en cours, rouge pour un retard, gris pour un jeu rendu. La directive `@class` reçoit un tableau. Une valeur sans clé est toujours écrite. Une clé est écrite seulement si sa valeur vaut `true`. Voici le composant `badge` :

```XML
@props(['status'])

<span @class([
    'inline-flex items-center rounded-full px-2.5 py-0.5 text-xs font-medium whitespace-nowrap ring-1 ring-inset',
    'bg-sky-50 text-sky-700 ring-sky-600/20 dark:bg-sky-400/10 dark:text-sky-300 dark:ring-sky-400/30' => status===’ongoing’,
’bg-red-50text-red-700ring-red-600/20dark:bg-red-400/10dark:text-red-300dark:ring-red-400/30’=>status === 'ongoing',
    'bg-red-50 text-red-700 ring-red-600/20 dark:bg-red-400/10 dark:text-red-300 dark:ring-red-400/30' => status===’ongoing’,
’bg-red-50text-red-700ring-red-600/20dark:bg-red-400/10dark:text-red-300dark:ring-red-400/30’=>status === 'overdue',
    'bg-zinc-100 text-zinc-600 ring-zinc-500/20 dark:bg-zinc-400/10 dark:text-zinc-300 dark:ring-zinc-400/30' => $status === 'returned',
])>
    {{ $slot }}
</span>
```

```XML
<x-badge status="overdue">En retard</x-badge>
```

```
<span class="inline-flex items-center rounded-full px-2.5 py-0.5 text-xs font-medium whitespace-nowrap ring-1 ring-inset bg-red-50 text-red-700 ring-red-600/20 dark:bg-red-400/10 dark:text-red-300 dark:ring-red-400/30">
    En retard
</span>
```

`@class` a gardé la première chaîne et la seule clé dont la condition est vraie. `ring-1 ring-inset ring-red-600/20` trace un contour de 1 pixel à l'intérieur de la pastille, dans un rouge opaque à 20 %.

Le statut ne vient pas d'une colonne de la base. Le modèle `Loan` le calcule à partir des dates du prêt :

```php
public function status(): string
{
    if (this->returnedat≠=null)return’returned’;if(this→returned_at ≠= null) {
        return 'returned';
    }

    if (this->returnedat≠=null)return’returned’;if(this→due_at→lt(today())) {
        return 'overdue';
    }

    return 'ongoing';
}
```

`lt()` signifie « plus petit que » : la date de retour prévue est passée. Une seconde méthode, `statusLabel()`, traduit le statut avec l'expression `match` du chapitre 7 de la POO. La page des prêts écrit donc :

```xml
<x-badge :status="$loan→status()">{{ $loan→statusLabel() }}</x-badge>
```

Le piège des noms construits

Pour les catégories des jeux, une étiquette colorée semble plus simple à écrire avec une prop `color` placée dans le nom de la classe. Voici un composant `tag` écrit de cette façon :

```xml
@props(['color'])

<span class="rounded-md bg-{{ color }}-100 px-2 py-1 text-xs text-{{ color }}-800">
    {{ $slot }}
</span>
```

```xml
<x-tag color="lime">Coopératif</x-tag>
```

```
<span class="rounded-md bg-lime-100 px-2 py-1 text-xs text-lime-800">
    Coopératif
</span>
```

Le HTML contient bien `bg-lime-100`, mais l'étiquette s'affiche sans fond vert. Cherchez la classe dans le CSS produit :

```bash
npm run build
grep -c "bg-lime-100" public/build/assets/app-*.css
```

```
0
```

Sous PowerShell, `Select-String -Path public/build/assets/app-*.css -Pattern "bg-lime-100"` remplace `grep`, et n'affiche alors rien.

Aucune règle. Tailwind lit `tag.blade.php` comme du texte, sans exécuter Blade : il y trouve `bg-{{ $color }}-100`, et non `bg-lime-100`. Le nom complet n'existe qu'au moment où Laravel affiche la page, bien après le build. **Écrivez toujours les noms de classes en entier dans vos fichiers.** Faites correspondre chaque valeur de la prop à des classes complètes, avec `@class`, comme dans `badge` :

```xml
@props(['color'])

<span @class([
    'rounded-md px-2 py-1 text-xs',
    'bg-lime-100 text-lime-800' ⇒ color═══'lime','bg-sky-100text-sky-800'⇒color ═══
'lime',
    'bg-sky-100 text-sky-800' ⇒ color═══'lime','bg-sky-100text-sky-800'⇒color ═══ 'sky',
])>
    {{ $slot }}
</span>
```

```bash
npm run build
grep -c "bg-lime-100" public/build/assets/app-*.css
```

```
1
```

La règle vaut aussi pour PHP et JavaScript. `'bg-' . $color . '-100'` dans un contrôleur, ou une classe assemblée dans `app.js`, ne produisent pas de règle CSS.

Un bouton à variantes

Le bouton existe en trois **variantes**, trois aspects choisis par une prop : principal, secondaire et danger. Voici le composant `button` :

```xml
@props(['variant' ⇒ 'primary'])

<button {{ attributes->merge(['type'⇒'button'])
->class(['inline-flexitems-centerjustify-centergap-2rounded-lgpx-4py-2text-smfont-semiboldsh
adow-xstransition-colors','focus-visible:outline-2focus-visible:outline-offset-2',
'disabled:cursor-not-alloweddisabled:opacity-50',
'bg-brand-600text-whitehover:bg-brand-700focus-visible:outline-brand-600disabled:hover:bg-br
and-600'⇒attributes→merge(['type' ⇒ 'button'])→class([
    'inline-flex items-center justify-center gap-2 rounded-lg px-4 py-2 text-sm font-
semibold shadow-xs transition-colors',
    'focus-visible:outline-2 focus-visible:outline-offset-2',
    'disabled:cursor-not-allowed disabled:opacity-50',
    'bg-brand-600 text-white hover:bg-brand-700 focus-visible:outline-brand-600
disabled:hover:bg-brand-600' ⇒ attributes->merge(['type'⇒'button'])
->class(['inline-flexitems-centerjustify-centergap-2rounded-lgpx-4py-2text-smfont-semiboldsh
adow-xstransition-colors','focus-visible:outline-2focus-visible:outline-offset-2',
'disabled:cursor-not-alloweddisabled:opacity-50',
'bg-brand-600text-whitehover:bg-brand-700focus-visible:outline-brand-600disabled:hover:bg-br
and-600'⇒variant ⩵ 'primary',
    'bg-white text-zinc-900 ring-1 ring-zinc-300 ring-inset hover:bg-zinc-50 focus-
visible:outline-zinc-500 disabled:hover:bg-white dark:bg-zinc-800 dark:text-zinc-100
dark:ring-zinc-700 dark:hover:bg-zinc-700' ⇒ variant⩵'secondary',
'bg-red-600text-whitehover:bg-red-700focus-visible:outline-red-600disabled:hover:bg-red-600'
⇒variant ⩵ 'secondary',
    'bg-red-600 text-white hover:bg-red-700 focus-visible:outline-red-600 disabled:hover:bg-
red-600' ⇒ variant⩵'secondary',
'bg-red-600text-whitehover:bg-red-700focus-visible:outline-red-600disabled:hover:bg-red-600'
⇒variant ⩵ 'danger',
]) }}>
    {{ $slot }}
</button>
```

- **`→merge(['type' ⇒ 'button'])`** donne au bouton le type `button` par défaut. Sans attribut `type`, un `<button>` placé dans un formulaire l'envoie. Pour tout attribut autre que `class`, la valeur écrite dans la balise remplace celle de `merge()`.
- **`→class([...])`** reçoit le même tableau que `@class`, et y ajoute les classes écrites dans la balise `<x-button>`.
- **`disabled:hover:bg-brand-600`** garde la même couleur au survol d'un bouton désactivé : rien ne laisse croire qu'il réagit.

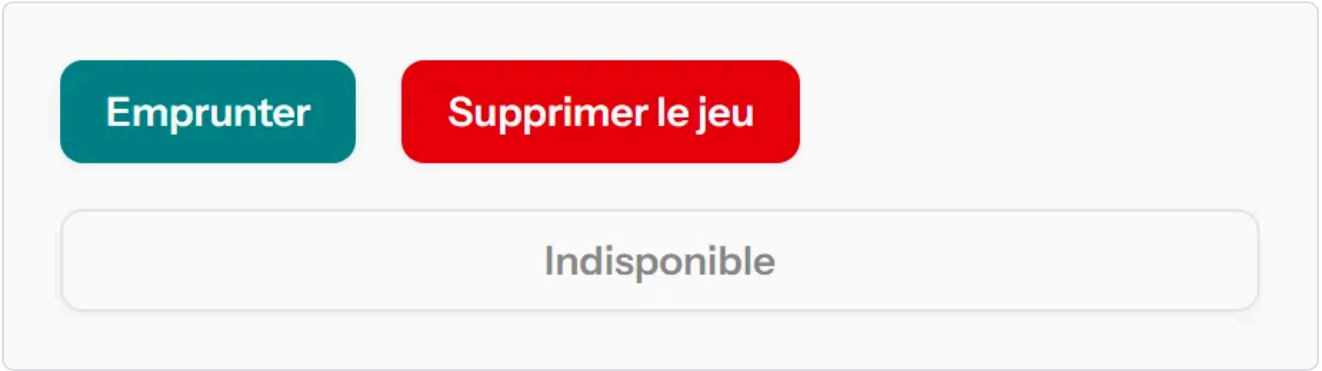
```xml
<x-button>Emprunter</x-button>
<x-button variant="danger" type="submit">Supprimer le jeu</x-button>
<x-button variant="secondary" class="w-full" disabled>Indisponible</x-button>
```

```
<button class="inline-flex items-center justify-center gap-2 rounded-lg px-4 py-2 text-sm
font-semibold shadow-xs transition-colors focus-visible:outline-2 focus-visible:outline-
offset-2 disabled:cursor-not-allowed disabled:opacity-50 bg-brand-600 text-white hover:bg-
brand-700 focus-visible:outline-brand-600 disabled:hover:bg-brand-600" type="button">
    Emprunter
</button>
<button class="inline-flex items-center justify-center gap-2 rounded-lg px-4 py-2 text-sm
font-semibold shadow-xs transition-colors focus-visible:outline-2 focus-visible:outline-
offset-2 disabled:cursor-not-allowed disabled:opacity-50 bg-red-600 text-white hover:bg-red-
700 focus-visible:outline-red-600 disabled:hover:bg-red-600" type="submit">
    Supprimer le jeu
</button>
<button class="inline-flex items-center justify-center gap-2 rounded-lg px-4 py-2 text-sm
font-semibold shadow-xs transition-colors focus-visible:outline-2 focus-visible:outline-
offset-2 disabled:cursor-not-allowed disabled:opacity-50 bg-white text-zinc-900 ring-1 ring-
zinc-300 ring-inset hover:bg-zinc-50 focus-visible:outline-zinc-500 disabled:hover:bg-white
dark:bg-zinc-800 dark:text-zinc-100 dark:ring-zinc-700 dark:hover:bg-zinc-700 w-full"
type="button" disabled="disabled">
    Indisponible
</button>
```

Le second bouton garde `type="submit"`, écrit dans la balise. Le troisième reçoit `w-full` à la fin de ses classes, et l'attribut `disabled` arrive dans la balise sous la forme `disabled="disabled"`. Chaque bouton principal du site utilise `brand-600` : si la ludothèque change de couleur, aucune vue ne change.

Une classe ajoutée dans la balise ne doit pas contredire une classe du composant. Essayez `<x-button class="bg-amber-600">Ambre</x-button>` puis `<x-button class="bg-orange-600">Orange</x-button>`. Les deux boutons reçoivent `bg-brand-600` et la seconde couleur. Le bouton « Ambre » reste bleu canard, et le bouton « Orange » devient orange.

Quand deux classes règlent la même propriété, le navigateur applique la règle placée le plus bas dans le fichier CSS. Tailwind range lui-même ses règles, et `.bg-amber-600` y précède `.bg-brand-600`, qui précède `.bg-orange-600`. L'ordre des classes dans l'attribut `class` ne compte pas. **Pour un autre aspect, ajoutez une variante au composant**, plutôt qu'une couleur dans la balise.



6. Le menu et son JavaScript

Sur un téléphone, les trois liens du menu prennent trop de place. La ludothèque les cache derrière un bouton à trois traits, le bouton « burger », et les affiche en ligne à partir de 768 pixels. Voici l'en-tête du layout :

```xml
<header class="border-b border-zinc-200 bg-white dark:border-zinc-800 dark:bg-zinc-900">
    <div class="mx-auto flex max-w-6xl flex-wrap items-center justify-between gap-x-6 gap-y-2 px-4 py-3 sm:px-6 lg:px-8">
        <a href="{{ route('games.index') }}" class="flex items-center gap-2 rounded-md text-lg font-semibold tracking-tight focus-visible:outline-2 focus-visible:outline-offset-4 focus-visible:outline-brand-600">
            <span class="grid size-8 place-items-center rounded-lg bg-brand-600 text-sm font-semibold text-white" aria-hidden="true">LT</span>
            Ludothèque des Tilleuls
        </a>

        <button
            type="button"
            id="menu-button"
            class="group rounded-md p-2 text-zinc-600 hover:bg-zinc-100 hover:text-zinc-900 focus-visible:outline-2 focus-visible:outline-brand-600 md:hidden dark:text-zinc-300 dark:hover:bg-zinc-800 dark:hover:text-white"
            aria-controls="main-menu"
            aria-expanded="false"
        >
            <span class="sr-only">Menu</span>
            <svg class="size-6 group-aria-expanded:hidden" viewBox="0 0 24 24" fill="none" stroke="currentColor" stroke-width="2" stroke-linecap="round" aria-hidden="true">
                <path d="M4 6h16M4 12h16M4 18h16" />
            </svg>
            <svg class="hidden size-6 group-aria-expanded:block" viewBox="0 0 24 24" fill="none" stroke="currentColor" stroke-width="2" stroke-linecap="round" aria-hidden="true">
                <path d="M6 6l12 12M18 6L6 18" />
            </svg>
        </button>

        <nav id="main-menu" class="hidden w-full md:block md:w-auto" aria-label="Navigation principale">
            <ul class="flex flex-col gap-1 border-t border-zinc-200 py-2 md:flex-row md:items-center md:border-0 md:py-0 dark:border-zinc-800">
                <li><x-nav-link :href="route('games.index')" :active="request()->routeIs('games.index', 'games.show')">Catalogue</x-nav-link></li>
                <li><x-nav-link :href="route('loans.index')" :active="request()->routeIs('loans.*')">Prêts</x-nav-link></li>
                <li><x-nav-link :href="route('games.create')" :active="request()->routeIs('games.create')">Ajouter un jeu</x-nav-link></li>
            </ul>
        </nav>
    </div>
</header>
```

- **Sur un téléphone**, `hidden` cache le `<nav>`, et la liste est en colonne avec `flex-col`. À partir de 768 pixels, `md:block` affiche le menu, `md:flex-row` met les liens en ligne, et `md:hidden` cache le bouton.
- `sr-only` cache le mot « Menu » à l'écran, mais les **lecteurs d'écran**, les programmes qui lisent la page à voix haute aux personnes aveugles ou malvoyantes, le lisent. Le bouton n'affiche qu'un dessin, et le lecteur annonce « Menu ».
- `group` marque le bouton. `group-aria-expanded:hidden` cache le premier dessin, les trois traits, quand le bouton porte `aria-expanded="true"`. `group-aria-expanded:block` affiche alors le second, une croix.

Le lien actif : `<x-nav-link>`

Chaque lien du menu est un composant qui reçoit son adresse et un booléen `active` :

```XML
@props(['href', 'active' => false])

<a
    href="{{ href }}"
    @if (active) aria-current="page" @endif
    @class([
        'block rounded-md px-3 py-2 text-sm font-medium transition-colors focus-visible:outline-2 focus-visible:outline-brand-600',
        'bg-brand-50 text-brand-700 dark:bg-brand-950 dark:text-brand-200' => active,
'text-zinc-600hover:bg-zinc-100hover:text-zinc-900dark:text-zinc-300dark:hover:bg-zinc-800dark:hover:text-white'=>!active,
        'text-zinc-600 hover:bg-zinc-100 hover:text-zinc-900 dark:text-zinc-300 dark:hover:bg-zinc-800 dark:hover:text-white' => ! active,
'text-zinc-600hover:bg-zinc-100hover:text-zinc-900dark:text-zinc-300dark:hover:bg-zinc-800dark:hover:text-white'=>!active,
    ])
>
    {{ $slot }}
</a>
```

`request()->routeIs('loans.*')` vaut `true` quand la page affichée vient d'une route dont le nom commence par `loans.`. Sur la page des prêts, le lien « Prêts » reçoit donc le fond `brand-50` et l'attribut `aria-current="page"`, qui annonce au lecteur d'écran la page en cours :

```XML
<x-nav-link href="/prets" :active="true">Prêts</x-nav-link>
```

```
<a
    href="/prets"
     aria-current="page"     class="block rounded-md px-3 py-2 text-sm font-medium transition-colors focus-visible:outline-2 focus-visible:outline-brand-600 bg-brand-50 text-brand-700 dark:bg-brand-950 dark:text-brand-200"
>
    Prêts
</a>
```

Le script du bouton

Il reste à ouvrir le menu quand vous cliquez sur le bouton. Ce code s'écrit dans `resources/js/app.js`, la seconde source de `@vite`. Dans un projet neuf, ce fichier ne contient qu'un commentaire vide, `//`. Voici celui de la ludothèque :

```javascript
// Le menu mobile : le bouton burger affiche ou cache la liste des liens.
const menuButton = document.querySelector('#menu-button');
const menu = document.querySelector('#main-menu');

if (menuButton && menu) {
    menuButton.addEventListener('click', () => {
        const isOpen = menuButton.getAttribute('aria-expanded') === 'true';

        menuButton.setAttribute('aria-expanded', String(!isOpen));
        menu.classList.toggle('hidden');
    });
}
```

- **`document.querySelector('#menu-button')`** renvoie l'élément dont l'`id` vaut `menu-button`, ou `null` s'il n'existe pas. Le `if` évite une erreur sur une page sans menu.
- **`addEventListener('click', …)`** exécute la fonction à chaque clic sur le bouton.
- **`getAttribute('aria-expanded') === 'true'`** lit l'état actuel du menu, et `setAttribute` écrit l'état inverse.
- **`classList.toggle('hidden')`** retire la classe `hidden` du `<nav>` si elle y est, et l'ajoute sinon.

Sur une page de 375 pixels, cliquez sur le bouton, puis ouvrez l'onglet Éléments des outils de développement :

```
<button type="button" id="menu-button" class="group …" aria-controls="main-menu" aria-
expanded="true">
…
<nav id="main-menu" class="w-full md:block md:w-auto" aria-label="Navigation principale">
```

La classe `hidden` a disparu du `<nav>`, et les trois liens s'affichent l'un sous l'autre. Le bouton porte `aria-expanded="true"` et montre la croix. Un second clic referme le menu.

Un seul attribut, `aria-expanded`, sert à la fois au lecteur d'écran, qui annonce si le menu est ouvert, et au choix du dessin par `group-aria-expanded:`. `aria-controls` indique au lecteur quel élément le bouton commande.

`@vite` charge `app.js` avec `type="module"`. Le navigateur exécute donc le script après avoir lu toute la page, et `querySelector` trouve le bouton et le menu. Tailwind lit aussi `app.js` : une classe que seul votre JavaScript ajoute doit y être écrite en entier, comme `'hidden'` ici.

7. Trouver une classe

La documentation officielle, tailwindcss.com/docs, consacre une page à chaque famille de classes. Sa recherche s'ouvre avec Ctrl+K. Tapez le nom de la propriété CSS que vous connaissez, en anglais : « letter spacing » mène à la page des classes `tracking-*`. Chaque page commence par un tableau qui donne, pour chaque classe, la règle CSS qu'elle produit. Voici celle de `tracking-tight`, le resserrement du titre du catalogue, dans le fichier produit :

```
.tracking-tight{--tw-tracking:var(--tracking-tight);letter-spacing:var(--tracking-tight)}
```

La variable `--tracking-tight` vaut `-.025em` : les lettres se rapprochent d'un quarantième de la taille du texte.

L'extension **Tailwind CSS IntelliSense**, dans la liste des extensions du chapitre 3, complète les noms de classes pendant que vous tapez dans un attribut `class`. Elle montre un carré de la couleur à côté de chaque classe de couleur. Au survol d'une classe, elle affiche la règle CSS correspondante.

Les tutoriels et les assistants de code IA proposent souvent du code écrit pour Tailwind 3. Certains anciens noms fonctionnent encore, d'autres ont changé d'effet ou n'existent plus :

Tailwind 3	Tailwind 4	L'ancien nom en Tailwind 4
un fichier `tailwind.config.js`	`@theme` dans `app.css`	le fichier n'est plus lu
`@tailwind base;` et deux lignes voisines	`@import 'tailwindcss';`	ne charge plus Tailwind en entier
`bg-opacity-50`	`bg-black/50`	aucun effet
`shadow-sm`	`shadow-xs`	une ombre plus marquée
`rounded-sm`	`rounded-xs`	un arrondi plus grand
`ring`	`ring-3`	un contour de 1 pixel
`border`	`border border-zinc-200`	une bordure de la couleur du texte
`bg-gradient-to-r`	`bg-linear-to-r`	fonctionne encore
`flex-shrink-0`	`shrink-0`	fonctionne encore

Si une classe trouvée en ligne n'a pas l'effet attendu, cherchez-la d'abord dans ce tableau, puis dans la documentation. Voici, raccourcies, les règles de `shadow-xs` et de `shadow-sm` dans le fichier produit :

```
.shadow-xs{--tw-shadow:0 1px 2px 0 var(--tw-shadow-color,#0000000d);…}
.shadow-sm{--tw-shadow:0 1px 3px 0 var(--tw-shadow-color,#0000001a), 0 1px 2px -1px var(--
tw-shadow-color,#0000001a);…}
```

`shadow-xs` reprend l'ombre de la section 1 : 1 pixel vers le bas, 2 pixels de flou, du noir à 5 %. `shadow-sm` superpose deux ombres noires à 10 %.

À retenir

- Une classe utilitaire applique une seule propriété CSS, et Tailwind ne produit que les classes trouvées dans vos fichiers.
- `@vite` charge le CSS et le JavaScript, et `npm run dev` tourne pendant le travail.
- Une classe sans préfixe vaut pour toutes les largeurs, et `md:` ou `lg:` l'adaptent aux écrans plus larges.
- Un composant Blade écrit une seule fois une longue liste de classes, choisies avec `@class` ou `$attributes->class()`.
- Un nom de classe s'écrit en entier, car Tailwind ne voit pas `bg-{{ $color }}-100`.

Six cartes

Qu'est-ce qu'une classe utilitaire ?

Une classe qui applique une seule propriété CSS, comme `p-6` pour la marge intérieure.

Pourquoi une nouvelle classe reste-t-elle sans effet après `npm run build` ?

Le CSS de `public/build/` ne contient que les classes présentes au moment du build. Lancez `npm run dev`.

Que veut dire `sm:grid-cols-2` ?

Deux colonnes à partir de 640 pixels de large.

Pourquoi `bg-{{ $color }}-100` ne colore-t-il rien ?

Tailwind lit le fichier comme du texte, et n'y trouve pas `bg-lime-100`.

`@class` ou `$attributes->class()` ?

`@class` sur une balise ordinaire. `$attributes->class()` dans un composant, qui garde les classes de `<x-…>`.

À quoi sert `@theme` ?

À déclarer des variables qui créent des classes : `--color-brand-600` crée `bg-brand-600`.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/tailwindcss

3.14 Tailwind CSS : exercices

Bloc Laravel · séances 6 et 7 · ~3 h

Au chapitre 13, vous avez vu les classes utilitaires, `@vite`, `@theme` et les composants Blade à variantes. Dans ce chapitre, vous mettez en forme un site entier, celui de la ludothèque des Tilleuls. Le site fonctionne déjà : il affiche un catalogue de jeux de société, la fiche de chaque jeu, la liste des prêts et un formulaire d'ajout. Ses vues ne contiennent pourtant aucune classe, et son layout ne charge aucun CSS.

À la fin de ce chapitre, vous serez capables de :

1. **brancher** Tailwind dans un layout, et **poser** un conteneur commun à toutes les pages,
2. **construire** des composants Blade à variantes avec `@class` et `$attributes→class()`,
3. **adapter** une grille, un tableau et un menu à la largeur de l'écran,
4. **placer** le JavaScript d'une page dans `resources/js/app.js`, et **décrire** l'état d'un menu avec `aria-expanded`,
5. **reproduire** un écran d'après une capture, et **vérifier** le résultat avec une liste de critères.

1. Le site de la ludothèque

Installer le dépôt (10 minutes, une seule fois)

Sur github.com/opmvpc/tailwind-26, créez votre copie avec le bouton Fork. Ouvrez ensuite un terminal dans le dossier de Herd ou de Laragon, et préparez le projet :

```bash
git clone https://github.com/VOTRE-COMPTE/tailwind-26.git
cd tailwind-26
composer install
cp .env.example .env
php artisan key:generate
php artisan migrate --seed
npm install
```

Les cinq premières commandes sont celles du dépôt `eloquent-26`, au chapitre 10. Avec `--seed`, `migrate` remplit aussi la base : douze jeux, huit membres et quinze prêts. Les dates des prêts partent du jour où vous lancez la commande, et vos dates diffèrent donc de celles des captures. Si `migrate` propose de créer `database/database.sqlite`, appuyez sur Entrée pour répondre `yes`. `npm install` recrée `node_modules/`, le dossier des paquets JavaScript, dont Vite et Tailwind.

Lancez ensuite cette commande dans un terminal, et laissez-le ouvert pendant tout le travail :

```bash
composer run dev
```

Comme au chapitre 13, `composer run dev` lance `npm run dev` : chaque nouvelle classe reçoit sa règle CSS dès que vous enregistrez la vue. Avec Herd, ouvrez `http://tailwind-26.test`. Avec Laragon, rechargez d'abord Laragon depuis son menu, pour qu'il découvre le nouveau dossier, puis ouvrez la

même adresse.

Les pages

Adresse	Vue	Ce qu'elle affiche
`/jeux`	`games/index.blade.php`	le catalogue, avec la couverture de chaque jeu
`/jeux/{id}`	`games/show.blade.php`	la fiche d'un jeu et l'historique de ses prêts
`/jeux/nouveau`	`games/create.blade.php`	le formulaire d'ajout d'un jeu
`/prets`	`loans/index.blade.php`	tous les prêts, dans un tableau
`/composants`	`styleguide.blade.php`	les boutons, les statuts et la carte du site

La page `/composants` rassemble chaque élément de l'interface, dans toutes ses variantes. Une page de ce genre s'appelle un **guide de style**, styleguide en anglais. Vous y vérifiez un composant sans chercher une page qui l'affiche dans le bon état.

Les routes, les contrôleurs et les modèles ne changent pas. Vous ne modifiez que des fichiers de `resources/` : les vues, `css/app.css` et `js/app.js`. Pour le texte, les fonds et les bordures, le projet utilise la teinte `zinc` du chapitre 13, un gris neutre, de `zinc-50` à `zinc-950`.

Lire un exercice

Les exercices 14.1 et 14.2 donnent tout le code, et expliquent chaque classe. Les exercices 14.3 à 14.5 donnent une capture de l'écran fini, une banque de classes et des critères à cocher. Chaque banque contient les classes du corrigé dans le désordre, et trois intrus qui ne servent pas dans l'exercice. Les exercices 14.6 à 14.8 ne donnent que la capture, les critères et un indice caché. L'exercice 14.8 est un bonus, à faire en dehors des séances.

Chaque critère se vérifie à l'écran :

- **une largeur** : ouvrez les outils de développement avec F12, puis le mode d'affichage mobile avec Ctrl+Maj+M, et tapez la largeur voulue,
- **le clavier** : cliquez dans la barre d'adresse, puis avancez d'un lien ou d'un bouton au suivant avec la touche Tab,
- **un attribut** : dans les outils de développement, l'onglet Éléments affiche le HTML de la page tel qu'il est en ce moment, attributs compris.

Le corrigé de chaque exercice se cache sous un bouton. Ouvrez-le après votre essai, et comparez avec votre code. Ses composants ressemblent à ceux du chapitre 13, sans leurs classes `dark:` : le mode sombre arrive avec le bonus 14.8.

2. Pas à pas : le cadre et le bouton

14.1 Brancher Tailwind et poser le cadre (25 minutes)

Charger le CSS. Ouvrez `resources/views/components/layouts/app.blade.php`. Ajoutez deux lignes à la fin du `<head>`, sous `<title>` :

```XML
    <title>{{ $title }} · Ludothèque des Tilleuls</title>
    @fonts
    @vite(['resources/css/app.css', 'resources/js/app.js'])
</head>
```

Rechargez `/jeux`. Avant toute classe, la page a déjà changé. Le titre « Catalogue » a la taille du texte courant, les listes n'ont plus de puces, et les liens ne sont plus ni bleus ni soulignés. `@import 'tailwindcss'` apporte en effet la remise à zéro des styles du navigateur, vue au chapitre 13. Chaque élément ne prend plus que le style de ses classes.

La couleur de marque. La ludothèque a choisi un bleu canard. Ajoutez onze variables de thème dans le bloc `@theme` de `resources/css/app.css`, sous `--font-sans` :

```CSS
@theme {
    --font-sans: 'Instrument Sans', ui-sans-serif, system-ui, sans-serif, 'Apple Color Emoji', 'Segoe UI Emoji',
        'Segoe UI Symbol', 'Noto Color Emoji';

    /* La couleur de la ludothèque : un bleu canard, de 50 (très clair) à 950 (très foncé). */
    --color-brand-50: oklch(0.98 0.016 195);
    --color-brand-100: oklch(0.95 0.04 195);
    --color-brand-200: oklch(0.9 0.07 195);
    --color-brand-300: oklch(0.83 0.1 195);
    --color-brand-400: oklch(0.74 0.12 195);
    --color-brand-500: oklch(0.64 0.12 196);
    --color-brand-600: oklch(0.54 0.105 198);
    --color-brand-700: oklch(0.46 0.09 200);
    --color-brand-800: oklch(0.4 0.075 201);
    --color-brand-900: oklch(0.35 0.06 202);
    --color-brand-950: oklch(0.25 0.045 204);
}
```

Comme au chapitre 13, chaque variable `--color-brand-…` crée toutes les classes de sa nuance : `bg-brand-600`, `text-brand-700`, `outline-brand-600`, et les autres. La fonction CSS `oklch()` reçoit trois nombres : la clarté, de 0 à 1, l'intensité de la couleur, et la teinte, un angle sur le cercle des couleurs. D'une nuance à l'autre, la teinte reste proche de 195 et la clarté baisse. Les couleurs de Tailwind sont écrites de la même façon.

Le corps de la page. Dans le layout, remplacez la balise `<body>`, la balise `<main>` et le pied de page :

```xml
<body class="flex min-h-screen flex-col bg-zinc-50 font-sans text-zinc-900 antialiased">
    …
    <main class="mx-auto w-full max-w-6xl flex-1 px-4 py-8 sm:px-6 sm:py-12 lg:px-8">
        {{ $slot }}
    </main>

    <footer class="border-t border-zinc-200">
        <div class="mx-auto flex max-w-6xl flex-col gap-2 px-4 py-6 text-sm text-zinc-500
sm:flex-row sm:justify-between sm:px-6 lg:px-8">
            <p>Ludothèque des Tilleuls, rue des Tilleuls 12, 1300 Wavre. Ouverte le mercredi
et le samedi, de 10 h à 17 h.</p>
            <p><a href="{{ route('styleguide') }}" class="underline underline-offset-4
hover:text-zinc-900">Les composants du site</a></p>
        </div>
    </footer>
</body>
```

- `flex min-h-screen flex-col` sur `<body>` range l'en-tête, `<main>` et le pied de page en colonne, sur au moins la hauteur de la fenêtre. `flex-1` donne à `<main>` toute la hauteur restante. Le pied de page reste donc en bas de la fenêtre, même sur une page courte.
- `mx-auto max-w-6xl` centre un bloc de 72 rem de large au plus. Dans une colonne Flex, un enfant aux marges `auto` ne prend que la largeur de son contenu : `w-full` lui rend toute la largeur, que `max-w-6xl` limite ensuite.
- `px-4 sm:px-6 lg:px-8` élargit la marge intérieure avec l'écran : 16 pixels sur un téléphone, 24 à partir de 640 pixels, 32 à partir de 1 024.
- `flex-col sm:flex-row sm:justify-between` place les deux paragraphes du pied de page l'un sous l'autre, puis côte à côte à partir de 640 pixels.
- `font-sans` applique la police Instrument Sans de `@theme`, et `antialiased` affine le dessin des lettres sur macOS.

L'en-tête. Remplacez la balise `<header>` et le lien du logo. Gardez la balise `<nav>` telle quelle :

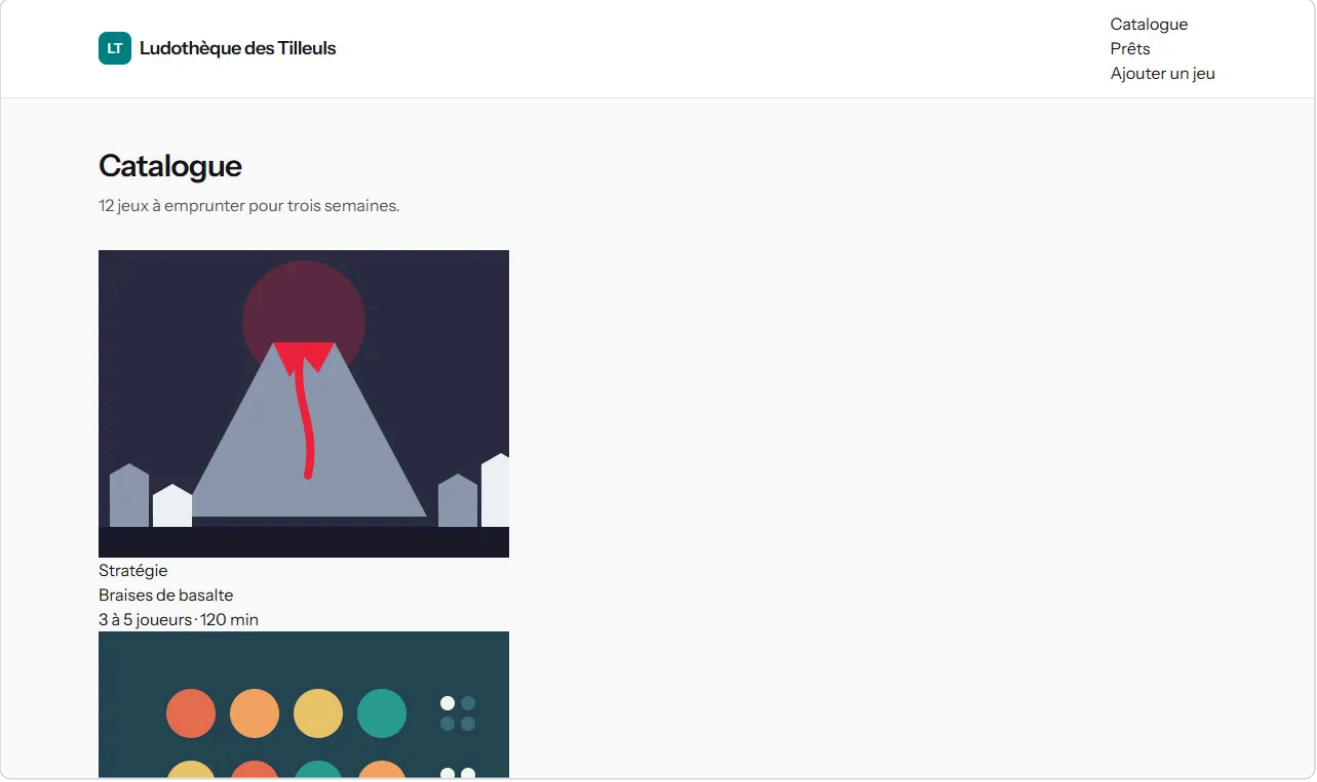
```xml
<header class="border-b border-zinc-200 bg-white">
    <div class="mx-auto flex max-w-6xl flex-wrap items-center justify-between gap-x-6 gap-y-
2 px-4 py-3 sm:px-6 lg:px-8">
        <a href="{{ route('games.index') }}" class="flex items-center gap-2 rounded-md text-
lg font-semibold tracking-tight focus-visible:outline-2 focus-visible:outline-offset-4
focus-visible:outline-brand-600">
            <span class="grid size-8 place-items-center rounded-lg bg-brand-600 text-sm
font-semibold text-white" aria-hidden="true">LT</span>
            Ludothèque des Tilleuls
        </a>
        <nav aria-label="Navigation principale">
            …
        </nav>
    </div>
</header>
```

Le `<div>` de l'en-tête reprend `mx-auto max-w-6xl` et les mêmes marges que `<main>` et le pied de page : le logo, le titre et l'adresse commencent à la même distance du bord. `grid size-8 place-items-center` dessine un carré de 32 pixels, dont `place-items-center` centre le contenu. `aria-hidden="true"` cache « LT » aux lecteurs d'écran, qui ne lisent que le nom de la ludothèque.

Le titre de chaque page. Dans `games/index.blade.php`, entourez le titre et sa phrase d'un `<div>` :

```xml
<div class="mb-8">
    <h1 class="text-3xl font-semibold tracking-tight">Catalogue</h1>
    <p class="mt-2 text-zinc-600">{{ $games->count() }} jeux à emprunter pour trois
semaines.</p>
</div>
```

Faites de même dans `loans/index.blade.php`, `games/create.blade.php` et `styleguide.blade.php`. Donnez aussi `text-zinc-600` aux paragraphes « Aucun jeu dans le catalogue. » et « Aucun prêt pour l'instant. ». Les titres sont en `font-semibold`, car le projet télécharge Instrument Sans en 400, 500 et 600, et non en 700.



Le menu reste une liste verticale à droite du logo, et les jeux restent empilés. Ils prennent leur forme aux exercices 14.3 et 14.5.

Critères

- [ ] Le fond de la page est gris très clair, et l'en-tête blanc est séparé du contenu par un trait.
- [ ] À 1 280 pixels, le logo, le titre de la page et l'adresse du pied de page commencent à la même distance du bord gauche.
- [ ] Sur `/composants`, une page courte, le pied de page touche le bas de la fenêtre.

- [ ] À 375 pixels, les deux paragraphes du pied de page sont l'un sous l'autre, et côte à côte à 640 pixels.
- [ ] Au clavier, le logo reçoit un contour bleu canard, écarté du texte.

14.2 Le bouton et ses variantes (20 minutes)

La page `/composants` affiche quatre boutons sans style : enregistrer, annuler, supprimer, et un bouton désactivé. Créez le composant :

```bash
php artisan make:component button --view
```

```
   INFO  View [resources/views/components/button.blade.php] created successfully.
```

La base. Remplacez tout le contenu du fichier créé :

```xml
@props(['variant' => 'primary'])

<button {{ attributes->merge(['type' => 'button'])->class([
    'inline-flex items-center justify-center gap-2 rounded-lg px-4 py-2 text-sm font-
semibold shadow-xs transition-colors',
]) }}>
    {{ slot }}
</button>
```

- `$attributes->merge(['type' => 'button'])` donne au bouton le type `button` par défaut. Sans attribut `type`, un `<button>` placé dans un formulaire l'envoie. Un `type` écrit dans la balise `<x-button>` remplace cette valeur.
- `->class([...])` reçoit un tableau comme `@class`, et ajoute à la fin les classes écrites dans la balise `<x-button>`.
- `inline-flex items-center justify-center gap-2` centre le texte du bouton, et l'écarte d'une icône s'il en reçoit une.
- `transition-colors` fait passer les couleurs d'un état à l'autre en 150 millisecondes, au lieu d'un changement brusque.

Les variantes et les états. Complétez le tableau :

```xml
@props(['variant' ⇒ 'primary'])

<button {{ attributes->merge(['type'⇒'button'])
->class(['inline-flexitems-centerjustify-centergap-2rounded-lgpx-4py-2text-smfont-semiboldsh
adow-xstransition-colors','focus-visible:outline-2focus-visible:outline-offset-2',
'disabled:cursor-not-alloweddisabled:opacity-50',
'bg-brand-600text-whitehover:bg-brand-700focus-visible:outline-brand-600disabled:hover:bg-br
and-600'⇒attributes→merge(['type' ⇒ 'button'])→class([
    'inline-flex items-center justify-center gap-2 rounded-lg px-4 py-2 text-sm font-
semibold shadow-xs transition-colors',
    'focus-visible:outline-2 focus-visible:outline-offset-2',
    'disabled:cursor-not-allowed disabled:opacity-50',
    'bg-brand-600 text-white hover:bg-brand-700 focus-visible:outline-brand-600
disabled:hover:bg-brand-600' ⇒ attributes->merge(['type'⇒'button'])
->class(['inline-flexitems-centerjustify-centergap-2rounded-lgpx-4py-2text-smfont-semiboldsh
adow-xstransition-colors','focus-visible:outline-2focus-visible:outline-offset-2',
'disabled:cursor-not-alloweddisabled:opacity-50',
'bg-brand-600text-whitehover:bg-brand-700focus-visible:outline-brand-600disabled:hover:bg-br
and-600'⇒variant ⩵ 'primary',
    'bg-white text-zinc-900 ring-1 ring-zinc-300 ring-inset hover:bg-zinc-50 focus-
visible:outline-zinc-500 disabled:hover:bg-white' ⇒ variant⩵'secondary',
'bg-red-600text-whitehover:bg-red-700focus-visible:outline-red-600disabled:hover:bg-red-600'
⇒variant ⩵ 'secondary',
    'bg-red-600 text-white hover:bg-red-700 focus-visible:outline-red-600 disabled:hover:bg-
red-600' ⇒ variant⩵'secondary',
'bg-red-600text-whitehover:bg-red-700focus-visible:outline-red-600disabled:hover:bg-red-600'
⇒variant ⩵ 'danger',
]) }}>
    {{ $slot }}
</button>
```

Ligne	Ce qu'elle fait
`focus-visible:outline-2 focus-visible:outline-offset-2`	au focus du clavier, un contour de 2 pixels, écarté du bouton de 2 pixels
`disabled:cursor-not-allowed disabled:opacity-50`	un bouton désactivé devient à moitié transparent, et le curseur montre un sens interdit
`focus-visible:outline-brand-600`, `-zinc-500`, `-red-600`	la couleur du contour, une par variante
`ring-1 ring-zinc-300 ring-inset`	le bord du bouton secondaire : un contour de 1 pixel, dessiné à l'intérieur du bouton
`disabled:hover:bg-brand-600`	au survol d'un bouton désactivé, le fond reste celui du repos

Les préfixes s'additionnent, comme `md:hover:` au chapitre 13. Sans `disabled:hover:bg-brand-600`, un bouton désactivé foncerait au survol, comme s'il répondait au clic.

Le guide de style. Dans `styleguide.blade.php`, sous le `<div>` du titre, regroupez les trois sections dans un `<div>`, et remplacez les quatre boutons :

```xml
<div class="space-y-10">
    <section>
        <h2 class="text-xl font-semibold tracking-tight">Boutons</h2>
        <p class="mt-4 flex flex-wrap gap-3">
            <x-button>Enregistrer</x-button>
            <x-button variant="secondary">Annuler</x-button>
            <x-button variant="danger">Supprimer</x-button>
            <x-button disabled>Indisponible</x-button>
        </p>
    </section>

    …
</div>
```

Donnez les classes du titre « Boutons » aux deux autres titres `<h2>`. Affichez ensuite le code source de la page avec Ctrl+U :

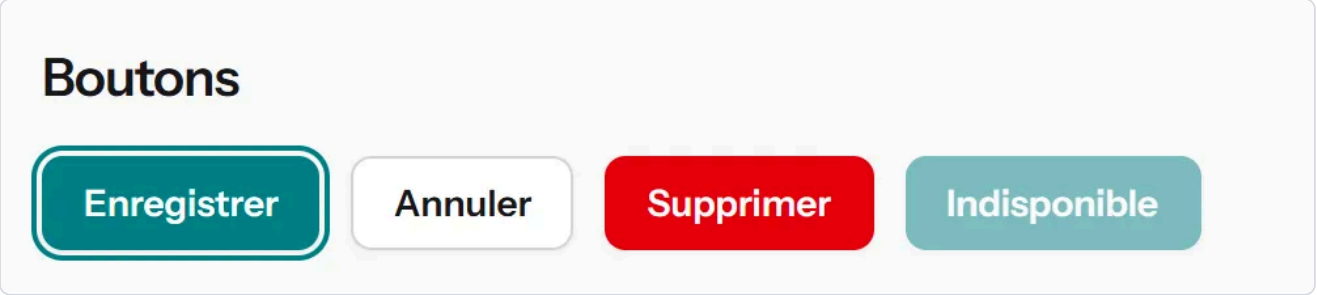
```
            <button class="inline-flex items-center justify-center gap-2 rounded-lg px-4 py-
2 text-sm font-semibold shadow-xs transition-colors focus-visible:outline-2 focus-
visible:outline-offset-2 disabled:cursor-not-allowed disabled:opacity-50 bg-brand-600 text-
white hover:bg-brand-700 focus-visible:outline-brand-600 disabled:hover:bg-brand-600"
type="button">
    Enregistrer
</button>
…
            <button class="inline-flex items-center justify-center gap-2 rounded-lg px-4 py-
2 text-sm font-semibold shadow-xs transition-colors focus-visible:outline-2 focus-
visible:outline-offset-2 disabled:cursor-not-allowed disabled:opacity-50 bg-brand-600 text-
white hover:bg-brand-700 focus-visible:outline-brand-600 disabled:hover:bg-brand-600"
type="button" disabled="disabled">
    Indisponible
</button>
```

`$attributes->class()` a gardé les trois premières chaînes et celle de la variante choisie. Le quatrième bouton n'a pas d'attribut `variant` : il prend `primary`, la valeur par défaut de `@props`.

Le formulaire. Dans `games/create.blade.php`, remplacez le bouton d'envoi :

```xml
<x-button type="submit">Ajouter le jeu</x-button>
```

Le `type="submit"` écrit dans la balise remplace le `type="button"` de `merge()`, et le bouton envoie toujours le formulaire.



Critères, sur `/composants` :

- [ ] Les quatre boutons sont bleu canard, blanc bordé de gris, rouge, et bleu canard pâle.
- [ ] Au survol, les trois premiers foncent progressivement. Le quatrième ne change pas, et le curseur montre un sens interdit.
- [ ] Au clavier, chaque bouton actif reçoit un contour de sa couleur, écarté du bouton. Le bouton désactivé ne reçoit pas le focus.
- [ ] Dans le code source, les quatre boutons portent `type="button"`, et celui de `/jeux/nouveau` porte `type="submit"`.
- [ ] Le formulaire de `/jeux/nouveau` s'envoie toujours : un clic sur « Ajouter le jeu », avec les champs remplis, affiche la fiche du nouveau jeu.

3. Avec une banque de classes : la carte, le tableau, le menu

14.3 La carte et la grille du catalogue (25 minutes)

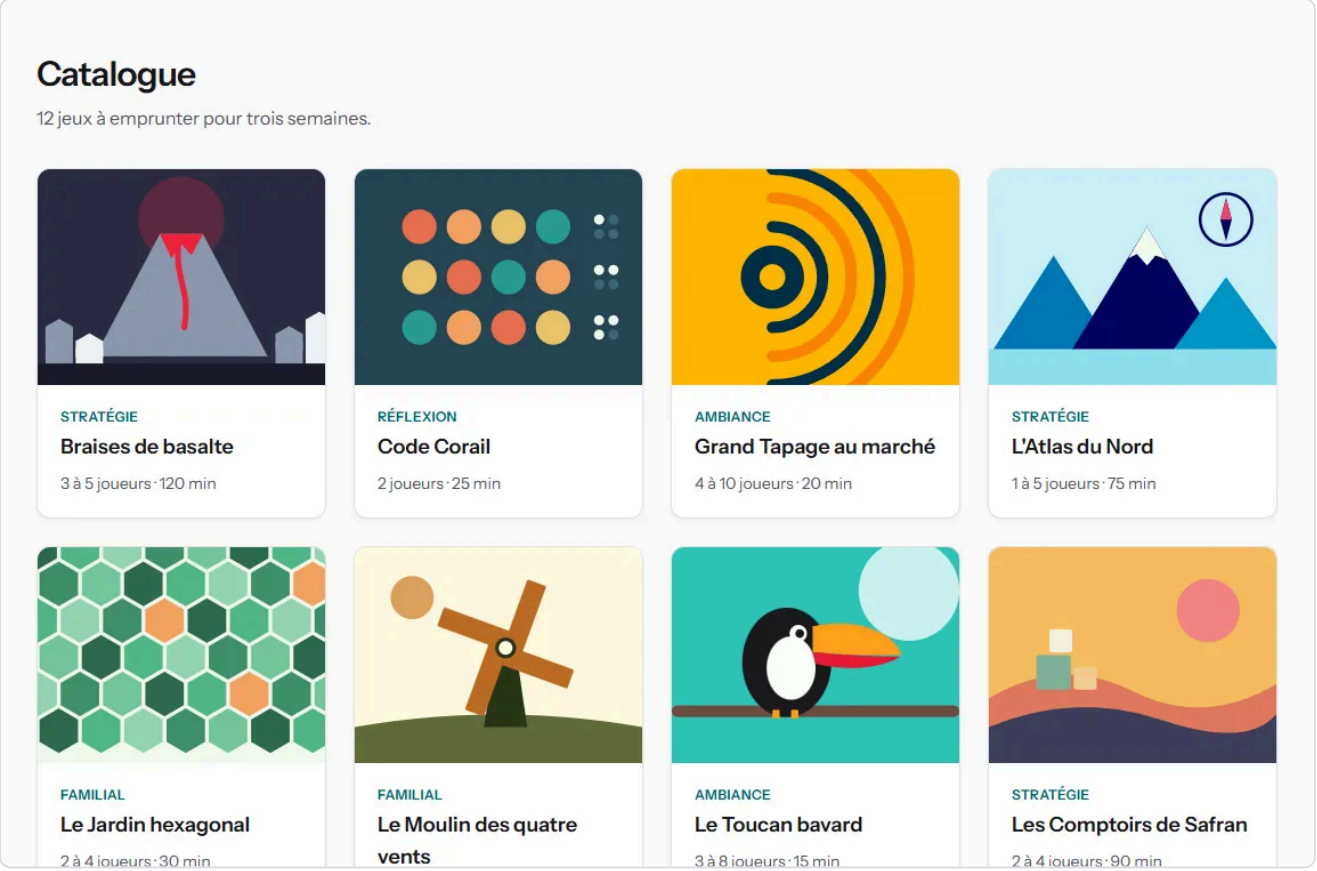
Chaque jeu du catalogue s'affiche dans une carte : la couverture en haut, puis la catégorie, le titre et une ligne d'informations. La carte du guide de style a le même cadre, sans image. Créez le composant `card` avec `php artisan make:component card --view`, et partez de cette structure :

```xml
<article {{ attributes->class([]) }}>
    @isset(image)
        {{ image }}
    @endisset

    <div>
        {{ slot }}
    </div>
</article>
```

L'image arrive par un slot nommé, `image`, comme `<x-slot:note>` au chapitre 9. `@isset` ne l'affiche que si la page en donne une. Dans `games/index.blade.php`, chaque `<article>` devient :

```xml
<x-card>
    <x-slot:image>
        <img src="{{ asset(game->coverPath()) }}" alt="" width="400" height="300">
    </x-slot:image>

    <p>{{ game->category }}</p>
    <h2><a href="{{ route('games.show', game->id) }}">{{ game->title }}</a></h2>
    <p>{{ game->playersLabel() }} · {{ game->duration }} min</p>
</x-card>
```

Dans `styleguide.blade.php`, remplacez de même l'`<article>` de la section « Carte » par `<x-card>`, sans slot `image`. Ajoutez ensuite les classes de la banque : celles du cadre dans le composant, celles de la grille sur la liste `<ul>`, et celles du contenu dans les deux vues.



Banque de classes, trois intrus compris :

Classe	Effet
`object-cover`	recadre l'image pour remplir sa boîte, sans la déformer
`sm:grid-cols-2`	deux colonnes à partir de 640 pixels
`tracking-wide`	lettres un peu plus espacées
`h-full`	toute la hauteur du parent
`whitespace-nowrap`	interdit le passage à la ligne
`rounded-xl`	coins arrondis de 12 pixels
`xl:grid-cols-4`	quatre colonnes à partir de 1 280 pixels
`text-zinc-600`	texte gris moyen
`p-5`	marge intérieure de 20 pixels

Classe	Effet
`uppercase`	tout en majuscules
`hover:shadow-md`	ombre plus marquée au survol
`aspect-4/3`	proportions de 4 de large pour 3 de haut
`max-w-sm`	largeur maximale de 24 rem
`grid`	range les enfants dans une grille
`tabular-nums`	chiffres de même largeur
`border border-zinc-200`	bordure de 1 pixel, gris clair
`transition-shadow`	l'ombre change progressivement
`text-brand-700`	texte bleu canard foncé
`lg:grid-cols-3`	trois colonnes à partir de 1 024 pixels
`overflow-hidden`	coupe ce qui dépasse du cadre, comme les coins de l'image
`hover:underline hover:underline-offset-4`	souligne au survol, 4 pixels sous le texte
`gap-6`	24 pixels entre les lignes et entre les colonnes
`shadow-xs`	ombre très légère
`divide-y`	trait horizontal entre les enfants
`w-full`	toute la largeur du parent
`rounded-sm`	coins arrondis de 4 pixels, ceux du contour de focus compris
`bg-white`	fond blanc
`text-xs`, `text-sm`, `text-lg`	texte très petit, petit, grand
`font-semibold`	lettres épaisses
`mt-1`, `mt-2`, `mt-4`	marge au-dessus de 4, 8 ou 16 pixels
`focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-brand-600`	le contour de focus des boutons principaux, vu en 14.2

Critères

- [ ] Le catalogue affiche quatre cartes par ligne à 1 280 pixels, trois à 1 024, deux à 640 et une seule à 375.
- [ ] Dans une même ligne, les cartes ont toutes la même hauteur, même quand un titre passe sur deux lignes.

- [ ] La couverture remplit le haut de la carte sans être déformée, et ses coins suivent l'arrondi de la carte.
- [ ] Au survol d'une carte, son ombre se renforce progressivement. Au survol du titre, il se souligne.
- [ ] Au clavier, le titre de chaque jeu reçoit un contour bleu canard.
- [ ] La catégorie s'écrit en petites majuscules bleu canard, au-dessus du titre.
- [ ] Sur `/composants`, la carte a le même cadre, sur 24 rem de large au plus.
- [ ] Les classes du cadre n'apparaissent que dans `components/card.blade.php`.

components/card.blade.php :

```XML
<article {{ attributes->class([
    'overflow-hidden rounded-xl border border-zinc-200 bg-white shadow-xs',
]) }}>
    @isset(image)
        {{ image }}
    @endisset

    <div class="p-5">
        {{ slot }}
    </div>
</article>
```

games/index.blade.php , sous le titre de la page :

```XML
<ul class="grid gap-6 sm:grid-cols-2 lg:grid-cols-3 xl:grid-cols-4">
    @foreach (gamesasgames as gamesasgame)
        <li>
            <x-card class="h-full transition-shadow hover:shadow-md">
                <x-slot:image>
                    <img src="{{ asset(game->coverPath()) }}" alt="" width="400"
height="300" class="aspect-4/3 w-full object-cover">
                </x-slot:image>

                <p class="text-xs font-semibold tracking-wide text-brand-700
uppercase">{{ game->category }}</p>
                <h2 class="mt-1 text-lg font-semibold">
                    <a href="{{ route('games.show', game->id) }}" class="rounded-sm
hover:underline hover:underline-offset-4 focus-visible:outline-2 focus-visible:outline-
offset-2 focus-visible:outline-brand-600">{{ game->title }}</a>
                </h2>
                <p class="mt-2 text-sm text-zinc-600">{{ game->playersLabel() }} · {{
game->duration }} min</p>
            </x-card>
        </li>
    @endforeach
</ul>
```

styleguide.blade.php , section « Carte » :

```XML
<x-card class="mt-4 max-w-sm">
    <h3 class="font-semibold">Titre de la carte</h3>
    <p class="mt-2 text-sm text-zinc-600">Le contenu de la carte : un texte court, une
liste ou une image.</p>
</x-card>
```

La grille étire chaque `<li>` à la hauteur de sa ligne. `h-full` étire ensuite la carte à la hauteur de son `<li>` : sans lui, une carte au titre court s'arrête plus haut que ses voisines. `overflow-hidden` coupe les coins de l'image, qui sinon dépasseraient de l'arrondi de la carte.

Les trois intrus : `whitespace-nowrap`, `tabular-nums` et `divide-y`. Ils servent à l'exercice suivant.

14.4 Le tableau des prêts et le badge de statut (25 minutes)

Chaque prêt est en cours, en retard ou rendu. Le modèle `Loan` calcule ce statut à partir des dates :

Méthode	Ce qu'elle renvoie
`$loan->status()`	`'ongoing'`, `'overdue'` ou `'returned'`
`$loan->statusLabel()`	`'En cours'`, `'En retard'` ou `'Rendu'`

Le badge. Créez le composant `badge`, avec une prop `status`. Écrivez ses classes avec `@class`, sur le modèle du badge du chapitre 13 : une chaîne toujours écrite, puis une chaîne par statut. Comme au chapitre 13, le texte du badge arrive par le slot, et `statusLabel()` le fournit :

```xml
<x-badge :status="$loan->status()">{{ $loan->statusLabel() }}</x-badge>
```

Le badge ne choisit donc que ses couleurs. Dans `styleguide.blade.php`, remplacez les trois `<span>` par `<x-badge status="ongoing">En cours</x-badge>` et ses deux voisins.

Le tableau. Un tableau de six colonnes ne tient pas sur un téléphone de 375 pixels. Dans `loans/index.blade.php`, entourez `<table>` d'un `<div>` : c'est ce `<div>` qui défilera de côté, et non toute la page. Donnez ensuite leurs classes au `<div>`, à `<table>`, à `<thead>`, à `<tbody>`, aux lignes et aux cellules. Remplacez enfin `{{ $loan->statusLabel() }}` par le badge.

Prêts

15 prêts, du plus récent au plus ancien.

Jeu	Membre	Emprunté le	À rendre le	Rendu le	Statut
Le Toucan bavard	Yasmine El Idrissi yasmine.elidrissi@exemple.be	04/10/2026	25/10/2026	—	En cours
Code Corail	Thomas Dubois thomas.dubois@exemple.be	02/10/2026	23/10/2026	—	En cours
Mille Pétales	Mei Lin Chen meilin.chen@exemple.be	29/09/2026	20/10/2026	—	En cours
Les Sept Lanternes	Kofi Mensah kofi.mensah@exemple.be	25/09/2026	16/10/2026	—	En cours
Marée d'étoiles	Sofia Rossi sofia.rossi@exemple.be	20/09/2026	11/10/2026	—	En cours
Les Comptoirs de Safran	Thomas Dubois thomas.dubois@exemple.be	15/09/2026	06/10/2026	27/09/2026	Rendu
L'Atlas du Nord	Amina Benali amina.benali@exemple.be	12/09/2026	03/10/2026	—	En retard

Prêts

15 prêts, du plus récent au plus ancien.

Jeu	Membre
Le Toucan bavard	Yasmine El Idrissi yasmine.elidrissi@e
Code Corail	Thomas Dubois thomas.dubois@ex
Mille Pétales	Mei Lin Chen meilin.chen@exem
Les Sept Lanternes	Kofi Mensah kofi.mensah@exen
Marée d'étoiles	Sofia Rossi sofia.rossi@exemp

Banque de classes, trois intrus compris :

Classe	Effet
`whitespace-nowrap`	interdit le passage à la ligne
`bg-red-50 text-red-700 ring-red-600/20`	fond rouge très pâle, texte rouge, contour rouge à 20 % d'opacité
`min-w-full`	au moins toute la largeur du parent
`px-4 py-3`	marge intérieure de 16 pixels sur les côtés, 12 en haut et en bas
`inline-flex items-center`	aligne le contenu au milieu, sans passer à la ligne avant ni après
`tabular-nums`	chiffres de même largeur, alignés d'une ligne à l'autre
`aspect-4/3`	proportions de 4 de large pour 3 de haut
`overflow-x-auto`	ajoute une barre de défilement horizontale quand le contenu est trop large
`ring-1 ring-inset`	contour de 1 pixel, dessiné à l'intérieur
`hover:bg-zinc-50`	fond gris très clair au survol
`divide-y divide-zinc-200`	trait gris clair entre les enfants
`text-left`	texte aligné à gauche
`bg-zinc-100 text-zinc-600 ring-zinc-500/20`	fond gris pâle, texte gris, contour gris à 20 % d'opacité
`rounded-full`	coins entièrement arrondis, en forme de pilule
`font-medium`	lettres moyennement épaisses
`object-cover`	recadre l'image pour remplir sa boîte
`bg-zinc-50`	fond gris très clair
`px-2.5 py-0.5`	marge intérieure de 10 pixels sur les côtés, 2 en haut et en bas
`text-zinc-500`	texte gris
`bg-sky-50 text-sky-700 ring-sky-600/20`	fond bleu ciel très pâle, texte bleu, contour bleu à 20 % d'opacité
`xl:grid-cols-4`	quatre colonnes à partir de 1 280 pixels
`text-xs`, `text-sm`	texte très petit, petit
`font-semibold`	lettres épaisses
`mt-4 flex flex-wrap gap-3`	la rangée de boutons du guide de style, vue en 14.2

Classe	Effet
`rounded-xl border border-zinc-200 bg-white shadow-xs`	le cadre de la carte, vu en 14.3
`text-brand-700`, avec les classes du lien de la carte	le lien du titre, vu en 14.3, en bleu canard

Critères

- [ ] À 375 pixels, la page ne défile pas de côté. Seul le tableau défile, à l'intérieur de son cadre arrondi.
- [ ] Le nom et l'adresse du membre restent sur deux lignes, et aucune autre cellule ne passe à la ligne.
- [ ] Les en-têtes sont en gras sur un fond gris très clair, et un trait fin sépare les lignes.
- [ ] Au survol, toute la ligne prend un fond gris très clair.
- [ ] Dans les colonnes de dates, les chiffres sont alignés d'une ligne à l'autre.
- [ ] Le nom du jeu est un lien bleu canard, souligné au survol, avec un contour au clavier.
- [ ] Un prêt en cours a un badge bleu, un retard un badge rouge, un jeu rendu un badge gris, ici comme sur `/composants`.
- [ ] `loans/index.blade.php` ne contient aucune condition sur le statut : seul `badge.blade.php` choisit les couleurs.

components/badge.blade.php :

```xml
@props(['status'])

<span @class([
    'inline-flex items-center rounded-full px-2.5 py-0.5 text-xs font-medium
whitespace-nowrap ring-1 ring-inset',
    'bg-sky-50 text-sky-700 ring-sky-600/20' => status===’ongoing’,
’bg-red-50text-red-700ring-red-600/20’=>status === 'ongoing',
    'bg-red-50 text-red-700 ring-red-600/20' => status===’ongoing’,
’bg-red-50text-red-700ring-red-600/20’=>status === 'overdue',
    'bg-zinc-100 text-zinc-600 ring-zinc-500/20' => $status === 'returned',
])>
    {{ $slot }}
</span>
```

loans/index.blade.php , sous le titre de la page :

XML

```
<div class="overflow-x-auto rounded-xl border border-zinc-200 bg-white shadow-xs">
    <table class="min-w-full divide-y divide-zinc-200 text-left text-sm">
        <thead class="bg-zinc-50">
            <tr>
                <th scope="col" class="px-4 py-3 font-semibold whitespace-
nowrap">Jeu</th>
                …
                <th scope="col" class="px-4 py-3 font-semibold whitespace-
nowrap">Statut</th>
            </tr>
        </thead>
        <tbody class="divide-y divide-zinc-200">
            @foreach (loansasloans as loansasloan)
                <tr class="hover:bg-zinc-50">
                    <td class="px-4 py-3 font-medium whitespace-nowrap">
                        <a href="{{ route('games.show', loan->game->id) }}"
class="rounded-sm text-brand-700 hover:underline hover:underline-offset-4 focus-
visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-brand-600">{{
loan->game->title }}</a>
                    </td>
                    <td class="px-4 py-3 whitespace-nowrap">
                        {{ loan->member->name }}
                        <br>
                        <span class="text-zinc-500">{{ loan->member->email }}</span>
                    </td>
                    <td class="px-4 py-3 whitespace-nowrap tabular-nums">{{ loan
->borrowed_at->format('d/m/Y') }}</td>
                    <td class="px-4 py-3 whitespace-nowrap tabular-nums">{{ loan
->due_at->format('d/m/Y') }}</td>
                    <td class="px-4 py-3 whitespace-nowrap tabular-nums">
                        @if (loan->returned_at !== null)
                            {{ $loan->returned_at->format('d/m/Y') }}
                        @else
                            <span class="text-zinc-500">—</span>
                        @endif
                    </td>
                    <td class="px-4 py-3">
                        <x-badge :status="loan->status()">{{ $loan->statusLabel() }}
</x-badge>
                    </td>
                </tr>
            @endforeach
        </tbody>
    </table>
</div>
```

Les six `<th>` portent les mêmes classes. `styleguide.blade.php`, section « Statuts d'un prêt » :

```xml
<p class="mt-4 flex flex-wrap gap-3">
    <x-badge status="ongoing">En cours</x-badge>
    <x-badge status="overdue">En retard</x-badge>
    <x-badge status="returned">Rendu</x-badge>
</p>
```

`min-w-full` donne au tableau au moins la largeur du cadre sur un grand écran. Sur un téléphone, `whitespace-nowrap` garde chaque cellule sur une ligne : le tableau devient plus large que le cadre, et `overflow-x-auto` le fait défiler. Les intrus : `aspect-4/3`, `object-cover` et `xl:grid-cols-4`.

14.5 Le menu responsive (30 minutes)

Sur un téléphone, le logo et les trois liens du menu ne tiennent pas sur une ligne. Le menu se cache alors derrière un bouton, et s'affiche en ligne à partir de 768 pixels. Le lien de la page ouverte se distingue des deux autres.

Le lien de menu. Créez le composant `nav-link`, avec deux props : `href`, l'adresse, et `active`, qui vaut `false` par défaut. Partez de cette structure, et ajoutez ses classes avec `@class` : une chaîne toujours écrite, une pour le lien actif, une pour les autres. `! $active` vaut `true` quand le lien n'est pas actif.

```xml
@props(['href', 'active' => false])

<a
    href="{{ href }}"
    @if (active) aria-current="page" @endif
>
    {{ $slot }}
</a>
```

L'attribut `aria-current="page"` indique aux lecteurs d'écran que ce lien mène à la page ouverte. Dans le layout, les trois liens deviennent :

```xml
<li><x-nav-link :href="route('games.index')" :active="request()->routeIs('games.index',
'games.show')">Catalogue</x-nav-link></li>
<li><x-nav-link :href="route('loans.index')" :active="request()
->routeIs('loans.*')">Prêts</x-nav-link></li>
<li><x-nav-link :href="route('games.create')" :active="request()
->routeIs('games.create')">Ajouter un jeu</x-nav-link></li>
```

`request()->routeIs()` renvoie `true` quand le nom de la route de la page ouverte figure dans sa liste. `'loans.*'` désigne toute route dont le nom commence par `loans.`. Le lien « Catalogue » reste actif sur la fiche d'un jeu, car elle appartient au catalogue.

Le bouton. Placez ce bouton entre le logo et `<nav>`, et donnez à `<nav>` l'attribut `id="main-menu"` :

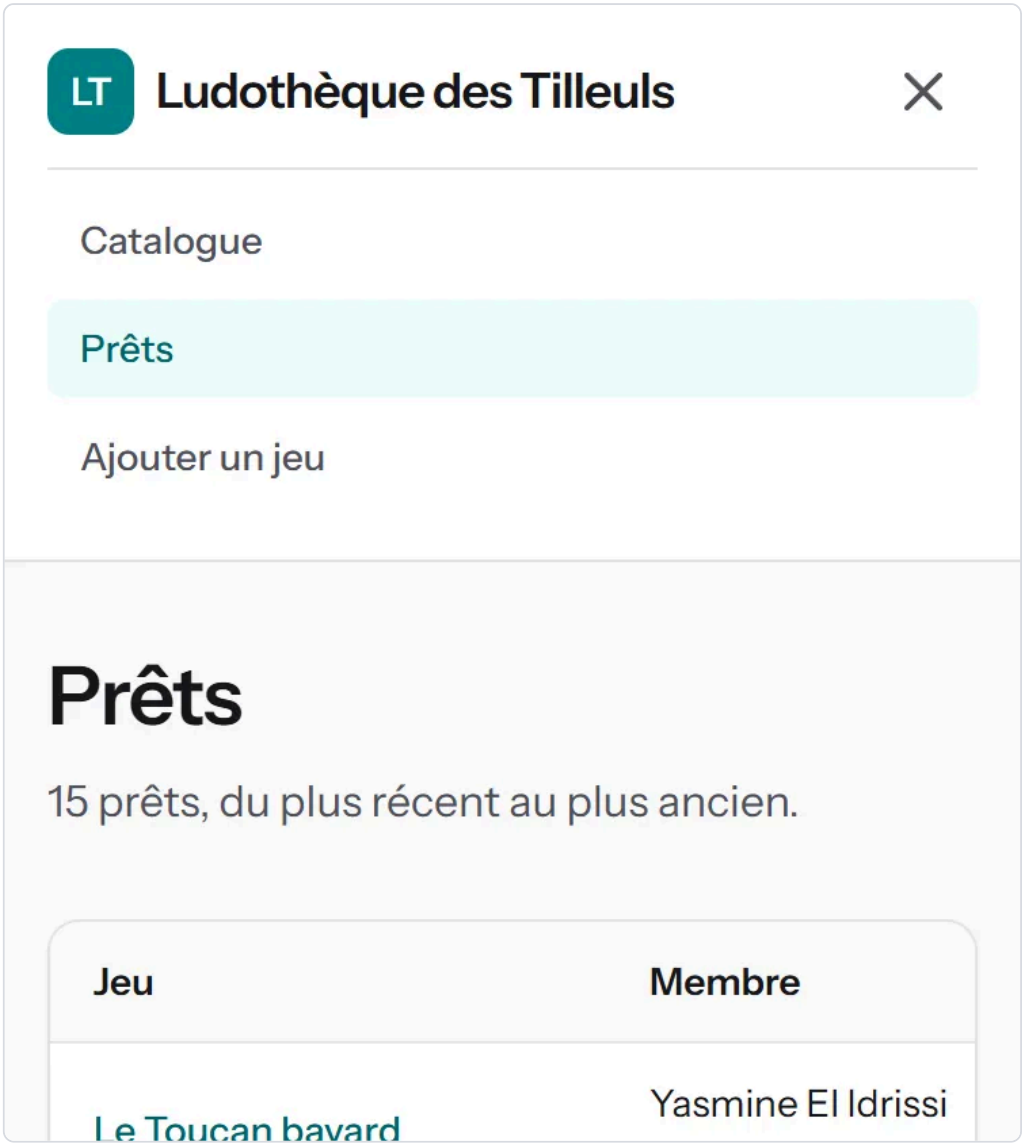
```xml
<button type="button" id="menu-button" aria-controls="main-menu" aria-expanded="false">
    <span class="sr-only">Menu</span>
    <svg class="size-6 group-aria-expanded:hidden" viewBox="0 0 24 24" fill="none"
stroke="currentColor" stroke-width="2" stroke-linecap="round" aria-hidden="true">
        <path d="M4 6h16M4 12h16M4 18h16" />
    </svg>
    <svg class="hidden size-6 group-aria-expanded:block" viewBox="0 0 24 24" fill="none"
stroke="currentColor" stroke-width="2" stroke-linecap="round" aria-hidden="true">
        <path d="M6 6l12 12M18 6L6 18" />
    </svg>
</button>
```

- **Les deux** `<svg>` dessinent trois traits, l'icône appelée burger, et une croix. `stroke="currentColor"` leur donne la couleur du texte du bouton.
- `sr-only` cache le mot « Menu » à l'écran, mais les lecteurs d'écran le lisent : le bouton a un nom.
- `group-aria-expanded:hidden` cache les trois traits quand le parent marqué de la classe `group` porte `aria-expanded="true"`. `group-aria-expanded:block` affiche alors la croix. La classe `group`, dans la banque, va sur le bouton.

L'icône suit donc l'attribut `aria-expanded`, sans autre ligne de JavaScript.

Le script. Le code d'une page Laravel va dans `resources/js/app.js`, la seconde source de `@vite`. Le layout ne contient aucune balise `<script>` : Vite sert `app.js` pendant `npm run dev`, et l'inclut dans `public/build/` au build. Recopiez-y le script du chapitre 13, section 6, puis vérifiez ces deux points :

- le script lit l'état du menu dans `aria-expanded`, avec `getAttribute()`, puis écrit l'état contraire,
- il ne s'exécute que si `querySelector()` a trouvé le bouton et le menu, car `querySelector()` renvoie `null` pour un élément absent.



Banque de classes, trois intrus compris :

Classe	Effet
`md:hidden`	cache l'élément à partir de 768 pixels
`group`	marque le parent que lisent les préfixes `group-…:` de ses enfants
`border-t border-zinc-200`	trait gris clair au-dessus
`transition-colors`	les couleurs changent progressivement
`md:flex-row`	enfants en ligne à partir de 768 pixels
`bg-brand-50 text-brand-700`	fond bleu canard très clair, texte bleu canard foncé
`tabular-nums`	chiffres de même largeur

Classe	Effet
`hidden`	cache l'élément
`md:w-auto`	largeur du contenu à partir de 768 pixels
`p-2`	marge intérieure de 8 pixels
`md:block`	affiche l'élément à partir de 768 pixels
`flex flex-col`	enfants en colonne
`hover:bg-zinc-100 hover:text-zinc-900`	fond gris pâle et texte presque noir au survol
`md:border-0`	plus de bordure à partir de 768 pixels
`block`	le lien occupe toute la largeur de sa ligne, et toute sa surface est cliquable
`ring-inset`	contour dessiné à l'intérieur
`gap-1`	4 pixels entre les enfants
`md:items-center`	enfants centrés en hauteur à partir de 768 pixels
`w-full`	toute la largeur du parent
`px-3 py-2`	marge intérieure de 12 pixels sur les côtés, 8 en haut et en bas
`py-2`, `md:py-0`	marge intérieure de 8 pixels en haut et en bas, aucune à partir de 768 pixels
`aspect-4/3`	proportions de 4 de large pour 3 de haut
`rounded-md`	coins arrondis de 6 pixels
`text-zinc-600`	texte gris moyen
`text-sm font-medium`	texte petit, lettres moyennement épaisses
`focus-visible:outline-2 focus-visible:outline-brand-600`	contour de focus bleu canard, collé à l'élément

Critères

- [ ] À 375 pixels, l'en-tête n'affiche que le logo et l'icône à trois traits.
- [ ] Un clic sur l'icône affiche les trois liens l'un sous l'autre, sous un trait, et l'icône devient une croix. Un second clic referme le menu.
- [ ] Dans l'onglet Éléments, le bouton passe de `aria-expanded="false"` à `aria-expanded="true"` quand le menu s'ouvre.
- [ ] À partir de 768 pixels, le bouton disparaît, et les trois liens s'alignent à droite du logo, menu fermé ou non.
- [ ] Le lien de la page ouverte a un fond bleu canard très clair et porte `aria-current="page"`. Sur la fiche d'un jeu, c'est « Catalogue ».

- [ ] Au clavier, le bouton et chaque lien reçoivent un contour bleu canard.
- [ ] Le layout ne contient aucune balise `<script>`.

components/nav-link.blade.php :

```XML
@props(['href', 'active' => false])

<a
    href="{{ href }}"
    @if (active) aria-current="page" @endif
    @class([
        'block rounded-md px-3 py-2 text-sm font-medium transition-colors focus-visible:outline-2 focus-visible:outline-brand-600',
        'bg-brand-50 text-brand-700' => active,
’text-zinc-600hover:bg-zinc-100hover:text-zinc-900’⇒!active,
        'text-zinc-600 hover:bg-zinc-100 hover:text-zinc-900' => ! active,
’text-zinc-600hover:bg-zinc-100hover:text-zinc-900’⇒!active,
    ])
>
    {{ $slot }}
</a>
```

Le layout, entre le logo et la fin du `<div>` de l'en-tête :

```XML
<button
    type="button"
    id="menu-button"
    class="group rounded-md p-2 text-zinc-600 hover:bg-zinc-100 hover:text-zinc-900 focus-visible:outline-2 focus-visible:outline-brand-600 md:hidden"
    aria-controls="main-menu"
    aria-expanded="false"
>
    …
</button>

<nav id="main-menu" class="hidden w-full md:block md:w-auto" aria-label="Navigation principale">
    <ul class="flex flex-col gap-1 border-t border-zinc-200 py-2 md:flex-row md:items-center md:border-0 md:py-0">
        …
    </ul>
</nav>
```

resources/js/app.js :

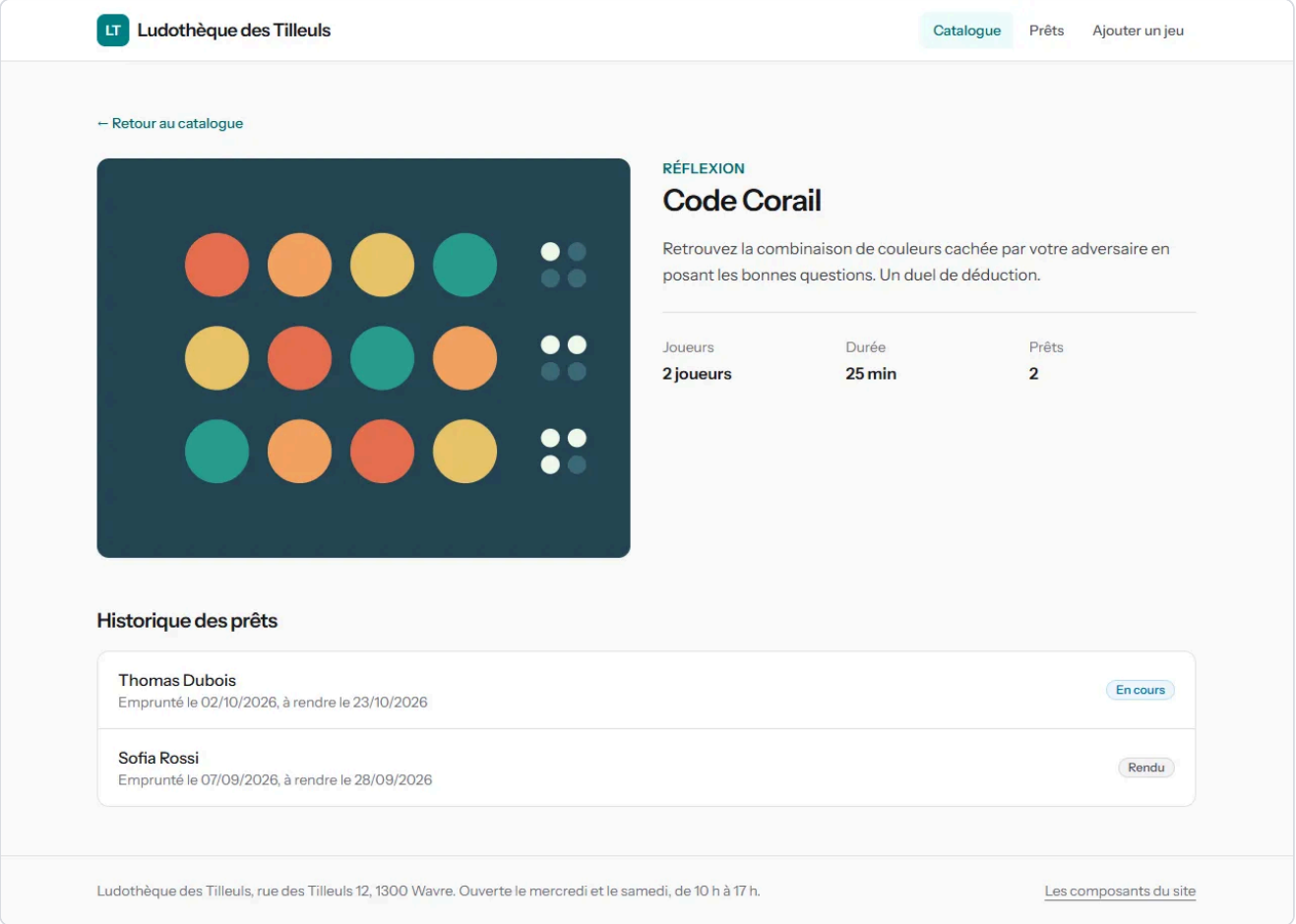
```javascript
// Le menu mobile : le bouton burger affiche ou cache la liste des liens.
const menuButton = document.querySelector('#menu-button');
const menu = document.querySelector('#main-menu');

if (menuButton && menu) {
    menuButton.addEventListener('click', () => {
        const isOpen = menuButton.getAttribute('aria-expanded') === 'true';

        menuButton.setAttribute('aria-expanded', String(!isOpen));
        menu.classList.toggle('hidden');
    });
}
```

`w-full` place le menu ouvert sur sa propre ligne, sous le logo : le `<div>` de l'en-tête a `flex-wrap`. À partir de 768 pixels, `md:block` l'emporte sur `hidden`, que le script l'ait retiré ou non. Les intrus : `tabular-nums`, `ring-inset` et `aspect-4/3`.

4. D'après la capture : la fiche, le formulaire, le mode sombre

14.6 La fiche d'un jeu (20 minutes)

Mettez en forme `games/show.blade.php`. Dans chaque `<li>` de l'historique, regroupez d'abord le nom du membre et la phrase des dates dans un `<div>`. Remplacez ensuite le paragraphe du statut par `<x-badge>`, comme en 14.4. La ligne n'a plus que deux enfants, le `<div>` et le badge, et le badge peut se placer à droite.



Critères

- [ ] Le lien « ← Retour au catalogue » est petit, bleu canard, souligné au survol, avec un contour au clavier.
- [ ] À partir de 768 pixels, la couverture et le texte sont côte à côte, en deux colonnes égales, alignés en haut. En dessous, la couverture est au-dessus du texte.
- [ ] La couverture a des coins arrondis et une ombre légère, sans être déformée.
- [ ] La catégorie et le titre reprennent le style du catalogue. Le titre a la taille des autres titres de page.
- [ ] La description a des lignes plus espacées que le texte courant.
- [ ] Joueurs, durée et nombre de prêts sont sur trois colonnes, sous un trait. Chaque libellé gris est au-dessus de sa valeur en gras.
- [ ] L'historique est une liste encadrée, avec un trait entre deux prêts et le badge à droite. Si la place manque, le badge passe sous la date.

`md:grid-cols-2`, `md:items-start`, `grid-cols-3`, `leading-relaxed` et `text-zinc-700`.

Les espacements `mb-6`, `mt-6`, `mt-12`, `pt-6`, `gap-4`, `gap-8`, `px-5` et `py-4` suivent l'échelle de `mt-4` et de `gap-6` : un pas vaut 4 pixels, et `mt-12` laisse donc 48 pixels. Toutes les autres classes viennent des exercices 14.1 à 14.5.

XML

```
<x-layouts.app>
    <x-slot:title>{{ game->title }}</x-slot:title>

    <p class="mb-6">
        <a href="{{ route('games.index') }}" class="rounded-sm text-sm font-medium
text-brand-700 hover:underline hover:underline-offset-4 focus-visible:outline-2 focus-
visible:outline-offset-2 focus-visible:outline-brand-600">← Retour au catalogue</a>
    </p>

    <article class="grid gap-8 md:grid-cols-2 md:items-start">
        <img src="{{ asset(game->coverPath()) }}" alt="" width="400" height="300"
class="aspect-4/3 w-full rounded-xl object-cover shadow-xs">

        <div>
            <p class="text-sm font-semibold tracking-wide text-brand-700 uppercase">{{
game->category }}</p>
            <h1 class="mt-1 text-3xl font-semibold tracking-tight">{{ game->title }}
</h1>

            @if (game->description !== null)
                <p class="mt-4 leading-relaxed text-zinc-700">{{ $game->description }}
</p>
            @endif

            <dl class="mt-6 grid grid-cols-3 gap-4 border-t border-zinc-200 pt-6">
                <div>
                    <dt class="text-sm text-zinc-500">Joueurs</dt>
                    <dd class="mt-1 font-semibold">{{ $game->playersLabel() }}</dd>
                </div>
                …
            </dl>
        </div>
    </article>

    <section class="mt-12">
        <h2 class="text-xl font-semibold tracking-tight">Historique des prêts</h2>

        @if (loans->isEmpty())
            <p class="mt-4 text-zinc-600">Ce jeu n'a encore jamais été emprunté.</p>
        @else
            <ul class="mt-4 divide-y divide-zinc-200 rounded-xl border border-zinc-200
bg-white">
                @foreach (loansasloans as loansasloan)
                    <li class="flex flex-wrap items-center justify-between gap-x-6 gap-
y-2 px-5 py-4">
                        <div>
                            <p class="font-medium">{{ loan->member->name }}</p>
                            <p class="text-sm text-zinc-500">Emprunté le {{ loan
->borrowed_at->format('d/m/Y') }}, à rendre le {{ loan->due_at->format('d/m/Y') }}</p>
                        </div>
                        <x-badge :status="loan->status()">{{ $loan->statusLabel() }}
</x-badge>
                    </li>
                @endforeach
```

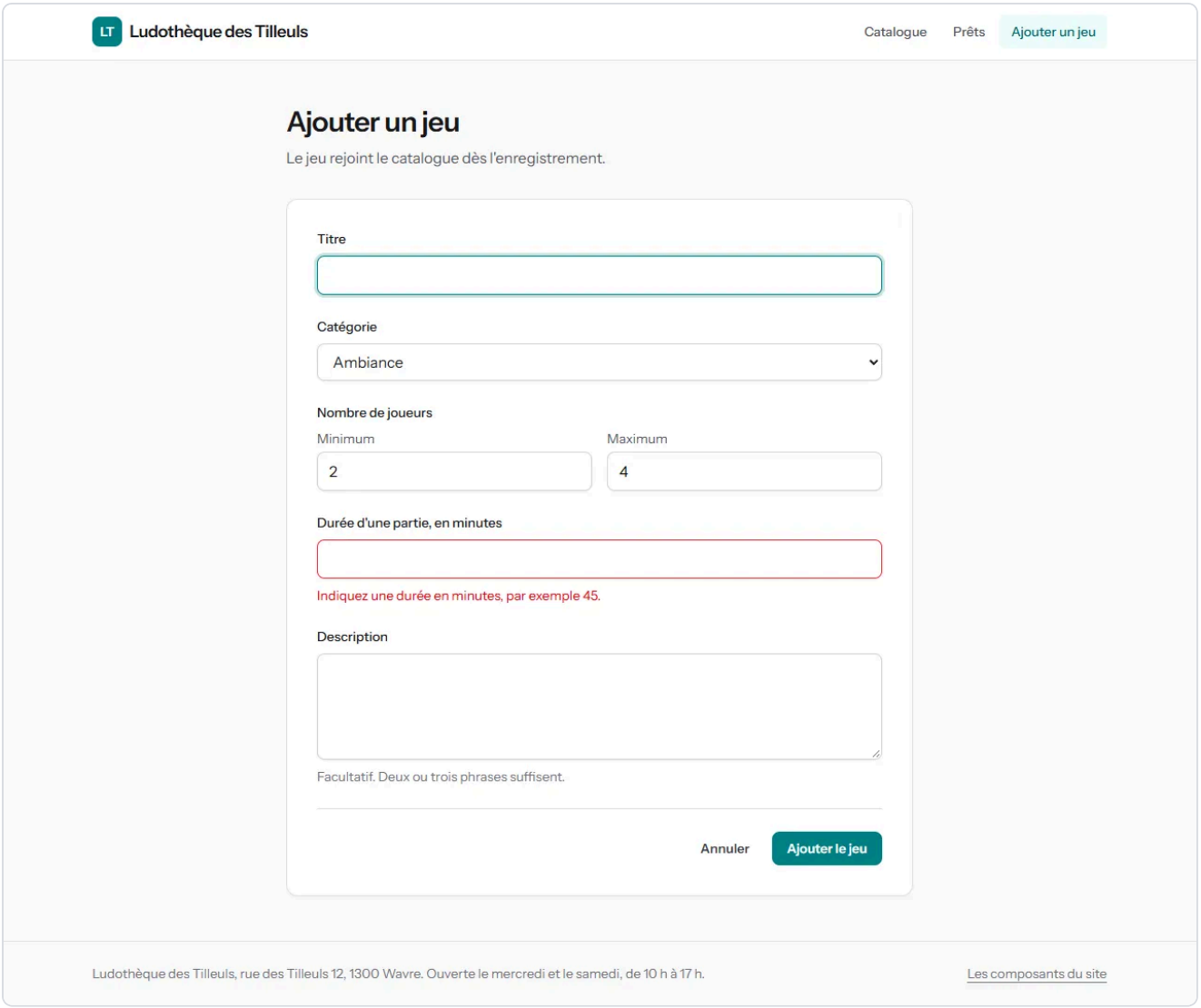
```
            </ul>
        @endif
    </section>
</x-layouts.app>
```

Les deux autres blocs de `<dl>` portent les mêmes classes. `md:items-start` aligne les deux colonnes en haut : sans lui, la grille étire l'image à la hauteur du texte.

14.7 Le formulaire d'ajout (25 minutes)

Mettez en forme `games/create.blade.php`. Le message sous le champ « Durée » est écrit en dur, car le formulaire ne vérifie encore rien : il sert à donner son style à un champ en erreur. Les attributs du champ sont déjà en place. `aria-invalid="true"` annonce aux lecteurs d'écran que la valeur est refusée, et `aria-describedby="duration-error"` relie le champ à son message.

Dans la dernière rangée du formulaire, placez le lien « Annuler » avant `<x-button>`. Sur un ordinateur, « Annuler » s'affiche ainsi à gauche du bouton. Sur un téléphone, une classe de la colonne inverse l'ordre, et le bouton passe au-dessus.



Critères

- [ ] Le titre et le formulaire sont centrés, sur 42 rem de large au plus. Le formulaire a le cadre d'une carte, et ses blocs sont espacés régulièrement.
- [ ] Chaque libellé est au-dessus de son champ. Un clic sur le libellé place le curseur dans le champ.
- [ ] Les champs occupent toute la largeur du cadre, et « Minimum » et « Maximum » se partagent une ligne.
- [ ] Au focus, la bordure du champ devient bleu canard, et un halo bleu canard pâle l'entoure, au clic comme au clavier.
- [ ] Le champ « Durée » a une bordure rouge et son message est rouge. Au focus, son halo est rouge.
- [ ] L'aide « Facultatif. Deux ou trois phrases suffisent. » est grise et plus petite que le libellé.
- [ ] À 375 pixels, le bouton « Ajouter le jeu » occupe toute la largeur, au-dessus du lien « Annuler ». À partir de 640 pixels, les deux sont alignés à droite, sur une ligne, « Annuler » à gauche du bouton.
- [ ] Le lien « Annuler » reçoit un contour bleu canard au clavier.

Un champ de saisie reçoit le focus au clic comme au clavier, et la personne doit voir où elle tape dans les deux cas : écrivez `focus:` , et non `focus-visible:` . `focus:border-brand-600` colore la bordure, et `focus:ring-3 focus:ring-brand-600/20` dessine le halo. `focus:outline-hidden` retire le contour du navigateur. La bordure et le halo le remplacent, comme le demande le chapitre 13. En mode de contraste élevé de Windows, `outline-hidden` laisse le contour du navigateur, et le champ reste repérable.

Le champ « Durée » prend les mêmes classes en rouge : `border-red-500` , `text-red-900` , `focus:border-red-600` et `focus:ring-red-600/20` . Son message est en `text-red-600` .

Autres classes nouvelles : `max-w-2xl` , `space-y-6` , `grid-cols-2` , `border-zinc-300` , `placeholder:text-zinc-400` , `text-center` , `flex-col-reverse` , `sm:items-center` et `sm:justify-end` . Les marges `p-6` et `sm:p-8` suivent l'échelle de `p-5` .

XML

```
<x-layouts.app title="Ajouter un jeu">
    <div class="mx-auto max-w-2xl">
        <div class="mb-8">
            <h1 class="text-3xl font-semibold tracking-tight">Ajouter un jeu</h1>
            <p class="mt-2 text-zinc-600">Le jeu rejoint le catalogue dès
l'enregistrement.</p>
        </div>

        <form method="POST" action="{{ route('games.store') }}" class="space-y-6
rounded-xl border border-zinc-200 bg-white p-6 shadow-xs sm:p-8">
            @csrf

            <div>
                <label for="title" class="block text-sm font-medium">Titre</label>
                <input type="text" id="title" name="title" required class="mt-2 block
w-full rounded-lg border border-zinc-300 bg-white px-3 py-2 shadow-xs placeholder:text-
zinc-400 focus:border-brand-600 focus:ring-3 focus:ring-brand-600/20 focus:outline-
hidden">
            </div>

            …

            <fieldset>
                <legend class="text-sm font-medium">Nombre de joueurs</legend>

                <div class="mt-2 grid grid-cols-2 gap-4">
                    <div>
                        <label for="min_players" class="block text-sm text-zinc-
600">Minimum</label>
                        <input type="number" id="min_players" name="min_players"
min="1" max="20" value="2" required class="mt-1 block w-full rounded-lg border border-
zinc-300 bg-white px-3 py-2 shadow-xs focus:border-brand-600 focus:ring-3 focus:ring-
brand-600/20 focus:outline-hidden">
                    </div>
                    …
                </div>
            </fieldset>

            <div>
                <label for="duration" class="block text-sm font-medium">Durée d'une
partie, en minutes</label>
                <input type="number" id="duration" name="duration" min="5" required
aria-invalid="true" aria-describedby="duration-error" class="mt-2 block w-full rounded-
lg border border-red-500 bg-white px-3 py-2 text-red-900 shadow-xs focus:border-red-600
focus:ring-3 focus:ring-red-600/20 focus:outline-hidden">
                <p id="duration-error" class="mt-2 text-sm text-red-600">Indiquez une
durée en minutes, par exemple 45.</p>
            </div>

            <div>
                <label for="description" class="block text-sm font-
medium">Description</label>
                <textarea id="description" name="description" rows="4" aria-
describedby="description-hint" class="mt-2 block w-full rounded-lg border border-zinc-
```

```
300 bg-white px-3 py-2 shadow-xs focus:border-brand-600 focus:ring-3 focus:ring-brand-
600/20 focus:outline-hidden"></textarea>
                <p id="description-hint" class="mt-2 text-sm text-zinc-500">Facultatif.
Deux ou trois phrases suffisent.</p>
            </div>

            <div class="flex flex-col-reverse gap-3 border-t border-zinc-200 pt-6
sm:flex-row sm:items-center sm:justify-end">
                <a href="{{ route('games.index') }}" class="rounded-md px-3 py-2 text-
center text-sm font-semibold text-zinc-700 hover:text-zinc-900 focus-visible:outline-2
focus-visible:outline-brand-600">Annuler</a>
                <x-button type="submit">Ajouter le jeu</x-button>
            </div>
        </form>
    </div>
</x-layouts.app>
```

Le `<select>` de la catégorie porte les classes du champ « Titre », sans `placeholder:text-zinc-400`. Le champ « Maximum » porte celles de « Minimum ». `flex-col-reverse` inverse l'ordre de la colonne : sur un téléphone, le bouton passe au-dessus du lien, et la colonne l'étire sur toute la largeur.

Le code qui sent mauvais

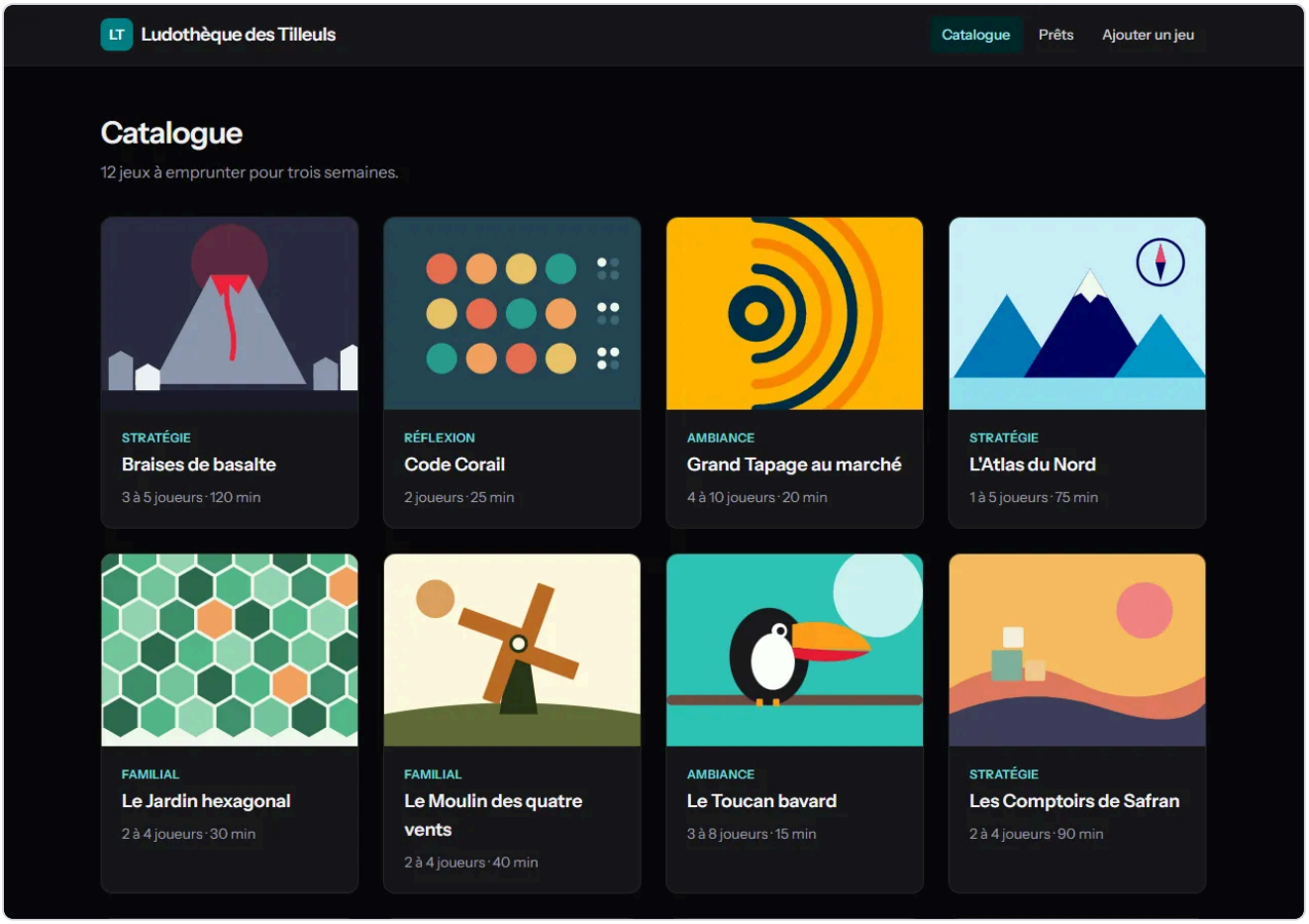
En binôme, cinq minutes. Dans le corrigé de 14.7, les six champs répètent presque la même liste de quatorze classes. La ludothèque veut maintenant des champs aux coins moins arrondis, `rounded-md` au lieu de `rounded-lg`. Combien de lignes modifiez-vous, et que faudrait-il créer pour n'en modifier qu'une ?

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/tailwind-css-exercices

14.8 Bonus : le mode sombre (30 minutes)

Cet exercice est un bonus, à faire en dehors des séances. Le préfixe `dark:` du chapitre 13 applique une classe quand le système est réglé en mode sombre. Donnez une version sombre à tout le site : le layout, les quatre composants et les cinq vues.

Pour tester sans changer le réglage de votre système, ouvrez les outils de développement de Chrome ou d'Edge avec F12. Appuyez ensuite sur Ctrl+Maj+P, tapez `dark`, et choisissez « Emulate CSS prefers-color-scheme: dark ».



Critères

- [ ] Sur les cinq pages, le fond est presque noir, et les surfaces sont un peu plus claires : en-tête, cartes, tableau, formulaire.
- [ ] Tout texte reste lisible. Les gris passent à une nuance plus claire, et le bleu canard à `brand-300`.
- [ ] Les bordures et les traits de séparation restent visibles, sans briller.
- [ ] Les trois badges, le bouton secondaire et le lien actif du menu ont leur version sombre. Les boutons principal et danger ne changent pas.
- [ ] Les champs du formulaire ont un fond sombre, et le texte tapé est clair.
- [ ] De retour en mode clair, rien n'a changé.

Clair	Sombre
`bg-zinc-50` , le fond de la page	`dark:bg-zinc-950`
`bg-white` , une surface	`dark:bg-zinc-900`
`border-zinc-200` , `divide-zinc-200`	`dark:border-zinc-800` , `dark:divide-zinc-800`
`text-zinc-900` , `text-zinc-600`	`dark:text-zinc-100` , `dark:text-zinc-400`
`text-brand-700`	`dark:text-brand-300`
`bg-sky-50 text-sky-700 ring-sky-600/20`	`dark:bg-sky-400/10 dark:text-sky-300 dark:ring-sky-400/30`

Le `<body>` et l'en-tête du layout :

```xml
<body class="flex min-h-screen flex-col bg-zinc-50 font-sans text-zinc-900 antialiased
dark:bg-zinc-950 dark:text-zinc-100">
    <header class="border-b border-zinc-200 bg-white dark:border-zinc-800 dark:bg-zinc-
900">
```

La carte et le bouton secondaire :

```xml
<article {{ $attributes->class([
    'overflow-hidden rounded-xl border border-zinc-200 bg-white shadow-xs',
    'dark:border-zinc-800 dark:bg-zinc-900',
]) }}>
```

```xml
'bg-white text-zinc-900 ring-1 ring-zinc-300 ring-inset hover:bg-zinc-50 focus-
visible:outline-zinc-500 disabled:hover:bg-white dark:bg-zinc-800 dark:text-zinc-100
dark:ring-zinc-700 dark:hover:bg-zinc-700' => $variant === 'secondary',
```

Le badge et le lien de menu :

```xml
'bg-sky-50 text-sky-700 ring-sky-600/20 dark:bg-sky-400/10 dark:text-sky-300 dark:ring-
sky-400/30' => status===’ongoing’,
’bg-red-50text-red-700ring-red-600/20dark:bg-red-400/10dark:text-red-300dark:ring-red-4
00/30’=>status === 'ongoing',
'bg-red-50 text-red-700 ring-red-600/20 dark:bg-red-400/10 dark:text-red-300 dark:ring-
red-400/30' => status===’ongoing’,
’bg-red-50text-red-700ring-red-600/20dark:bg-red-400/10dark:text-red-300dark:ring-red-4
00/30’=>status === 'overdue',
'bg-zinc-100 text-zinc-600 ring-zinc-500/20 dark:bg-zinc-400/10 dark:text-zinc-300
dark:ring-zinc-400/30' => $status === 'returned',
```

```xml
'bg-brand-50 text-brand-700 dark:bg-brand-950 dark:text-brand-200' => active,
’text-zinc-600hover:bg-zinc-100hover:text-zinc-900dark:text-zinc-300dark:hover:bg-zinc-
800dark:hover:text-white’=>!active,
'text-zinc-600 hover:bg-zinc-100 hover:text-zinc-900 dark:text-zinc-300 dark:hover:bg-
zinc-800 dark:hover:text-white' => ! active,
’text-zinc-600hover:bg-zinc-100hover:text-zinc-900dark:text-zinc-300dark:hover:bg-zinc-
800dark:hover:text-white’=>!active,
```

L'en-tête du tableau ajoute `dark:bg-zinc-800/50` . Grâce aux composants, le mode sombre du bouton, de la carte et du badge s'écrit une fois pour tout le site. Les champs du formulaire n'ont pas de composant : chacun ajoute `dark:border-zinc-700 dark:bg-zinc-950` , comme le champ « Titre » :

```xml
<input type="text" id="title" name="title" required class="mt-2 block w-full rounded-lg
border border-zinc-300 bg-white px-3 py-2 shadow-xs placeholder:text-zinc-400
focus:border-brand-600 focus:ring-3 focus:ring-brand-600/20 focus:outline-hidden
dark:border-zinc-700 dark:bg-zinc-950">
```

À retenir

- `@vite` dans le layout charge le CSS et le JavaScript de chaque page, et `composer run dev` produit la règle de chaque nouvelle classe.
- Un conteneur `mx-auto max-w-6xl`, répété avec les mêmes marges, aligne l'en-tête, le contenu et le pied de page.
- Un composant Blade écrit ses classes une fois et choisit sa variante avec `@class` ou `$attributes->class()`.
- Une classe sans préfixe vise le téléphone, et `sm:`, `md:`, `lg:` ou `xl:` la remplacent à partir d'une largeur.
- Le script du menu tient `aria-expanded` à jour pour les lecteurs d'écran.

Six cartes

Pourquoi `w-full` sur `<main>` ?

Dans une colonne Flex, un enfant aux marges `auto` ne prend que la largeur de son contenu. `w-full` lui rend toute la largeur.

Pourquoi `h-full` sur la carte du catalogue ?

La grille étire le `<li>` à la hauteur de la ligne, et `h-full` étire la carte à la hauteur du `<li>`.

Pourquoi un `<div class="overflow-x-auto">` autour du tableau ?

Sur un téléphone, seul ce `<div>` défile de côté, et non toute la page.

Que fait `group-aria-expanded:hidden` ?

Il cache l'élément quand le parent marqué `group` porte `aria-expanded="true"`.

`focus:` ou `focus-visible:` ?

`focus-visible:` pour un bouton ou un lien, `focus:` pour un champ, qui montre son focus même au clic.

Où va le script du menu ?

Dans `resources/js/app.js`, jamais dans une balise `<script>` du layout.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/tailwind-css-exercices

3.15 Formulaires : créer et valider

Bloc Laravel · séance 8 · ~60 min

Slides du chapitre (PDF) PDF

Jusqu'ici, les données de vos sites venaient d'un seeder ou d'une route bac à sable. Dans ce chapitre, Nadia, qui tient une friterie à Wavre, ajoute elle-même un produit à sa carte depuis un formulaire. Vous écrivez ce formulaire, puis vous vérifiez chaque champ avant de l'enregistrer.

À la fin de ce chapitre, vous serez capables de :

1. **écrire** un formulaire d'ajout et la route `POST` qui le reçoit,
2. **valider** les champs reçus, et **lire** ce que Laravel fait quand une règle échoue,
3. **afficher** les erreurs et les anciennes valeurs dans le formulaire,

4. **traduire** les messages en français,
5. **confirmer** l'enregistrement par un message flash.

1. La friterie

Installer le dépôt (10 minutes, une seule fois)

Sur github.com/opmvpc/friterie-26, créez votre copie avec le bouton Fork. Dans un terminal ouvert dans le dossier de Herd ou de Laragon, préparez le projet comme au chapitre 14 :

```bash
git clone https://github.com/VOTRE-COMPTE/friterie-26.git
cd friterie-26
composer install
cp .env.example .env
php artisan key:generate
php artisan migrate --seed
npm install
```

Si `migrate` propose de créer `database/database.sqlite`, appuyez sur Entrée. La base reçoit la carte de Nadia : quatre catégories, huit allergènes et seize produits. Lancez ensuite cette commande, et laissez son terminal ouvert :

```bash
composer run dev
```

Ouvrez `http://friterie-26.test`. Avec Laragon, rechargez d'abord Laragon depuis son menu.

Les pages

Adresse	Vue	Ce qu'elle affiche
`/carte`	`products/index`	les produits par catégorie, avec prix, allergènes et badge « En rupture »
`/produits/{id}`	`products/show`	la fiche d'un produit
`/categories/{id}`	`categories/show`	une catégorie et ses produits
`/composants`	`styleguide`	le guide de style, comme au chapitre 14

Le site n'a pas de comptes utilisateurs : l'authentification est au programme du cours « Projet Web Dynamique ».

Ces pages sont déjà écrites, avec ce que vous connaissez des chapitres 7 à 12. Avant de continuer, lisez `app/Http/Controllers/ProductController.php`, puis les modèles `Category`, `Product` et `Allergen`, dans `app/Models/`. Repérez dans `Product` la liste `#[Fillable]` et la méthode `isSoldOut()` : ce chapitre et le suivant s'en servent. Voici `routes/web.php`, sous ses trois lignes `use` :

```php
Route::redirect('/', '/carte');

Route::get('/carte', [ProductController::class, 'index'])->name('products.index');
Route::get('/produits/{id}', [ProductController::class, 'show'])->whereNumber('id')
->name('products.show');

Route::get('/categories/{id}', [CategoryController::class, 'show'])->whereNumber('id')
->name('categories.show');

Route::view('/composants', 'styleguide')->name('styleguide');
```

Quatre composants servent aux formulaires : `<x-label>`, `<x-input>`, `<x-select>` et `<x-textarea>`. Ce sont ceux imaginés à la fin du chapitre 14 : chacun écrit une seule fois les classes de l'exercice 14.7, et `/composants` les montre normaux puis en erreur. Ils diffèrent du corrigé du chapitre 14 sur deux points : chacun lit lui-même les erreurs, sans prop `invalid`, et porte sa marge `mt-2`.

Le code qui sent mauvais

En binôme, cinq minutes. Un collègue de Nadia veut retirer un produit d'un simple clic. Il ajoute cette route, et un lien « Retirer » sur chaque fiche :

```php
Route::get('/produits/{id}/retirer', fn (int id)⇒Product::findOrFail(id) ⇒
Product::findOrFail(id)⇒Product::findOrFail(id)→delete());
```

Le lien fonctionne : `/produits/16/retirer` affiche `1`, la valeur renvoyée par `delete()`, et le café a disparu de la carte. Qu'arrive-t-il la première fois qu'un moteur de recherche, ou le préchargement des liens du navigateur, suit ce lien ?

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/formulaires-creer-et-valider

2. Le formulaire d'ajout

Nadia veut ajouter la viandelle à sa carte. Il lui faut deux routes : l'une affiche le formulaire, l'autre reçoit ce qu'il envoie. Ajoutez-les sous la route `/carte` :

```php
Route::get('/produits/nouveau', [ProductController::class, 'create'])
->name('products.create');
Route::post('/produits', [ProductController::class, 'store'])->name('products.store');
```

Ici, `whereNumber('id')` protège déjà la route `/produits/{id}`, car `nouveau` n'est pas un nombre. Déclarez quand même `/produits/nouveau` avant `/produits/{id}`, comme le demande le piège de l'ordre du chapitre 7 : sans `whereNumber`, l'ordre serait la seule protection.

Dans `ProductController`, la méthode `create()` passe les catégories à la vue, dans l'ordre de la carte :

```php
public function create(): View
{
    categories=Category::orderBy(’position’)->get();returnview(’products.create’,
[’categories’⇒categories = Category::orderBy('position')→get();

    return view('products.create', ['categories' ⇒ categories=Category::orderBy(’position’)
->get();returnview(’products.create’,[’categories’⇒categories]);
}
```

```bash
php artisan make:view products.create
```

```
   INFO  View [resources/views/products/create.blade.php] created successfully.
```

Chez vous, Artisan affiche le chemin complet du fichier, depuis la racine du disque. Le fichier ne contient qu'un `<div>` et une citation. Remplacez tout son contenu par le formulaire :

```xml
<x-layouts.app title="Ajouter un produit">
    <div class="mx-auto max-w-2xl">
        <h1 class="mb-8 text-3xl font-semibold tracking-tight">Ajouter un produit</h1>

        <form method="POST" action="{{ route('products.store') }}" class="space-y-6 rounded-xl border border-zinc-200 bg-white p-6 shadow-xs sm:p-8 dark:border-zinc-800 dark:bg-zinc-900">
            @csrf

            <div>
                <x-label for="name">Nom</x-label>
                <x-input name="name" />
            </div>

            <div>
                <x-label for="category_id">Catégorie</x-label>
                <x-select name="category_id">
                    <option value="">Choisissez une catégorie</option>
                    @foreach (categoriesascategories as categoriesascategory)
                        <option value="{{ category->id }}">{{ category->name }}</option>
                    @endforeach
                </x-select>
            </div>

            <div>
                <x-label for="price">Prix (€)</x-label>
                <x-input name="price" type="number" step="0.10" min="0" />
            </div>

            <div>
                <x-label for="description">Description</x-label>
                <x-textarea name="description"></x-textarea>
            </div>

            <div class="flex justify-end">
                <x-button type="submit">Ajouter le produit</x-button>
            </div>
        </form>
    </div>
</x-layouts.app>
```

Chaque `<option>` envoie le numéro de la catégorie, que la colonne `category_id` attend, et non son nom. `step="0.10"` fait avancer les flèches du prix de dix centimes. Le navigateur refuse alors un prix comme `2.85` et propose « 2,8 » ou « 2,9 » : écrivez `step="0.01"` si la carte a besoin des centimes. Sans `type="submit"`, le composant `button` écrit `type="button"`, et le clic n'envoie rien. `@csrf` ajoute le jeton du chapitre 9.

```bash
php artisan route:list --path=produits
```

```
  POST       produits ..................................... products.store ›
ProductController@store
  GET|HEAD   produits/nouveau ........................... products.create ›
ProductController@create
  GET|HEAD   produits/{id} .............................. products.show ›
ProductController@show

                                                                  Showing [3]
routes
```

Enregistrer le produit

Au chapitre 9, `input()` lisait un champ de la requête. La méthode `only()` renvoie un tableau des seuls champs cités. Importez `RedirectResponse` et `Request` sous `Product`, puis ajoutez `store()` sous `create()` :

PHP
```php
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
```

PHP
```php
public function store(Request $request): RedirectResponse
{
    $product = Product::create($request->only(['name', 'category_id', 'price',
'description']));

    return redirect()->route('products.show', $product->id);
}
```

`create()` remplit les colonnes de `#[Fillable]`, comme au chapitre 10, et la redirection du chapitre 7 ouvre la fiche du nouveau produit. Sur `http://friterie-26.test/produits/nouveau`, tapez `Viandelle`, choisissez Snacks, tapez `2.80` et « Une saucisse de viande hachée, frite à la commande. ». Le navigateur arrive sur `/produits/17`, la fiche de la viandelle à 2,80 €.

Envoyez maintenant le formulaire sans rien remplir. Le navigateur affiche une erreur 500 :

```
Illuminate\Database\QueryException
SQLSTATE[23000]: Integrity constraint violation: 19 NOT NULL constraint failed:
products.category_id (Connection: sqlite, …, SQL: insert into "products" ("name",
"category_id", "price", "description", "updated_at", "created_at") values (?, ?, ?, ?, 2026-
10-07 15:27:23, 2026-10-07 15:27:23))
```

Laravel transforme chaque champ vide en `null`. La colonne `category_id` n'accepte pas `null`, et la base refuse la ligne. Le message ne cite que la première colonne fautive de la table, mais `name` et `price` manquent aussi.

La base ne refuse pas tout. Tapez le nom « Frite géante de la maison avec supplément de sauce et oignons frits croustillants », choisissez Frites et un prix de `0`. `/produits/18` affiche ces 81 caractères à `0,00 €`. La migration déclare `string('name', 60)`, mais SQLite ne vérifie pas cette longueur, et aucune colonne n'interdit un prix de zéro.

3. Valider

Vérifier chaque champ reçu avant de s'en servir s'appelle la **validation**. La méthode `validate()` de la requête reçoit, pour chaque champ, une liste de conditions :

```php
public function store(Request $request): RedirectResponse
{
    $validated = $request->validate([
        'name' => ['required', 'string', 'max:60'],
        'category_id' => ['required', 'integer', 'exists:categories,id'],
        'price' => ['required', 'numeric', 'min:0.5', 'max:50'],
        'description' => ['nullable', 'string', 'max:500'],
    ]);

    $product = Product::create($request->only(['name', 'category_id', 'price',
'description']));

    return redirect()->route('products.show', $product->id);
}
```

Chaque condition s'appelle une **règle** :

Règle	Ce qu'elle vérifie
`required`	le champ est rempli
`string`, `max:60`	un texte de 60 caractères au plus
`numeric`, `min:0.5`, `max:50`	un nombre entre 0,5 et 50
`nullable`	le champ peut rester vide, et ses autres règles ne s'appliquent alors pas
`integer`	un nombre entier, et non un texte ou une liste de valeurs
`exists:categories,id`	la table `categories` a une ligne dont l'`id` vaut la valeur reçue

Pour `exists`, Laravel envoie `select count(*) as "aggregate" from "categories" where "id" = ?`, avec la valeur reçue à la place du `?`. Le `<select>` ne propose que quatre catégories, mais les outils de développement changent une option en deux clics. Une requête modifiée peut même envoyer une liste de numéros, que `exists` accepterait : `integer` la refuse. De même, `min="0"` fait bloquer un prix négatif par le navigateur, mais une requête modifiée passe sans ce contrôle. **Vérifiez chaque champ sur le serveur, même ceux que le navigateur contrôle déjà.**

Quand une règle échoue

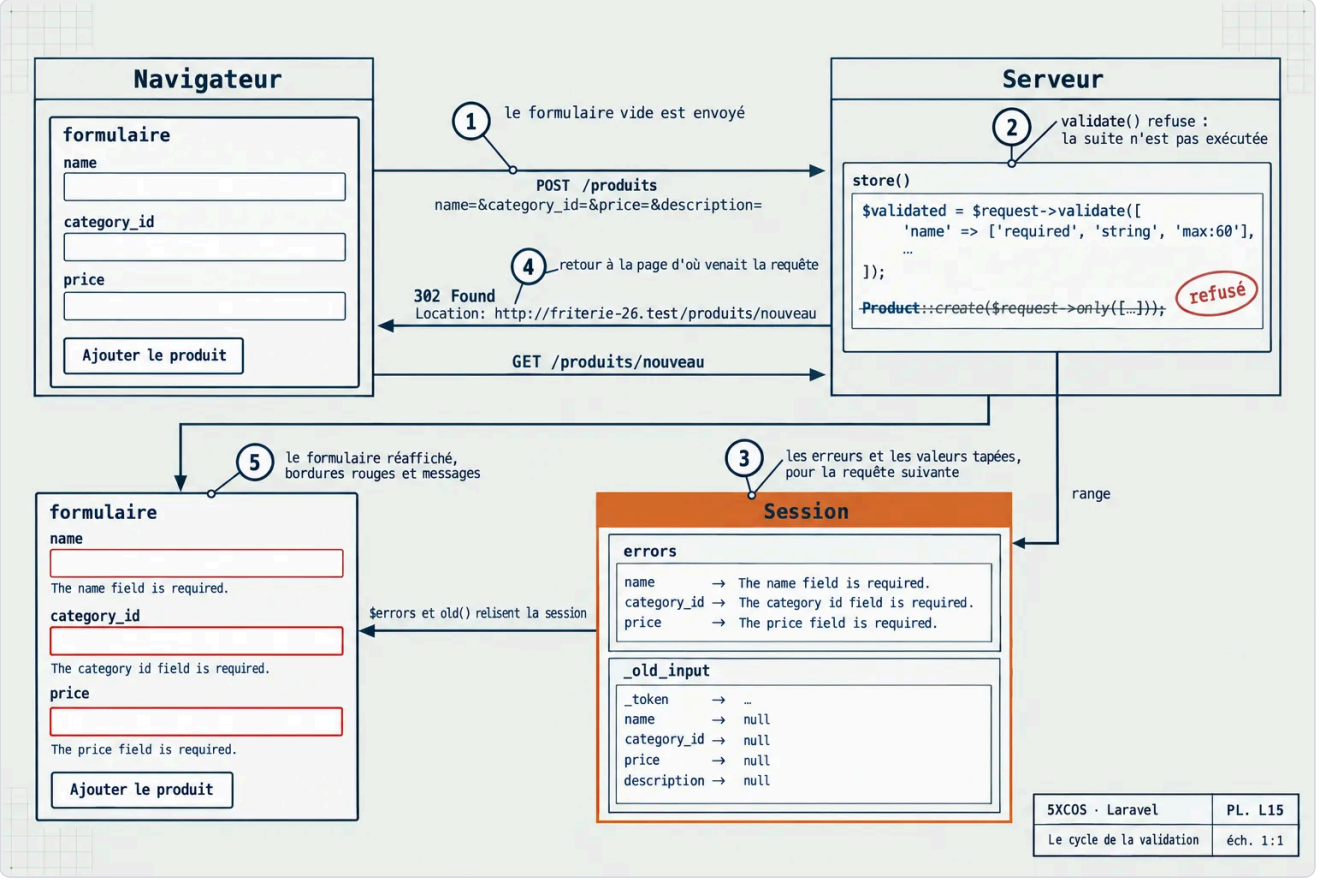
Ouvrez les outils de développement avec F12, onglet Réseau, et cochez « Conserver le journal ». Envoyez de nouveau le formulaire vide. Cette fois, le formulaire revient, vide, avec trois champs bordés de rouge et sans un mot d'explication. L'onglet Réseau montre deux requêtes : `produits` en `POST`, code 302, puis `nouveau` en `GET`, code 200. La réponse à la première commence ainsi :

```
HTTP/1.1 302 Found
…
Location: http://friterie-26.test/produits/nouveau
…
```

Dans l'ordre, Laravel a fait ceci :

1. `validate()` a refusé trois champs, et la suite de `store()` n'a pas été exécutée.
2. Laravel a rangé dans la session les messages d'erreur et les valeurs tapées.
3. Il a répondu par une redirection 302 vers la page d'où venait la requête.
4. Le navigateur a suivi la redirection, et redemandé le formulaire.

La session, vue au chapitre 9, est un espace de stockage que le serveur réserve à chaque visiteur. Les erreurs et les valeurs tapées y attendent la requête suivante, celle du formulaire.



Afficher les erreurs

Dans chaque vue, Laravel fournit la variable `$errors`, qui contient les messages rangés dans la session. Les bordures rouges viennent des composants, qui lisent cette variable. Ouvrez `resources/views/components/input.blade.php` :

```XML
{{--
    Un champ de saisie. Le composant lit lui-même errors : si le serveur a refusé la valeur,
    la bordure passe en rouge et le champ annonce son message (aria-invalid, aria-
describedby).
    La vue garde @error('name') … @enderror sous le champ, avec id="name-error".
    Classes de l'exercice 14.7 du chapitre Tailwind.
--}}
@props(['name', 'type' => 'text'])

@php
    invalid = errors->has(errors→has(errors->has(name);
@endphp

<input
    type="{{ type }}"
    name="{{ name }}"
    @if ($invalid) aria-invalid="true" aria-describedby="{{ $name }}-error" @endif
    {{ $attributes→merge(['id' => $name])→class([
        'mt-2 block w-full rounded-lg border bg-white px-3 py-2 shadow-xs focus:ring-3
focus:outline-hidden dark:bg-zinc-950',
        'border-zinc-300 placeholder:text-zinc-400 focus:border-brand-600 focus:ring-brand-
600/20 dark:border-zinc-700' => ! $invalid,
        'border-red-500 text-red-900 focus:border-red-600 focus:ring-red-600/20 dark:border-
red-400 dark:text-red-100' => $invalid,
    ]) }}
>
```

`@php … @endphp` exécute quelques lignes de PHP dans le template. Ici, elles ne lisent pas la base comme au chapitre 5 : elles rangent une condition d'affichage dans `$invalid`. `$errors→has('name')` vaut `true` quand le champ `name` a été refusé. Le composant choisit alors les classes rouges et les attributs `aria-*` de l'exercice 14.7. Le message, lui, s'écrit dans la vue, sous chaque champ :

XML

```
<x-layouts.app title="Ajouter un produit">
    <div class="mx-auto max-w-2xl">
        <h1 class="mb-8 text-3xl font-semibold tracking-tight">Ajouter un produit</h1>

        <form method="POST" action="{{ route('products.store') }}" class="space-y-6 rounded-
xl border border-zinc-200 bg-white p-6 shadow-xs sm:p-8 dark:border-zinc-800 dark:bg-zinc-
900">
            @csrf

            <div>
                <x-label for="name">Nom</x-label>
                <x-input name="name" :value="old('name')" />
                @error('name')
                    <p id="name-error" class="mt-2 text-sm text-red-600 dark:text-red-400">
{{ message }}</p>
                @enderror
            </div>

            <div>
                <x-label for="category_id">Catégorie</x-label>
                <x-select name="category_id">
                    <option value="">Choisissez une catégorie</option>
                    @foreach (categories as category)
                        <option value="{{ $category->id }}" @selected(old('category_id') ==
category->id)>{{ category->name }}</option>
                    @endforeach
                </x-select>
                @error('category_id')
                    <p id="category_id-error" class="mt-2 text-sm text-red-600 dark:text-
red-400">{{ message }}</p>
                @enderror
            </div>

            <div>
                <x-label for="price">Prix (€)</x-label>
                <x-input name="price" type="number" step="0.10" min="0"
:value="old('price')" />
                @error('price')
                    <p id="price-error" class="mt-2 text-sm text-red-600 dark:text-red-400">
{{ message }}</p>
                @enderror
            </div>

            <div>
                <x-label for="description">Description</x-label>
                <x-textarea name="description">{{ old('description') }}</x-textarea>
                @error('description')
                    <p id="description-error" class="mt-2 text-sm text-red-600 dark:text-
red-400">{{ message }}</p>
                @enderror
            </div>

            <div class="flex justify-end">
                <x-button type="submit">Ajouter le produit</x-button>
            </div>
```

```
        </form>
    </div>
</x-layouts.app>
```

- `@error('name')` n'affiche son contenu que si le champ `name` a un message, rangé dans `$message`. L'`id` `name-error` est celui qu'annonce `aria-describedby`.
- `old('name')` renvoie la valeur tapée avant la redirection, ou `null`. Les deux-points de `:value` transmettent l'expression PHP, comme au chapitre 9.
- `@selected(…)` écrit `selected` quand la condition est vraie, et `{{ old('description') }}` remplit le slot du `<x-textarea>`.

Envoyez de nouveau le formulaire vide :

```
The name field is required.
The category id field is required.
The price field is required.
```

Tapez ensuite `Boulette` et `3.50`, sans catégorie. Le formulaire revient avec un seul message, sous la catégorie, et garde le nom et le prix. **Quand une règle échoue, `@error` dit ce qui ne va pas, et `old()` rend ce qui avait été tapé.**

`$validated`

`validate()` renvoie un tableau, rangé ici dans `$validated`. Ajoutez `dd($validated);` sous l'appel, puis envoyez `Boulette`, Snacks, `3.50`, sans description :

```
array:4 [▼ // app/Http/Controllers/ProductController.php:43
  "name" => "Boulette"
  "category_id" => "2"
  "price" => "3.50"
  "description" => null
]
```

Le tableau ne contient que les champs qui ont des règles et qui figurent dans la requête, avec des valeurs qui les respectent. Retirez le `dd()`, et remplacez la ligne de `create()` :

```php
product=Product::create(product = Product::create(product=Product::create(validated);
```

ATTENTION

`#[Fillable]` ne remplace pas `$validated`. Sa liste contient `sold_out_at`, la colonne de la rupture. `$request->all()` renvoie tous les champs reçus, cités ou non dans les règles. Avec `Product::create($request->all())`, une requête qui ajoute un champ `sold_out_at` crée un produit déjà en rupture, sans qu'aucune règle l'ait examiné.

Remettez la carte à neuf, pour retirer la frite géante à zéro euro (sortie raccourcie) :

```bash
php artisan migrate:fresh --seed
```

```
…
   INFO  Seeding database.

  Database\Seeders\CategorySeeder ..................................... RUNNING
  Database\Seeders\CategorySeeder ................................ 36 ms DONE

  Database\Seeders\AllergenSeeder ..................................... RUNNING
  Database\Seeders\AllergenSeeder ................................ 62 ms DONE

  Database\Seeders\ProductSeeder ..................................... RUNNING
  Database\Seeders\ProductSeeder ................................ 229 ms DONE
```

La carte retrouve ses seize produits, sans la viandelle.

4. Les messages en français

Nadia lit « The category id field is required. ». Au chapitre 6, vous avez vu qu'un projet neuf n'a pas de dossier `lang/` pour les traductions. Le paquet **Laravel-Lang** fournit les traductions de tous les messages du framework. Installez-le :

```bash
composer require laravel-lang/common --dev
```

Sortie raccourcie, la commande installe vingt-cinq paquets :

```
./composer.json has been updated
Running composer update laravel-lang/common
…
  - Installing laravel-lang/common (6.8.0): Extracting archive
…
Using version ^6.8 for laravel-lang/common
```

Copiez ensuite les fichiers français dans le projet :

```bash
php artisan lang:add fr
```

Sortie raccourcie :

```
   INFO  Collecting translations...
…
   INFO  Storing changes...

  fr.json ..................................................... 12.22ms DONE
  fr/actions.php .............................................. 66.85ms DONE
  fr/auth.php ..................................................  3.64ms DONE
  fr/http-statuses.php ......................................... 4.88ms DONE
  fr/pagination.php ............................................ 3.08ms DONE
  fr/passwords.php ............................................. 2.78ms DONE
  fr/validation.php ............................................ 30.08ms DONE
```

`lang:add` crée le dossier `lang/` à la racine, avec un dossier `fr/` et un fichier `fr.json` à côté. `fr/validation.php` contient un message par règle. `fr.json` traduit les phrases que Laravel écrit directement en anglais, comme le titre de la page 404.

```
lang/
├── fr/
│   ├── actions.php
│   ├── auth.php
│   ├── http-statuses.php
│   ├── pagination.php
│   ├── passwords.php
│   └── validation.php
└── fr.json
```

`--dev` range le paquet parmi les dépendances de développement, comme Pest. Les traductions copiées dans `lang/` font partie du projet : le serveur qui héberge le site n'a besoin que d'elles. Le chapitre Localisation rangera aussi dans `lang/` les textes du site lui-même.

Dans `.env`, passez la langue de l'application au français, et faites la même modification dans `.env.example`, son modèle versionné :

```
APP_LOCALE=fr
```

Envoyez de nouveau le formulaire vide. Si les messages restent en anglais, arrêtez `composer run dev` avec Ctrl+C et relancez-le : le serveur lancé avant la modification garde parfois l'ancienne valeur de `.env`. Même remède si vous avez un jour lancé `php artisan config:cache`, avec `php artisan config:clear` en plus. Vous lisez alors :

```
Le champ nom est obligatoire.
Le champ category id est obligatoire.
Le champ prix est obligatoire.
```

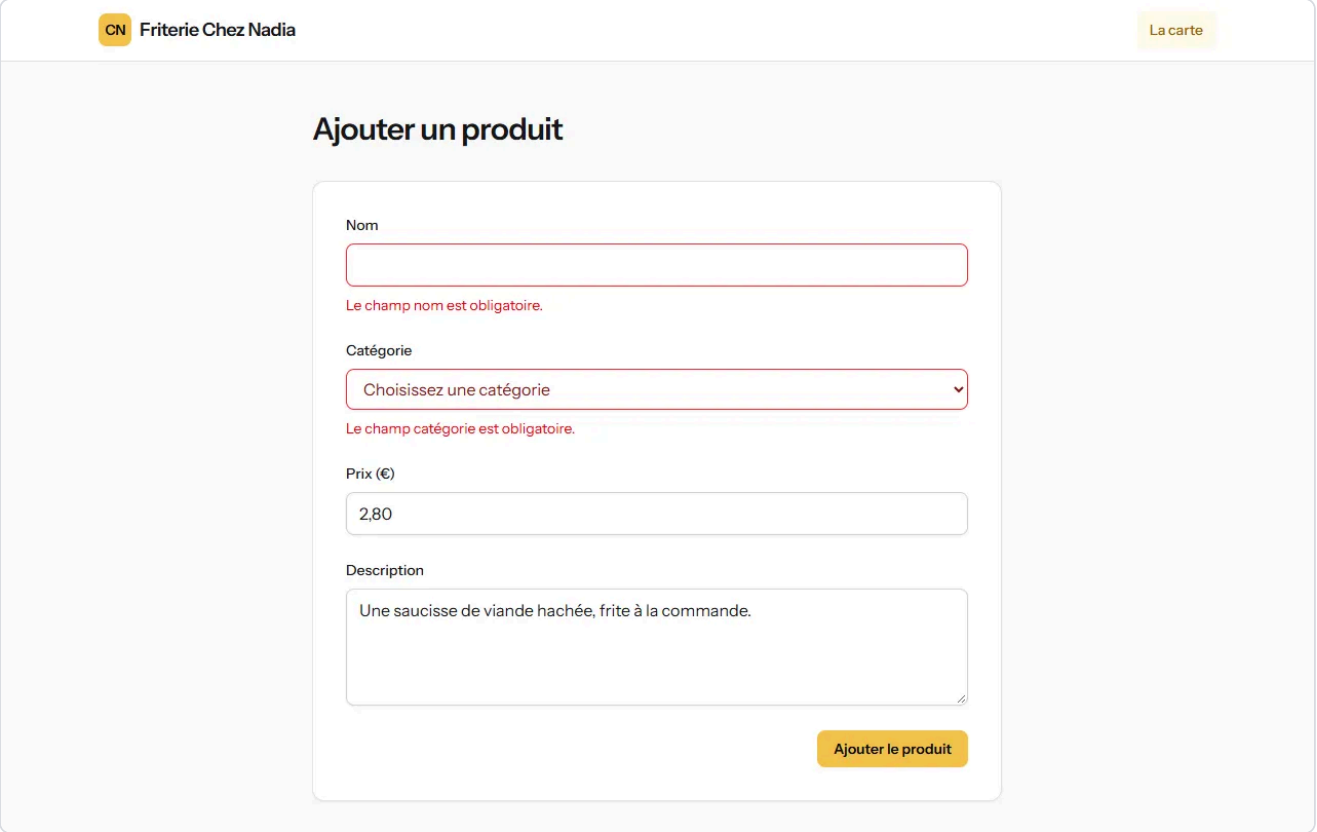
Dans `lang/fr/validation.php`, le message de `required` est `'Le champ :attribute est obligatoire.'`. Laravel remplace `:attribute` par le nom du champ, traduit par le tableau `attributes` en bas du fichier. Ce tableau contient déjà `name` et `price`. Pour `category_id`, absent, Laravel remplace

le `_` par une espace. Ajoutez la ligne après `'body'`, dans l'ordre alphabétique :

```php
'category_id'              => 'catégorie',
```

```
Le champ nom est obligatoire.
Le champ catégorie est obligatoire.
Le champ prix est obligatoire.
```

Laissez maintenant le nom et la catégorie vides, tapez `2.80` dans le prix, et envoyez. Le formulaire revient avec deux messages, et garde le prix :



5. Le message flash

Après un ajout, Nadia arrive sur la fiche sans confirmation. Dans `store()`, ajoutez `with()` à la redirection :

```php
return redirect()->route('products.show', $product->id)->with('status', 'Le produit a été ajouté.');
```

Le navigateur ne garde de la session qu'un identifiant, dans le cookie `friterie-chez-nadia-session`. Les données restent dans la table `sessions`, car `.env` contient `SESSION_DRIVER=database`.

`with('status', …)` écrit la phrase dans la session, sous la clé `status`, pour la requête suivante seulement. Une valeur gardée ainsi le temps d'une requête s'appelle un **message flash**. Les erreurs et les valeurs tapées de la section 3 voyagent de la même façon.

Dans `resources/views/components/layouts/app.blade.php`, en haut de `<main>`, remplacez le commentaire `{{-- Le chapitre 15 affiche ici le message flash. --}}` :

```xml
@session('status')
    <p role="status" class="mb-8 rounded-lg border border-green-200 bg-green-50 px-4 py-3 text-sm font-medium text-green-800 dark:border-green-900 dark:bg-green-950 dark:text-green-200">
        {{ $value }}
    </p>
@endsession
```

`@session('status')` n'affiche son contenu que si la session contient la clé `status`. Dans ce bloc, `$value` contient la phrase rangée sous cette clé. `role="status"` demande aux lecteurs d'écran d'annoncer le message. Ajoutez de nouveau la viandelle : `/produits/17` s'ouvre sous un encadré vert.

```
Le produit a été ajouté.
```

Rechargez la page : l'encadré a disparu, car la requête précédente a consommé le message. **Un message flash s'affiche une fois, sur la page qui suit la redirection.**

6. La liaison de modèle

Dans le dépôt, `show()` cherche le produit elle-même, comme au chapitre 10 :

```php
public function show(int $id): View
{
    $product = Product::findOrFail($id);

    return view('products.show', ['product' => $product]);
}
```

Laravel peut faire ce `findOrFail` à votre place. Dans `routes/web.php`, renommez le paramètre en `{product}`. `whereNumber('id')` porte sur un paramètre qui n'existe plus : retirez-le. Simplifiez ensuite `show()` :

```php
Route::get('/produits/{product}', [ProductController::class, 'show'])
->name('products.show');
```

```php
public function show(Product $product): View
{
    return view('products.show', ['product' => $product]);
}
```

Le paramètre `{product}` et l'argument `$product` portent le même nom, et l'argument a pour type un modèle. Laravel cherche alors le produit dont l'`id` vaut la valeur lue dans l'adresse, le passe à la méthode, et répond 404 s'il ne le trouve pas. Ce mécanisme s'appelle la **liaison de modèle**.

`/produits/17` affiche la viandelle. `/produits/999` et `/produits/abc` répondent 404, avec « Non trouvé », traduit par `fr.json`.

Sans `whereNumber`, l'ordre des routes compte. Déplacez `/produits/nouveau` sous `/produits/{product}` : le formulaire répond 404, car Laravel cherche un produit dont l'`id` vaut `nouveau`. Remettez la route au-dessus.

```bash
php artisan route:list --path=produits
```

```
  POST       produits ................................. products.store ›
ProductController@store
  GET|HEAD   produits/nouveau ......................... products.create ›
ProductController@create
  GET|HEAD   produits/{product} ........................ products.show ›
ProductController@show

                                                                Showing [3]
routes
```

`findOrFail()` reste correct, et les deux écritures donnent la même page. La liaison évite de répéter cette ligne dans chaque méthode qui reçoit un produit.

À retenir

- Un formulaire d'ajout demande deux routes : un `GET` qui l'affiche, et un `POST` qui reçoit ses champs.
- `validate()` vérifie chaque champ avec ses règles, et renvoie les seules valeurs validées, à passer à `create()`.
- Quand une règle échoue, Laravel range les erreurs et les valeurs tapées dans la session, puis renvoie au formulaire, qui les affiche avec `@error` et `old()`.
- Laravel-Lang copie les messages traduits dans `lang/`, et le tableau `attributes` donne à chaque champ son nom français.
- `with()` place un message flash dans la session, et la page qui suit la redirection l'affiche une seule fois.

Exercices

Comptez 20 minutes pour ces deux exercices sur papier. Les exercices de code viennent au chapitre 17, après le chapitre 16 : vous y écrirez le site d'un refuge pour animaux, avec ce que la friterie vous a appris.

15.1 Prédire les messages (sur papier, 10 minutes)

La méthode `store()` de la section 3 reçoit ces trois requêtes, écrites comme le navigateur les envoie, avec `+` pour une espace. La troisième a été modifiée dans les outils de développement. Pour chaque champ refusé, cherchez dans `lang/fr/validation.php` le message de la règle qui échoue, puis remplacez `:attribute` et les autres marques par leur valeur. Écrivez les messages affichés, une fois `category_id` ajouté à `attributes`.

```
1. name=Cornet+de+frites&category_id=&price=3&description=
2. name=Sauce+brasil&category_id=3&price=0.30&description=
3.
name=MitrailletteMitrailletteMitrailletteMitrailletteMitrailletteMitraillette&category_id=99
&price=60&description=
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/formulaires-creer-et-valider

15.2 Le bon verbe (sur papier, 10 minutes)

Relisez la route du code qui sent mauvais, à la section 1 :

```php
Route::get('/produits/{id}/retirer', fn (int id)⇒Product::findOrFail(id) ⇒
Product::findOrFail(id)⇒Product::findOrFail(id)→delete());
```

Pour chaque action, choisissez le verbe HTTP parmi ceux du chapitre 7, et dites si elle part d'un lien ou d'un formulaire avec `@csrf`.

1. Afficher la carte.
2. Afficher le formulaire d'ajout.
3. Enregistrer un nouveau produit.
4. Corriger le prix de la Grande frite.
5. Marquer la Sauce lapin en rupture.
6. Retirer le café de la carte.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/formulaires-creer-et-valider

Six cartes

Que fait `validate()` quand une règle échoue ?

Il arrête la méthode, range les erreurs et les valeurs tapées dans la session, et redirige vers le formulaire.

À quoi sert `old()` ?

À réafficher dans le champ la valeur tapée avant la redirection.

Dans `@error`, d'où vient `$message` ?

De `$errors`, les messages que la validation a rangés dans la session.

Pourquoi `exists:categories,id` ?

Parce qu'une requête peut envoyer n'importe quel numéro, et que la catégorie doit exister dans la table.

Que fait `->with('status', …)` ?

Il écrit un message flash dans la session, que `@session('status')` affiche une seule fois.

Pourquoi `--dev` suffit-il pour Laravel-Lang ?

Parce que `lang:add` copie les traductions dans `lang/`, et le site en ligne n'a besoin que de ces fichiers.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/formulaires-creer-et-valider

3.16 Formulaires : modifier et supprimer

Bloc Laravel · séance 8 · ~60 min

Slides du chapitre (PDF) PDF

Au chapitre 15, Nadia a ajouté la viandelle à sa carte. Une carte change pourtant chaque semaine : un prix augmente, un allergène a été oublié, les croquettes manquent un samedi soir, une sauce ne se vend plus. Dans ce chapitre, vous écrivez les formulaires qui modifient un produit, le marquent en rupture et le retirent de la carte. Vous comparez ensuite vos routes à celles que `Route::resource()` écrirait.

À la fin de ce chapitre, vous serez capables de :

1. **pré-remplir** un formulaire de modification et **l'envoyer** en `PUT`,
2. **expliquer** le champ caché `_method`,
3. **partager** les règles de validation entre l'ajout et la modification,
4. **enregistrer** des cases à cocher avec `sync()`,
5. **supprimer** une ligne depuis un formulaire avec confirmation.

1. Le code qui sent mauvais

En binôme, cinq minutes. Un collègue de Nadia écrit le formulaire qui modifie la Grande frite. Il a retenu du chapitre 7 qu'une modification s'envoie avec le verbe `PUT`, et l'écrit dans l'attribut `method` :

```xml
<form method="PUT" action="/produits/2">
    @csrf
    …
    <x-button type="submit">Enregistrer</x-button>
</form>
```

Nadia passe le prix de 4,00 € à 4,50 € et clique sur Enregistrer. Quelle requête le navigateur envoie-t-il, et qu'affiche la page ?

> Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/formulaires-modifier-et-supprimer

2. Modifier

Il faut deux routes, comme pour l'ajout : l'une affiche le formulaire, l'autre reçoit ce qu'il envoie. Ajoutez-les sous la route `products.show` :

```php
Route::get('/produits/{product}/modifier', [ProductController::class, 'edit'])
->name('products.edit');
Route::put('/produits/{product}', [ProductController::class, 'update'])
->name('products.update');
```

Grâce à la liaison de modèle du chapitre 15, section 6, `edit()` reçoit le produit déjà lu dans la base, sans `findOrFail()`. Elle le passe à la vue avec les catégories :

```php
public function edit(Product $product): View
{
    $categories = Category::orderBy('position')->get();

    return view('products.edit', ['product' => $product, 'categories' => $categories]);
}
```

```bash
php artisan make:view products.edit
```

```
   INFO  View [resources/views/products/edit.blade.php] created successfully.
```

Remplacez tout le contenu du fichier, un `<div>` et une citation, par le formulaire d'ajout, avec cinq changements : le titre, l'adresse d'envoi, la ligne `@method('PUT')`, le second argument de chaque `old()` et le texte du bouton.

XML

```xml
<x-layouts.app :title="'Modifier ' . product->name">
    <div class="mx-auto max-w-2xl">
        <h1 class="mb-8 text-3xl font-semibold tracking-tight">Modifier {{ $product->name }}
</h1>

        <form method="POST" action="{{ route('products.update', $product->id) }}"
class="space-y-6 rounded-xl border border-zinc-200 bg-white p-6 shadow-xs sm:p-8
dark:border-zinc-800 dark:bg-zinc-900">
            @csrf
            @method('PUT')

            <div>
                <x-label for="name">Nom</x-label>
                <x-input name="name" :value="old('name', product->name)" />
                @error('name')
                    <p id="name-error" class="mt-2 text-sm text-red-600 dark:text-red-400">
{{ message }}</p>
                @enderror
            </div>

            <div>
                <x-label for="category_id">Catégorie</x-label>
                <x-select name="category_id">
                    <option value="">Choisissez une catégorie</option>
                    @foreach (categories as category)
                        <option value="{{ $category->id }}" @selected(old('category_id',
product->category_id) == category->id)>{{ $category->name }}</option>
                    @endforeach
                </x-select>
                @error('category_id')
                    <p id="category_id-error" class="mt-2 text-sm text-red-600 dark:text-
red-400">{{ $message }}</p>
                @enderror
            </div>

            <div>
                <x-label for="price">Prix (€)</x-label>
                <x-input name="price" type="number" step="0.10" min="0" :value="old('price',
product->price)" />
                @error('price')
                    <p id="price-error" class="mt-2 text-sm text-red-600 dark:text-red-400">
{{ message }}</p>
                @enderror
            </div>

            <div>
                <x-label for="description">Description</x-label>
                <x-textarea name="description">{{ old('description', product->description)
}}</x-textarea>
                @error('description')
                    <p id="description-error" class="mt-2 text-sm text-red-600 dark:text-
red-400">{{ $message }}</p>
                @enderror
            </div>
```

```
            <div class="flex justify-end">
                <x-button type="submit">Enregistrer</x-button>
            </div>
        </form>
    </div>
</x-layouts.app>
```

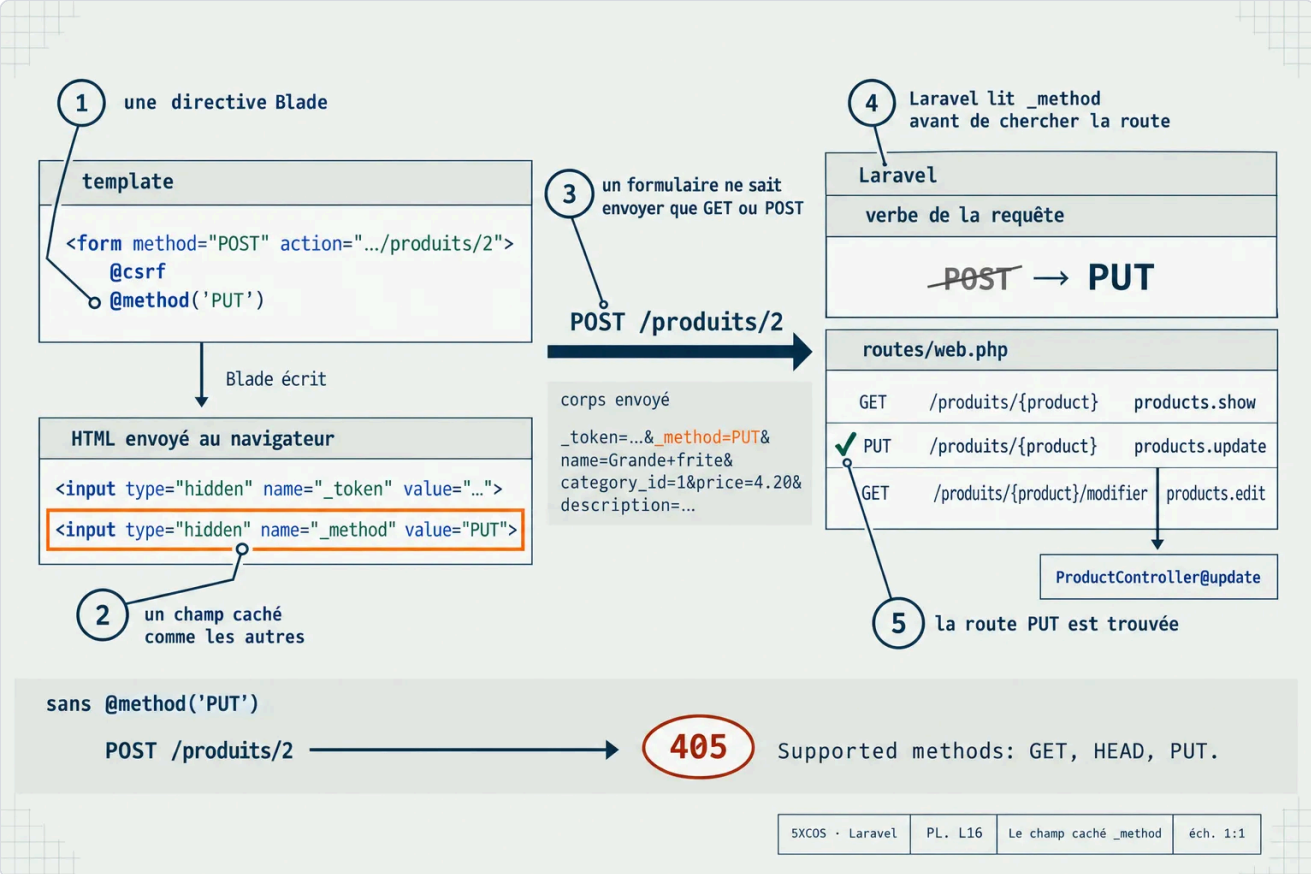
Ouvrez `http://friterie-26.test/produits/2/modifier` : les quatre champs affichent la Grande frite. `old('name', $product→name)` renvoie la valeur gardée dans la session après un refus, et sinon son second argument. Au premier affichage, la session ne contient aucune valeur tapée, et le champ montre le nom enregistré. Après un refus, il montre ce que Nadia avait tapé.

Le champ caché `_method`

Affichez le code source de la page avec Ctrl+U. Sous la balise `<form>`, `@csrf` et `@method('PUT')` ont écrit deux champs cachés :

```xml
<form method="POST" action="http://friterie-26.test/produits/2" class="…">
    <input type="hidden" name="_token" value="K5vj…" autocomplete="off">            <input type="hidden" name="_method" value="PUT">
```

Le formulaire part en `POST`, le seul de ses deux verbes qui envoie les champs hors de l'adresse. Avant de chercher la route, Laravel regarde si une requête `POST` contient un champ `_method`. Si oui, il prend sa valeur comme verbe, puis cherche dans `routes/web.php` une route `PUT /produits/{product}`. La directive `@method('PUT')` ne fait qu'écrire ce champ.



Enregistrer la modification

La méthode `update()` reçoit la requête et le produit. Elle valide les champs comme `store()`, puis modifie la ligne :

```php
public function update(Request request,Productrequest, Product request,Productproduct):
RedirectResponse
{
    validated=validated = validated=request→validate([
        'name' ⇒ ['required', 'string', 'max:60'],
        'category_id' ⇒ ['required', 'integer', 'exists:categories,id'],
        'price' ⇒ ['required', 'numeric', 'min:0.5', 'max:50'],
        'description' ⇒ ['nullable', 'string', 'max:500'],
    ]);

    product->update(product→update(product->update(validated);

    return redirect()→route('products.show', $product→id)→with('status', 'Le produit a
été modifié.');
}
```

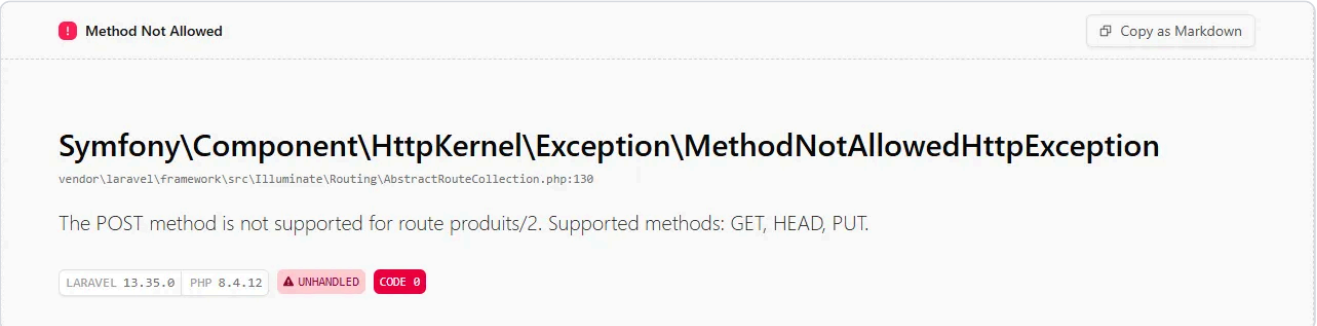
C'est le `update()` du chapitre 10, appelé cette fois par un formulaire : Nadia change un prix sans toucher au PHP. Tapez `4.20` dans le prix de la Grande frite et enregistrez. La fiche s'ouvre sous l'encadré vert, et la carte suit :

```
Le produit a été modifié.
4,20 €
```

Un prix de `0` renvoie au formulaire avec « La valeur de prix doit être supérieure ou égale à 0.5. », et le champ garde le `0` tapé.

Retirez maintenant la ligne `@method('PUT')` et envoyez de nouveau le formulaire. Laravel répond par le code 405, comme pour `DELETE /livres` au chapitre 7 :

```
Symfony\Component\HttpKernel\Exception\MethodNotAllowedHttpException
The POST method is not supported for route produits/2. Supported methods: GET, HEAD, PUT.
```



L'adresse `/produits/2` n'a pas de route `POST`. Remettez la ligne. Pour une modification, HTTP propose deux verbes : **`PUT`** remplace toute la ressource, **`PATCH`** n'en change qu'une partie. Le formulaire envoie tous les champs de la fiche, et `update()` les remplace tous : c'est un `PUT`. La rupture, qui ne touche qu'une colonne, aura sa propre route `PATCH`.

Il manque un chemin vers ce formulaire. Dans `products/show.blade.php`, sous le bloc des allergènes, ajoutez un bouton « Modifier ». La prop `href` du composant `button` en fait un lien :

```xml
<div class="mt-8 flex flex-wrap gap-3 border-t border-zinc-200 pt-6 dark:border-zinc-800">
    <x-button variant="secondary" :href="route('products.edit', $product->id)">Modifier</x-button>
</div>
```

3. Les règles écrites une fois

`store()` et `update()` répètent les mêmes quatre règles. Le jour où Nadia accepte des noms de 80 caractères, il faudra penser aux deux. Laravel peut ranger ces règles dans une classe à part :

```bash
php artisan make:request ProductRequest
```

```
   INFO  Request [app/Http/Requests/ProductRequest.php] created successfully.
```

```php
<?php

namespace App\Http\Requests;

use Illuminate\Contracts\Validation\ValidationRule;
use Illuminate\Foundation\Http\FormRequest;

class ProductRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize(): bool
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array<string, ValidationRule|array<mixed>|string>
     */
    public function rules(): array
    {
        return [
            //
        ];
    }
}
```

Copiez le tableau de `validate()` dans le `return` de `rules()`. Dans `ProductController`, importez la classe, puis changez le type du paramètre `$request` de `store()` et d'`update()` :

```php
use App\Http\Requests\ProductRequest;
```

```php
public function update(ProductRequest request,Productrequest, Product
request,Productproduct): RedirectResponse
{
    product->update(product→update(product->update(request→validated());

    return redirect()→route('products.show', $product→id)→with('status', 'Le produit a
été modifié.');
}
```

Faites de même dans `store()`, avec `Product::create($request→validated())`, et supprimez l'import de `Request`, qui ne sert plus. Enregistrez la Grande frite :

```
403 | Cette action n'est pas autorisée.
```

Laravel répond par le code 403, « interdit » : il a appelé `authorize()` avant tout le reste, et la méthode a répondu `false`. Elle sert à vérifier que la personne connectée a le droit d'envoyer ce formulaire. Le site de Nadia n'a pas de comptes, et l'autorisation est au programme du cours « Projet Web Dynamique ». Faites renvoyer `true` à `authorize()`, puis réessayez : la fiche affiche « Le produit a été modifié. ».

Une classe comme `ProductRequest` s'appelle une **Form Request** : elle contient les règles d'un formulaire. Laravel voit le type `ProductRequest` dans la signature, crée l'objet, appelle `authorize()` puis valide les champs avec `rules()`. Il n'appelle `update()` que si tout passe. Sinon, il renvoie au formulaire avec les erreurs, comme au chapitre 15, section 3. `$request->validated()` donne le tableau des champs validés, comme `$validated`. **Une règle de validation s'écrit une fois, dans la Form Request, et sert à chaque méthode qui la reçoit.**

4. Créer depuis la catégorie

Le samedi, Nadia ajoute souvent une sauce. Sur `/categories/3`, la page des sauces, un formulaire court lui suffirait : la catégorie est déjà dans l'adresse. Remplacez d'abord `{id}` par `{category}` sur la route `categories.show` et retirez son `whereNumber('id')`, comme pour `products.show` au chapitre 15. Ajoutez ensuite la route qui reçoit ce formulaire :

```php
Route::get('/categories/{category}', [CategoryController::class, 'show'])
->name('categories.show');
Route::post('/categories/{category}/produits', [CategoryController::class, 'storeProduct'])
->name('categories.products.store');
```

Dans `CategoryController`, `show()` reçoit maintenant `Category $category` et perd sa ligne `findOrFail`. Importez `ProductRequest` et `RedirectResponse`, puis ajoutez :

```php
public function storeProduct(ProductRequest request,Categoryrequest, Category
request,Categorycategory): RedirectResponse
{
    category->products()->create(category->products()->create(category->products()
->create(request->validated());

    return redirect()->route('categories.show', $category->id)->with('status', 'Le produit a
été ajouté.');
}
```

`products()`, avec parenthèses, renvoie la relation du chapitre 11. Sa méthode `create()` enregistre un produit et remplit elle-même `category_id` avec l'`id` de la catégorie. Dans `categories/show.blade.php`, sous la liste des produits, ajoutez le formulaire, sans `<select>` :

```xml
<form method="POST" action="{{ route('categories.products.store', category->id) }}"
class="mt-12 max-w-2xl space-y-6 rounded-xl border border-zinc-200 bg-white p-6 shadow-xs
sm:p-8 dark:border-zinc-800 dark:bg-zinc-900">
    @csrf

    <h2 class="text-xl font-semibold tracking-tight">Ajouter un produit dans cette
catégorie</h2>

    <div>
        <x-label for="name">Nom</x-label>
        <x-input name="name" :value="old('name')" />
        @error('name')
            <p id="name-error" class="mt-2 text-sm text-red-600 dark:text-red-400">{{
message }}</p>
        @enderror
    </div>

    <div>
        <x-label for="price">Prix (€)</x-label>
        <x-input name="price" type="number" step="0.10" min="0" :value="old('price')" />
        @error('price')
            <p id="price-error" class="mt-2 text-sm text-red-600 dark:text-red-400">{{
message }}</p>
        @enderror
    </div>

    <div class="flex justify-end">
        <x-button type="submit">Ajouter à {{ category->name }}</x-button>
    </div>
</form>
```

Tapez `Sauce cocktail` et `0.80`, puis envoyez. Le formulaire revient avec les deux valeurs, sans message et sans encadré vert. Rien n'a été enregistré : `ProductRequest` exige `category_id`, que ce formulaire n'envoie pas, et la vue n'a pas de `@error('category_id')` pour le dire.

La catégorie se trouve pourtant dans l'adresse. Avant de valider, Laravel appelle la méthode `prepareForValidation()` de la Form Request, si elle existe. Ajoutez-la dans `ProductRequest`, sous `authorize()` :

```php
protected function prepareForValidation(): void
{
    if ($this->route('category')) {
        $this->merge(['category_id' => $this->route('category')->id]);
    }
}
```

`$this->route('category')` renvoie la catégorie liée au paramètre `{category}` de l'adresse, et `null` sur une route qui n'a pas ce paramètre. Quand la catégorie existe, `merge()` ajoute son `id` aux champs de la requête, comme si le formulaire avait envoyé `category_id`.

Envoyez de nouveau la sauce cocktail : la page des sauces s'affiche sous « Le produit a été ajouté. ».

Pour voir ce que la Form Request a gardé, ajoutez `dd($request->validated());` en tête de `storeProduct()`, et envoyez une troisième fois `Sauce cocktail` et `0.80` :

```
array:3 [▼ // app/Http/Controllers/CategoryController.php:24
  "name" => "Sauce cocktail"
  "category_id" => 3
  "price" => "0.80"
]
```

`category_id` vaut le nombre `3`, sans guillemets, car il vient de l'adresse et non d'un champ du formulaire. Retirez le `dd()`. Dans SQLite Viewer, la table `products` montre le nouveau produit, colonnes de dates omises :

```
id | category_id | name           | price | description
18 | 3           | Sauce cocktail | 0.8   |
```

Le formulaire d'ajout n'a pas de catégorie dans son adresse, et `category_id` y reste obligatoire. Sur `/produits/nouveau`, supprimez la balise `<select>` dans les outils de développement, tapez un nom et un prix, puis envoyez. Le formulaire revient avec « Le champ catégorie est obligatoire. », et rien n'est enregistré.

5. Les allergènes cochés

La fiche de la viandelle affiche « Aucun allergène déclaré. ». Nadia doit pouvoir cocher ses allergènes, à l'ajout comme à la modification. Plutôt que d'écrire les cases dans les deux vues, rangez d'abord les champs communs dans un composant anonyme, comme `<x-alert>` au chapitre 9, section 5.

Les champs écrits une fois

BASH

```
php artisan make:component product-fields --view
```

```
   INFO  View [resources/views/components/product-fields.blade.php] created successfully.
```

Remplacez le contenu de `product-fields.blade.php` par les quatre `<div>` des champs de `edit.blade.php`, sans la balise `<form>` ni le bouton. Ajoutez en tête la directive `@props` et un bloc `@php`, puis les cases sous la description :

```xml
@props(['product', 'categories', 'allergens'])

@php
    checkedAllergens=session()->hasOldInput()?(array)old('allergens',[]):checkedAllergens =
session()->hasOldInput()
        ? (array) old('allergens', [])
        : checkedAllergens=session()->hasOldInput()?(array)old('allergens',[]):product
->allergens->pluck('id')->all();
@endphp

<div>
    <x-label for="name">Nom</x-label>
    <x-input name="name" :value="old('name', product->name)" />
    @error('name')
        <p id="name-error" class="mt-2 text-sm text-red-600 dark:text-red-400">{{ $message
}}</p>
    @enderror
</div>

<div>
    <x-label for="category_id">Catégorie</x-label>
    <x-select name="category_id">
        <option value="">Choisissez une catégorie</option>
        @foreach (categories as category)
            <option value="{{ $category->id }}" @selected(old('category_id', product
>category_id) == category->id)>{{ $category->name }}</option>
        @endforeach
    </x-select>
    @error('category_id')
        <p id="category_id-error" class="mt-2 text-sm text-red-600 dark:text-red-400">{{
$message }}</p>
    @enderror
</div>

<div>
    <x-label for="price">Prix (€)</x-label>
    <x-input name="price" type="number" step="0.10" min="0" :value="old('price', product
->price)" />
    @error('price')
        <p id="price-error" class="mt-2 text-sm text-red-600 dark:text-red-400">{{ message
}}</p>
    @enderror
</div>

<div>
    <x-label for="description">Description</x-label>
    <x-textarea name="description">{{ old('description', product->description) }}</x-
textarea>
    @error('description')
        <p id="description-error" class="mt-2 text-sm text-red-600 dark:text-red-400">{{
message }}</p>
    @enderror
</div>

<fieldset>
```

```
    <legend class="text-sm font-medium">Allergènes</legend>
    <div class="mt-2 grid grid-cols-2 gap-2 sm:grid-cols-4">
        @foreach (allergens as allergen)
            <label class="flex items-center gap-2 text-sm">
                <input type="checkbox" name="allergens[]" value="{{ $allergen->id }}"
@checked(in_array(allergen->id, $checkedAllergens)) class="size-4 accent-brand-600">
                {{ $allergen->name }}
            </label>
        @endforeach
    </div>
    @error('allergens.*')
        <p class="mt-2 text-sm text-red-600 dark:text-red-400">{{ $message }}</p>
    @enderror
</fieldset>
```

Comme au chapitre 9, section 5, `@props` déclare ce que le composant attend : le produit, les catégories du `<select>` et les allergènes des cases. Le bloc `@php` range dans `$checkedAllergens` les `id` des allergènes à cocher. `edit.blade.php` se réduit à ceci :

XML

```
<form method="POST" action="{{ route('products.update', product->id) }}" class="…">
    @csrf
    @method('PUT')

    <x-product-fields :product="product" :categories="categories":allergens="categories"
:allergens="categories":allergens="allergens" />

    <div class="flex justify-end">
        <x-button type="submit">Enregistrer</x-button>
    </div>
</form>
```

Les deux-points transmettent les variables elles-mêmes, et non leur nom écrit en texte, comme `:concert="$concert"` au chapitre 9. `create.blade.php` se réduit de même, avec `route('products.store')`, sans `@method`, et avec le bouton « Ajouter le produit ».

Les deux vues transmettent `$allergens`. `create()` et `edit()` passent donc la liste des allergènes, et `create()` un produit vide. Importez `Allergen` dans le contrôleur :

PHP

```
return view('products.create', [
    'product' => new Product,
    'categories' => $categories,
    'allergens' => Allergen::all(),
]);
```

PHP

```
return view('products.edit', [
    'product' => product,'categories'=>product,
    'categories' => product,'categories'=>categories,
    'allergens' => Allergen::all(),
]);
```

`new Product` crée un objet absent de la base : ses colonnes valent `null` et il n'a aucun allergène. Dans le formulaire d'ajout, `old('name', $product→name)` renvoie donc `null`, comme `old('name')` au chapitre 15. Ouvrez `/produits/nouveau` puis `/produits/2/modifier` : les deux formulaires affichent les huit cases.

Ce que les cases envoient

Chaque case porte le nom `allergens[]`. Les crochets demandent à PHP de ranger toutes les valeurs cochées dans un tableau, `$request→input('allergens')`. La valeur de chaque case est l'`id` de son allergène. `pluck('id')→all()` donne le tableau des `id` des allergènes du produit, et `@checked(…)` écrit `checked` quand l'`id` de la case s'y trouve. Dans le code source de `/produits/3/modifier`, la case Gluten de la fricadelle est cochée :

```XML
<input type="checkbox" name="allergens[]" value="1" checked class="size-4 accent-brand-600">
```

Ajoutez `dd($request→all());` en tête d'`update()`. Cochez Gluten et Soja pour la viandelle, puis enregistrez. Cliquez sur la flèche d'`allergens` pour déplier le tableau :

```
array:7 [▼ // app/Http/Controllers/ProductController.php:67
  "_token" ⇒ "2H4wT0Ci…"
  "_method" ⇒ "PUT"
  "name" ⇒ "Viandelle"
  "category_id" ⇒ "2"
  "price" ⇒ "2.80"
  "description" ⇒ "Une saucisse de viande hachée, frite à la commande."
  "allergens" ⇒ array:2 [▼
    0 ⇒ "1"
    1 ⇒ "5"
  ]
]
```

Revenez au formulaire, décochez les deux cases, et enregistrez :

```
array:6 [▼ // app/Http/Controllers/ProductController.php:67
  "_token" ⇒ "2H4wT0Ci…"
  "_method" ⇒ "PUT"
  "name" ⇒ "Viandelle"
  "category_id" ⇒ "2"
  "price" ⇒ "2.80"
  "description" ⇒ "Une saucisse de viande hachée, frite à la commande."
]
```

Un navigateur n'envoie pas une case décochée. Sans case cochée, la clé `allergens` est absente de la requête. Retirez le `dd()`.

C'est pour cela que le composant ne se contente pas de `old('allergens', …)`. `session()→hasOldInput()` dit si un envoi précédent a laissé des valeurs dans la session. Après un refus, une liste absente veut dire qu'aucune case n'était cochée, et `old('allergens', [])` renvoie un tableau vide. Au

premier affichage, la session ne contient rien, et les cases reprennent les allergènes enregistrés.

Sur `/produits/3/modifier`, décochez Gluten et Soja, tapez un prix de `0`, puis enregistrez. Le formulaire revient avec le message du prix, et la case Gluten reste décochée :

```XML
<input type="checkbox" name="allergens[]" value="1"  class="size-4 accent-brand-600">
```

Avec `old('allergens', $product→allergens→pluck('id')→all())`, `old()` ne trouverait pas de clé `allergens` dans la session, et Gluten et Soja reviendraient cochés.

Reste le `(array)` devant `old()`. Les crochets d'`allergens[]` sont une convention du navigateur, et rien n'oblige à la suivre : avec `curl`, ou en renommant la case `allergens` dans les outils de développement, on envoie `allergens=1`, une valeur seule. Les règles de la section suivante la refuseront, mais le formulaire sera réaffiché, et `old('allergens', [])` renverra la chaîne `"1"`. Sans le `(array)`, `in_array()` s'arrête :

```
in_array(): Argument #2 ($haystack) must be of type array, string given
```

`(array)` convertit cette chaîne en tableau d'un élément, `["1"]`, et laisse un tableau tel quel. La case 1 revient cochée, et la page s'affiche. **Une vue ne fait pas plus confiance à la requête qu'un contrôleur** : ce que le navigateur envoie d'habitude n'est pas ce qu'il envoie toujours.

Valider et enregistrer

Ajoutez deux règles dans `ProductRequest` :

```PHP
'allergens' ⇒ ['nullable', 'array'],
'allergens.*' ⇒ ['integer', 'exists:allergens,id'],
```

La première accepte un tableau ou rien. Dans **`allergens.*`**, l'étoile désigne chaque élément du tableau : chaque valeur reçue doit être un entier, `id` d'une ligne d'`allergens`. Cochez Gluten et Soja, puis changez la valeur de la case Soja en `99` dans les outils de développement. L'envoi donne « Le champ allergens.1 sélectionné est invalide. ».

Laravel nomme chaque case par sa position dans le tableau envoyé : `allergens.0`, `allergens.1`. La méthode facultative `attributes()` d'une Form Request donne un nom aux champs de ce formulaire seulement, comme le tableau `attributes` de `lang/fr/validation.php` :

```PHP
public function attributes(): array
{
    return [
        'allergens.*' ⇒ 'allergène',
    ];
}
```

Le message devient « Le champ allergène sélectionné est invalide. ». Sa voisine `messages()` remplace le message entier d'une règle. Avec `'name.required' ⇒ 'Donnez un nom au produit.'` dans son tableau, un nom vide affiche « Donnez un nom au produit. ».

Dans `update()`, enregistrez ensuite la liste cochée :

```php
validated=validated = validated=request→validated();

product->update(product→update(product->update(validated);
product->allergens()->sync(product→allergens()→sync(product->allergens()
->sync(validated['allergens'] ?? []);
```

Vous connaissez `sync()` depuis le chapitre 12, section 4 : il rend les lignes du produit dans `allergen_product` identiques à la liste reçue. `?? []` lui donne un tableau vide quand aucune case n'est cochée, et `sync([])` retire alors tous les allergènes. `update()` ignore la clé `allergens`, absente de `[Fillable]`, comme au chapitre 10. Faites le même ajout dans `store()` : rangez `$request→validated()` dans `$validated`, passez-le à `Product::create()`, puis appelez `sync()` sur `$product`.

Affichez les lignes de la viandelle dans la table pivot, avant puis après avoir coché et enregistré Gluten et Soja :

```bash
php artisan tinker --execute="echo DB::table('allergen_product')→where('product_id', 17)
→get();"
```

```
[]
```

```
[{"allergen_id":1,"product_id":17},{"allergen_id":5,"product_id":17}]
```

6. Marquer en rupture

Un samedi soir, les croquettes de fromage manquent. Nadia ne veut pas les retirer de la carte, seulement les marquer « En rupture ». La colonne `sold_out_at` existe déjà : le badge s'affiche dès qu'elle contient une date. L'action ne change qu'une colonne du produit : c'est un `PATCH`.

```php
Route::patch('/produits/{product}/rupture', [ProductController::class, 'soldOut'])
→name('products.sold-out');
```

```php
public function soldOut(Product $product): RedirectResponse
{
    if (! $product->isSoldOut()) {
        $product->update(['sold_out_at' => now()]);
    }

    return redirect()->route('products.show', $product->id)->with('status', 'Le produit est marqué en rupture.');
}
```

`now()` renvoie la date et l'heure actuelles. Le `if` garde la date de la première rupture : un second envoi, depuis une fiche restée ouverte dans un autre onglet, ne la change pas. Sur la fiche, à côté de « Modifier », ajoutez un formulaire d'un seul bouton, affiché seulement pour un produit disponible :

```xml
@if (! $product->isSoldOut())
    <form method="POST" action="{{ route('products.sold-out', $product->id) }}">
        @csrf
        @method('PATCH')
        <x-button type="submit" variant="secondary">Marquer en rupture</x-button>
    </form>
@endif
```

Le formulaire n'a pas de champ visible : l'adresse désigne le produit, et le verbe `PATCH` dit ce qu'il faut en faire. Un lien aurait été plus court, mais un robot qui le suit marquerait le produit en rupture : c'est la réponse du code qui sent mauvais du chapitre 15. Sur `/produits/5`, cliquez sur le bouton :

```
Le produit est marqué en rupture.
Croquette de fromage
En rupture
En rupture depuis 07/10/2026, 15 h 22
```

Le bouton a disparu, et la carte affiche le badge rouge sous les croquettes.

7. Retirer de la carte

Nadia ne vend plus de sauce tartare. Pour supprimer, HTTP propose le verbe `DELETE`. La route reprend l'adresse de `update()`, avec un autre verbe :

```php
Route::delete('/produits/{product}', [ProductController::class, 'destroy'])
->name('products.destroy');
```

```php
public function destroy(Product $product): RedirectResponse
{
    $product->delete();

    return redirect()->route('products.index')->with('status', 'Le produit a été retiré de la carte.');
}
```

La fiche n'existe plus après la suppression : la redirection mène à la carte. Sur la fiche, dans le `<div>` des boutons, après le formulaire de rupture, ajoutez :

```xml
<form method="POST" action="{{ route('products.destroy', $product->id) }}" onsubmit="return confirm('Retirer ce produit de la carte ?')">
    @csrf
    @method('DELETE')
    <x-button type="submit" variant="danger">Retirer de la carte</x-button>
</form>
```

L'attribut `onsubmit` contient du JavaScript, exécuté juste avant l'envoi. `confirm()` ouvre une boîte du navigateur avec la question, un bouton OK et un bouton Annuler. Elle renvoie `false` sur Annuler, et `return false` arrête l'envoi. `confirm()` évite un clic malheureux, mais ne protège rien côté serveur : une requête envoyée sans le formulaire ne passe par aucune boîte. Une boîte mise en forme avec Tailwind est un bonus du chapitre 17.

Comptez les lignes de la table pivot, retirez la Tartare, puis comptez de nouveau :

```bash
php artisan tinker --execute="echo DB::table('allergen_product')->count();"
```

```
22
```

```
20
```

La carte s'affiche sous « Le produit a été retiré de la carte. », sans la Tartare, et `/produits/12` répond 404. La Tartare avait deux allergènes, Œuf et Moutarde. Ses deux lignes ont disparu d'`allergen_product`, grâce au `cascadeOnDelete()` de la migration, comme au chapitre 12, section 7.

8. Les routes et `Route::resource()`

Le menu

Dans le menu du layout, à côté de « La carte », ajoutez un lien vers le formulaire d'ajout. Restreignez aussi la condition `active` de « La carte », écrite avec `routeIs()` comme au chapitre 13, section 6, pour qu'un seul lien soit marqué à la fois :

```xml
 <li><x-nav-link :href="route('products.index')" :active="request()
->routeIs('products.index', 'products.show', 'categories.show')">La carte</x-nav-link></li>
 <li><x-nav-link :href="route('products.create')" :active="request()
->routeIs('products.create')">Ajouter un produit</x-nav-link></li>
```

Neuf routes écrites à la main

Voici les routes des produits, écrites à la main :

```bash
php artisan route:list --path=produits
```

```
  POST      categories/{category}/produits ..... categories.products.store ›
CategoryController@storeProduct
  POST      produits ............................................ products.store ›
ProductController@store
  GET|HEAD  produits/nouveau ..................................... products.create ›
ProductController@create
  GET|HEAD  produits/{product} ...................................... products.show ›
ProductController@show
  PUT       produits/{product} ................................. products.update ›
ProductController@update
  DELETE    produits/{product} ................................ products.destroy ›
ProductController@destroy
  GET|HEAD  produits/{product}/modifier ............................ products.edit ›
ProductController@edit
  PATCH     produits/{product}/rupture ....................... products.sold-out ›
ProductController@soldOut


Showing [8] routes
```

Avec `products.index` sur `/carte`, `ProductController` donne un corps aux sept méthodes que `RecipeController --resource` laissait vides au chapitre 8, section 9 : `index`, `create`, `store`, `show`, `edit`, `update` et `destroy`. Ajoutez temporairement cette ligne à la fin de `routes/web.php`, puis listez les routes qu'elle déclare :

```php
Route::resource('produits', ProductController::class);
```

```bash
php artisan route:list --name=produits
```

```
  POST            produits ............................. produits.store ›
ProductController@store
  GET|HEAD        produits ............................. produits.index ›
ProductController@index
  GET|HEAD        produits/create ...................... produits.create ›
ProductController@create
  GET|HEAD        produits/{produit} ....................... produits.show ›
ProductController@show
  PUT|PATCH       produits/{produit} .................. produits.update ›
ProductController@update
  DELETE          produits/{produit} ................ produits.destroy ›
ProductController@destroy
  GET|HEAD        produits/{produit}/edit ................. produits.edit ›
ProductController@edit

                                                                  Showing [7]
routes
```

Ces sept routes ne sont pas les vôtres. Les adresses finissent par `create` et `edit`, en anglais, et les noms commencent par `produits.`. Le paramètre `{produit}` ne porte pas le nom de l'argument `$product` : Laravel ne fait pas la liaison de modèle, `show()` reçoit un produit vide, et la fiche s'arrête sur `Attempt to read property "id" on null`.

`Route::resource()` peut pourtant parler français. Dans `app/Providers/AppServiceProvider.php`, importez `Illuminate\Support\Facades\Route`, puis ajoutez cette ligne dans `boot()` :

```php
Route::resourceVerbs(['create' => 'nouveau', 'edit' => 'modifier']);
```

Relancez `php artisan route:list --name=produits`. Les deux adresses traduites deviennent :

```
  GET|HEAD        produits/nouveau ..................... produits.create ›
ProductController@create
  GET|HEAD        produits/{produit}/modifier .............. produits.edit ›
ProductController@edit
```

Ajoutés à `Route::resource()`, `->names('products')` et `->parameters(['produits' => 'product'])` corrigeraient aussi les noms et le paramètre. `/produits/{product}/rupture` et `/categories/{category}/produits` resteraient pourtant hors de `Route::resource()`, écrites à la main de toute façon. Le cours garde donc ses neuf routes écrites à la main : chaque ligne de `routes/web.php` se lit sans rien chercher ailleurs. Retirez la ligne `Route::resource()`, puis celle de `resourceVerbs()` et son import.

À retenir

- Un formulaire de modification se pré-remplit avec `old('name', $product->name)` : la valeur tapée après un refus, sinon la valeur enregistrée.

- Un formulaire HTML n'envoie que `GET` ou `POST`, et `@method('PUT')` ajoute le champ caché `_method` que Laravel lit pour choisir la route.
- Une Form Request range les règles d'un formulaire, et Laravel valide avant d'appeler la méthode qui la reçoit.
- Un navigateur n'envoie pas une case décochée, et `sync($validated['allergens'] ?? [])` remplace la liste entière, vide comprise.
- Tout ce qui modifie la base part d'un formulaire `POST` avec `@csrf`, et une suppression demande confirmation avec `confirm()`.

Exercices

Comptez 15 minutes pour cet exercice sur papier. Les katas du refuge, au chapitre 17, viennent ensuite : ils reprennent tout ce que les chapitres 15 et 16 ont montré sur la friterie.

16.1 Qui reçoit la requête ? (sur papier, 15 minutes)

Relisez le `route:list` de la section 8. Pour chaque requête, écrite comme le navigateur l'envoie, dites quelle méthode de quel contrôleur la reçoit. Si aucune ne la reçoit, donnez le code de la réponse.

```
1. GET  /produits/4/modifier
2. POST /produits/4            avec _method=PUT
3. POST /produits/4            avec _method=PATCH
4. POST /produits/4/rupture    avec _method=PATCH
5. GET  /produits/4/rupture
6. POST /categories/2/produits
7. GET  /produits/modifier
```

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/formulaires-modifier-et-supprimer

Six cartes

Que fait `@method('PUT')` ?

Il écrit un champ caché `_method` qui vaut `PUT`. Le formulaire part en `POST`, et Laravel cherche une route `PUT`.

Pourquoi `old('name', $product→name)` ?

Après un refus, le champ montre ce qui avait été tapé. Sinon, il montre la valeur enregistrée.

Une case décochée, que reçoit le serveur ?

Rien. Sans case cochée, la clé `allergens` est absente de la requête.

`sync([])` ou `detach()` ?

Les deux retirent tous les allergènes du produit. `sync()` reçoit la liste cochée, vide ou non, et `update()` n'a pas besoin d'un `if`.

`PUT` ou `PATCH` ?

`PUT` remplace toute la ressource, comme le formulaire de modification. `PATCH` en change une partie, comme la rupture.

Pourquoi un formulaire pour supprimer ?

Un lien envoie un `GET`, qu'un robot ou le préchargement peut suivre. Un formulaire `POST` avec `@csrf` et `@method('DELETE')` ne part que sur un clic.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/formulaires-modifier-et-supprimer

3.17 Le refuge : exercices

Bloc Laravel · séance 9 · ~3 h

Trois refuges du Brabant wallon, à Wavre, Rixensart et Ottignies, publient sur un même site les animaux qu'ils proposent à l'adoption. Les bénévoles y ajoutent un animal, corrigent sa fiche, cochent ses traits de caractère, le marquent adopté et suppriment une fiche créée par erreur. Le site n'a pas de compte : l'authentification arrive au cours suivant. Dans ce chapitre, vous écrivez ce site en huit katas, avec ce que la friterie vous a appris aux chapitres 15 et 16.

À la fin de ce chapitre, vous serez capables de :

1. **écrire** les routes, les contrôleurs et les vues d'un site, à partir d'un tableau de routes imposées,

2. **lire** un test rouge, et **trouver** ce qui manque dans votre code,
3. **valider** un formulaire, et **afficher** ses erreurs en français sous chaque champ,
4. **enregistrer** des cases à cocher, une adoption et une suppression depuis des formulaires,
5. **vérifier** la mise en forme de chaque page avec une liste de critères.

1. Le refuge

Installer le dépôt (10 minutes, une seule fois)

Sur github.com/opmvpc/refuge-26, créez votre copie avec le bouton Fork. Ouvrez ensuite un terminal dans le dossier de Herd ou de Laragon, et préparez le projet :

```bash
git clone https://github.com/VOTRE-COMPTE/refuge-26.git
cd refuge-26
composer install
cp .env.example .env
php artisan key:generate
php artisan migrate --seed
npm install
composer run dev
```

Ce sont les commandes de la friterie, au chapitre 15. Si `migrate` propose de créer `database/database.sqlite`, appuyez sur Entrée pour répondre `yes`. La base reçoit trois refuges, huit traits de caractère et douze animaux, dont deux adoptés, avec des dates fixes : vos pages ressemblent aux captures. Laissez `composer run dev` tourner. Avec Laragon, rechargez d'abord Laragon depuis son menu, puis ouvrez `http://refuge-26.test`. Au départ, seule la page `/composants` s'affiche, car les autres routes sont à écrire.

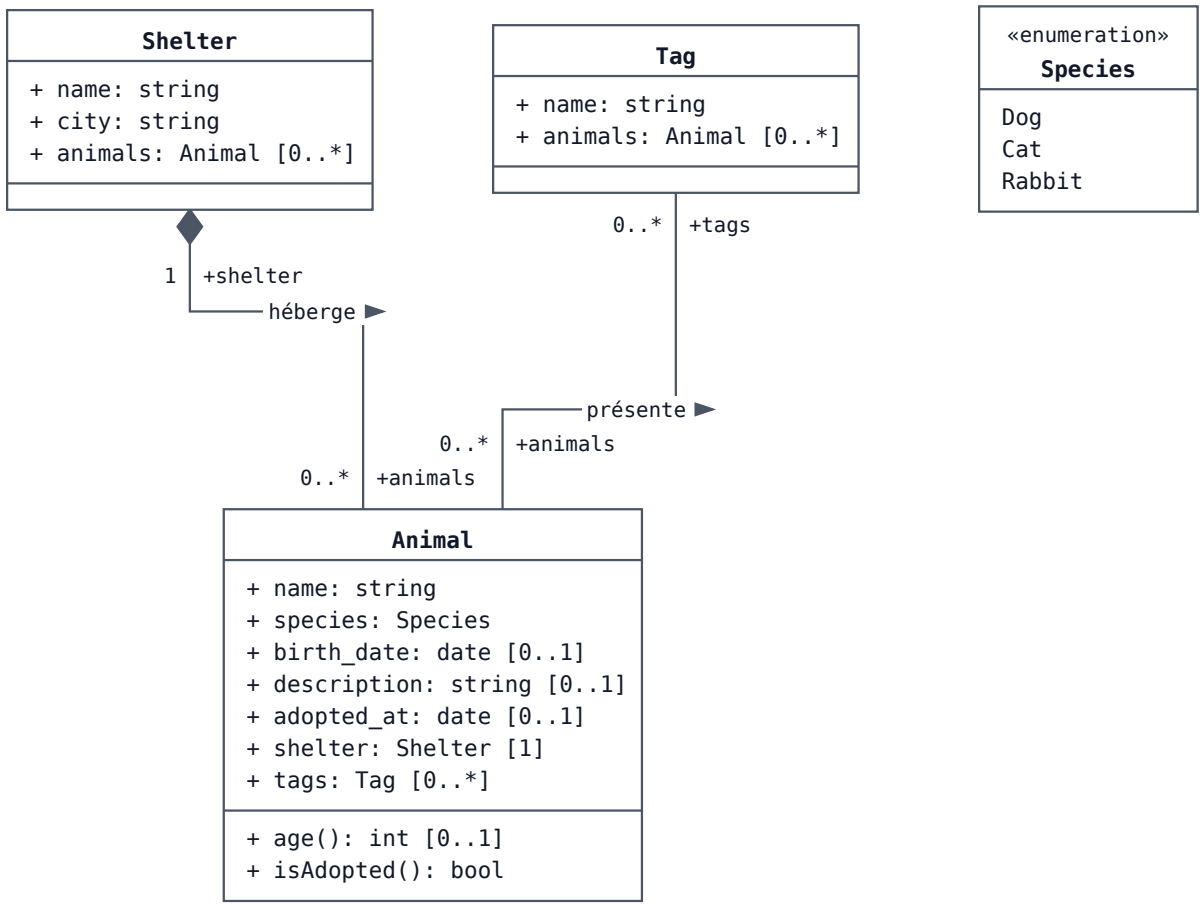
Ce que le dépôt contient

Déjà dans le dépôt	Ce que vous écrivez
les migrations des tables `shelters`, `tags`, `animals` et `animal_tag`	les routes de `routes/web.php`, sauf la redirection de `/` vers `/animaux` et la route `styleguide`
l'enum `App\Enums\Species` et les modèles `Shelter`, `Animal` et `Tag`, avec leurs relations, `#[Fillable]` et `casts()`	`AnimalController` et `ShelterController`
les fabriques, et les seeders des trois refuges, des huit traits et des douze animaux	les vues `animals/index`, `show`, `create`, `edit` et `shelters/show`
le layout vert mousse `<x-layouts.app>`, les composants `<x-button>`, `<x-badge>`, `<x-card>` et `<x-nav-link>`, la page `/composants`	les règles de validation, dans `validate()` ou dans une Form Request `AnimalRequest`
les composants de champ de la friterie : `<x-label>`, `<x-input>`, `<x-select>` et `<x-textarea>`	les traductions françaises, avec le paquet Laravel-Lang

Déjà dans le dépôt	Ce que vous écrivez
les pictogrammes `public/images/species/chien.svg`, `chat.svg` et `lapin.svg`	l'affichage du message flash dans le layout
les tests des huit katas dans `tests/Feature/`, et la CI dans `.github/workflows/tests.yml`	

La page `/composants` montre chaque composant dans ses variantes, avec ses props. Gardez-la ouverte : `<x-button>` y accepte une prop `href`, qui en fait un lien en forme de bouton, et la prop `variant` de `<x-badge>` vaut `adopted`, `species` ou `tag`.

Le schéma



Le losange plein ◆——— correspond à la clé étrangère `shelter_id`, qui supprime en cascade les animaux d'un refuge. La table pivot `animal_tag` relie les animaux et leurs traits, comme `allergen_product` à la friterie. La base enregistre l'espèce en texte, `chien`, `chat` ou `lapin`. `$animal->species->label()` affiche « Chien », et `$animal->species->image()` donne le chemin du pictogramme.

Le modèle des traits de caractère s'appelle `Tag`, et non `Trait`. `trait` est le mot-clé des traits du chapitre 7 de la POO, et `class Trait {}` s'arrête sur `Parse error: syntax error, unexpected token "trait", expecting identifier`. Dans les textes affichés, écrivez « trait de caractère ».

Le menu écrit en dur

Le layout écrit les adresses du menu à la main, sans `route()` :

```xml
<li><x-nav-link href="/animaux" :active="request()->is('animaux')">Les animaux</x-nav-link>
</li>
<li><x-nav-link href="/animaux/nouveau" :active="request()->is('animaux/nouveau')">Ajouter
un animal</x-nav-link></li>
```

Tant que la route `animals.index` n'existe pas, `route('animals.index')` lève une exception. Écrite dans le layout, elle ferait planter toutes les pages, `/composants` compris. `request()->is('animaux')` compare l'adresse de la page ouverte, là où `request()->routeIs()` du chapitre 13, section 6, compare le nom de la route.

2. Les routes imposées

Les tests envoient des requêtes sur ces adresses, avec ces verbes, et vérifient ces noms. Dans `routes/web.php`, le paramètre s'écrit `{animal}` avec la liaison de modèle du chapitre 15, section 6, ou `{id}` avec `findOrFail()`, comme au chapitre 10. Déclarez `/animaux/nouveau` avant `/animaux/{animal}`, ou posez `whereNumber` sur le paramètre, pour éviter le piège de l'ordre du chapitre 7.

Ce tableau décrit le site terminé. Le kata 17.2 crée un animal sans validation, le kata 17.3 ajoute la validation, et le kata 17.5 les cases des traits. Lisez ce tableau avant le kata 17.1. Les deux suivants servent aux katas 17.3 et 17.5, qui y renvoient. Le dernier se relit à chaque kata.

Verbe	Adresse	Nom	Méthode	Effet attendu
`GET`	`/animaux`	`animals.index`	`AnimalController@index`	les animaux **non adoptés**, triés par nom ; bonus : `?espece=chat` filtre, valeur inconnue ignorée
`GET`	`/animaux/nouveau`	`animals.create`	`create`	le formulaire d'ajout, avec le `<select>` des refuges et les cases des traits
`POST`	`/animaux`	`animals.store`	`store`	valide, crée, `sync` les traits, redirige vers la fiche, flash `status` = « L'animal a été ajouté. »

Verbe	Adresse	Nom	Méthode	Effet attendu
`GET`	`/animaux/{id}`	`animals.show`	`show`	nom, espèce (libellé), refuge, âge, traits, badge « Adopté » le cas échéant, boutons Modifier, Adopter (si non adopté), Supprimer ; 404 sinon
`GET`	`/animaux/{id}/modifier`	`animals.edit`	`edit`	le formulaire pré-rempli, traits pré-cochés, `@method('PUT')`
`PUT`	`/animaux/{id}`	`animals.update`	`update`	valide, met à jour, `sync`, redirige vers la fiche, flash « L'animal a été modifié. »
`PATCH`	`/animaux/{id}/adoption`	`animals.adopt`	`adopt`	pose `adopted_at` à aujourd'hui si nul (ne change rien sinon), redirige vers la fiche, flash « {nom} a trouvé une famille. »
`DELETE`	`/animaux/{id}`	`animals.destroy`	`destroy`	supprime, redirige vers la liste, flash « L'animal a été supprimé. »
`GET`	`/refuges/{id}`	`shelters.show`	`ShelterController@show`	bonus : le refuge, sa ville, ses animaux à adopter

Les champs portent les mêmes noms à l'ajout et à la modification :

Champ	Élément	Règles
`name`	`<input type="text">`	`required`, `string`, `max:50`
`species`	`<select>` sur `Species::cases()`, `value="{{ $species->value }}"`	`required`, `Rule::enum(Species::class)`
`shelter_id`	`<select>` sur les refuges	`required`, `integer`, `exists:shelters,id`
`birth_date`	`<input type="date">`	`nullable`, `date`, `before_or_equal:today`
`description`	`<textarea>`	`nullable`, `string`, `max:1000`
`tags`	cases `name="tags[]"` `value="{{ $tag->id }}"`	`nullable`, `array` ; `tags.*` : `integer`, `exists:tags,id`

Dans les messages d'erreur, chaque champ porte un nom français :

Champ	Nom dans les messages
`name`	nom
`species`	espèce
`shelter_id`	refuge
`birth_date`	date de naissance
`description`	description
`tags`	traits de caractère
`tags.*`	trait de caractère

Les tests cherchent ces textes dans les pages ou dans la session :

Texte	Quand
« L'animal a été ajouté. »	message flash, après l'ajout
« L'animal a été modifié. »	message flash, après la modification
« Rex a trouvé une famille. », avec le nom de l'animal adopté	message flash, après l'adoption
« L'animal a été supprimé. »	message flash, après la suppression
« Adopté »	badge, sur la fiche d'un animal adopté
« Aucun animal »	liste vide
« Chien », « Chat », « Lapin »	libellé de l'espèce, sur la fiche

Dans le HTML des formulaires, les tests cherchent aussi `method="POST"` et l'attribut `action` écrit avec `route()`, entre guillemets doubles, comme dans les tableaux des katas. Le reste du HTML, les classes Tailwind, l'ordre des champs et les autres textes sont libres. Les critères de chaque kata décrivent la mise en forme attendue.

3. Afficher et ajouter

Les katas 17.1 à 17.3 affichent les animaux, puis ajoutent un animal avec un formulaire validé. Ils reprennent le chapitre 15 : gardez-le ouvert à côté.

Kata	Groupe	À relire
17.1 La liste et la fiche (20 minutes)	`liste`	`ProductController` de la friterie (chapitre 15, section 1), la liaison de modèle (chapitre 15, section 6), la carte et sa grille (exercice 14.3)
17.2 Ajouter un animal (25 minutes)	`ajouter`	chapitre 15, sections 2 et 5

Kata	Groupe	À relire
17.3 Valider le formulaire (25 minutes, installation comprise)	`valider`	chapitre 15, sections 3 et 4

Lire un kata

Un kata est un groupe de tests Pest, dans un fichier de `tests/Feature/`. Les tests sont l'énoncé : chacun porte une phrase qui dit ce qu'il attend, et le tableau de chaque kata reprend ces phrases mot pour mot. Lancez le groupe avant d'écrire la moindre ligne :

```bash
php artisan test --group=liste
```

```
   FAIL  Tests\Feature\ListeTest
  × la route animals.index répond à GET /animaux
0.77s
  × la liste affiche les animaux à adopter, triés par nom
0.21s
  × la liste ne montre pas les animaux adoptés
0.04s
  × une liste vide affiche « Aucun animal »
0.03s
  × la fiche affiche le nom, l'espèce, le refuge et les traits de caractère
0.04s
  × la fiche d'un animal adopté affiche « Adopté »
0.03s
  ✓ la fiche d'un animal inconnu répond 404
0.03s
…
```

Sous la liste, Pest détaille chaque échec : la phrase, la raison, puis la ligne du test. Quatre raisons reviennent souvent :

```
   FAILED  Tests\Feature\ListeTest > la route animals.index répond à GET /animaux
  La route animals.index n'existe pas.
Failed asserting that false is true.
```

La route manque, ou elle porte un autre nom. La première ligne vient du test, la seconde de Pest.

```
   FAILED  Tests\Feature\ListeTest > la liste affiche les animaux à adopter, triés par nom
  Expected response status code [200] but received 404.
Failed asserting that 404 is identical to 200.
```

Aucune route ne répond à `GET /animaux`.

```
   FAILED  Tests\Feature\ModifierTest > envoyer le formulaire de mod…
RouteNotFoundException
  Route [animals.show] not defined.
```

Le test appelle `route('animals.show')` pour construire l'adresse de la redirection attendue. La route d'un kata précédent manque : écrivez d'abord celle du kata 17.1.

```
   FAILED  Tests\Feature\ValiderTest > les traductions françaises sont installées
  Lancez composer require laravel-lang/common --dev, puis php artisan lang:add fr, et faites
un commit de lang/.
Failed asserting that false is true.
```

Le message dit quoi faire. Pour le reste, ouvrez le fichier du test : il montre les données créées, l'adresse demandée et le texte cherché. Lancé depuis un assistant IA, `php artisan test` peut répondre en JSON, avec les mêmes phrases et les mêmes messages. Pour relancer un seul test, filtrez sur un morceau de sa phrase :

```bash
php artisan test --filter="triés par nom"
```

Six tests sur 41 sont verts dès le départ : les cinq « … répond 404 », et « adopter un animal déjà adopté ne change pas sa date d'adoption ». Ils doivent le rester quand vous écrivez les routes. Avant chaque commit, lancez `php artisan test` sans groupe : les katas 17.4 et 17.5 modifient `store()`, et peuvent faire repasser au rouge les groupes `ajouter` et `valider`. Un groupe vert valide ce que ses tests contrôlent. Les couleurs, les champs pré-remplis à l'écran et les boîtes de confirmation se vérifient dans le navigateur, avec les critères de chaque kata. Une largeur, le clavier et un attribut se vérifient comme au chapitre 14, section 1 : F12 puis Ctrl+Maj+M, la touche Tab, l'onglet Éléments.

Dès qu'un groupe est vert, faites un commit et poussez. La première fois, l'onglet Actions de votre fork demande d'activer les workflows, comme au chapitre 2 de la POO. Il affiche ensuite une coche par kata.

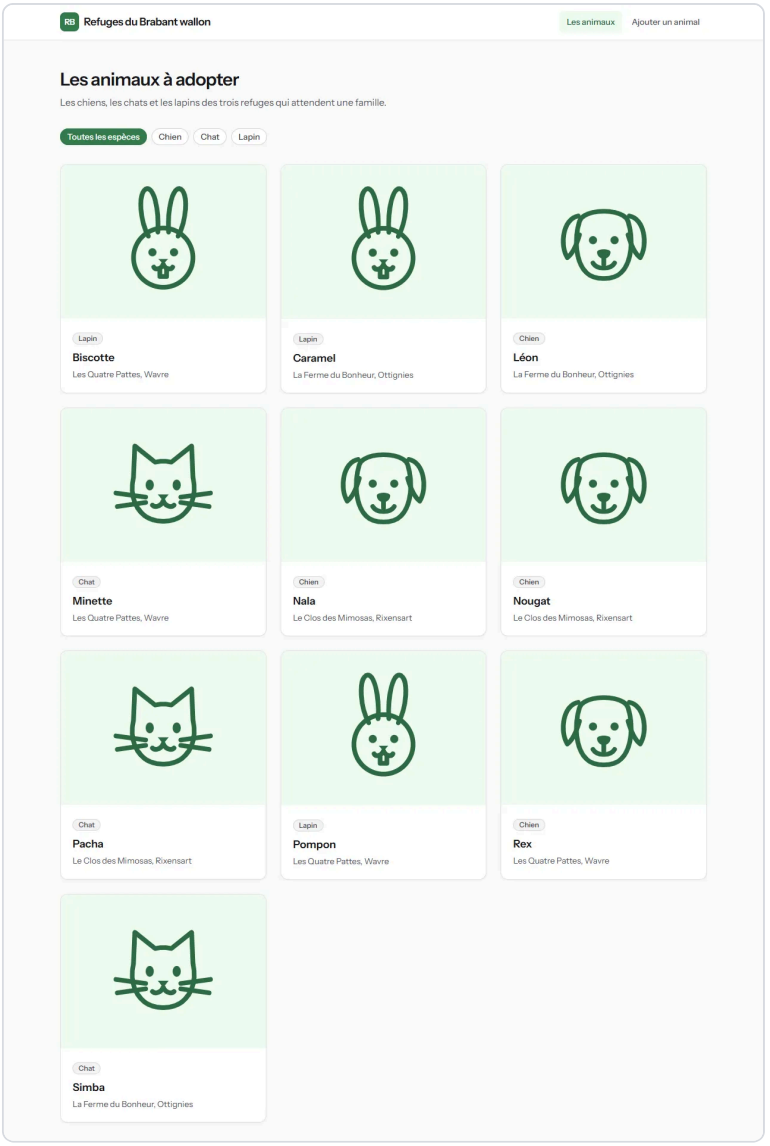
17.1 La liste et la fiche (20 minutes)

Créez `AnimalController` avec `index()` et `show()`, et les routes `animals.index` et `animals.show`. La liste ne montre que les animaux à adopter, triés par nom, dans une grille de cartes `<x-card>`. Chaque carte montre le pictogramme de l'espèce dans le slot `image`, avec `asset($animal->species->image())`, comme la couverture des jeux à l'exercice 14.3. Viennent ensuite le nom, en lien vers la fiche, puis le refuge. La fiche affiche le nom, l'espèce, le refuge, l'âge, les traits et la description. `$animal->age()` renvoie `null` quand la date de naissance est inconnue. Relisez `ProductController` de la friterie, et le chapitre 15, section 6.

Écrivez le nom du refuge sans lien pour l'instant. `route('shelters.show', …)` ferait planter la page tant que la route du bonus 17.8 n'existe pas, comme le menu écrit en dur. Les captures viennent de la solution, qui contient ce bonus.

```bash
php artisan make:controller AnimalController
php artisan test --group=liste
```

Test	Ce que vous écrivez
la route animals.index répond à GET /animaux	`Route::get('/animaux', …)->name('animals.index')`
la liste affiche les animaux à adopter, triés par nom	`orderBy('name')` dans `index()`
la liste ne montre pas les animaux adoptés	`whereNull('adopted_at')`, qui ne garde que les lignes où la colonne est vide
une liste vide affiche « Aucun animal »	`@forelse` avec `@empty`, ou `@if ($animals->isEmpty())`
la fiche affiche le nom, l'espèce, le refuge et les traits de caractère	`$animal->species->label()`, `$animal->shelter->name` et un `@foreach` sur `$animal->tags`
la fiche d'un animal adopté affiche « Adopté »	`@if ($animal->isAdopted())` autour de `<x-badge variant="adopted">Adopté</x-badge>`
la fiche d'un animal inconnu répond 404	`findOrFail()`, ou la liaison de modèle



La rangée de filtres, au-dessus de la grille, vient du bonus 17.8.

Critères

- [ ] La liste affiche une carte par ligne à 375 pixels, deux à partir de 640, et trois à partir de 1 024.
- [ ] Dans une même ligne, les cartes ont la même hauteur, comme au catalogue du chapitre 14.
- [ ] Sur la fiche d'Oscar, `/animaux/5`, absent de la liste parce qu'il est adopté, « Adopté » est un `<x-badge variant="adopted">` vert, à côté du badge de l'espèce.
- [ ] Les traits de la fiche sont des badges `variant="tag"`, qui passent à la ligne quand la place manque.

17.2 Ajouter un animal (25 minutes)

Ajoutez `create()` et `store()`, et les routes `animals.create` et `animals.store`. La vue `animals/create.blade.php` envoie en `POST` cinq champs : `name`, `species`, `shelter_id`, `birth_date` et `description`. Les traits arrivent au kata 17.5. Écrivez les champs avec les composants du dépôt, comme

le formulaire de la friterie au chapitre 15, section 2. Pour l'instant, `store()` ne valide rien : il crée l'animal avec `$request->only([...])`, puis redirige vers sa fiche avec un message flash. Affichez ce message dans le layout, comme au chapitre 15, section 5, à l'endroit marqué par un commentaire.

```bash
php artisan test --group=ajouter
```

Test	Ce que vous écrivez
GET /animaux/nouveau affiche un formulaire qui envoie en POST vers /animaux	`<form method="POST" action="{{ route('animals.store') }}">`, `@csrf`, les cinq champs, et les refuges passés par `create()`
POST /animaux enregistre l'animal et redirige vers sa fiche avec un message	`Animal::create(...)`, puis `redirect()->route('animals.show', $animal->id)->with('status', …)`
le message de confirmation s'affiche sur la fiche après l'ajout	`@session('status')` dans le layout

Le `<select>` des espèces fait une boucle sur `App\Enums\Species::cases()`, avec `value="{{ $species->value }}"` et le libellé `{{ $species->label() }}`.

Critères

- [ ] Le formulaire est centré dans le cadre d'une carte, comme celui de l'exercice 14.7, et chaque libellé est au-dessus de son champ.
- [ ] Après l'ajout, le message s'affiche au-dessus de la fiche, dans un encadré vert pâle au texte vert foncé.
- [ ] Dans l'onglet Éléments, le texte du message est dans une balise qui porte `role="status"`.
- [ ] Rechargée avec F5, la fiche n'affiche plus le message.

Le message s'affiche dans le layout, et non dans `animals/show.blade.php` : après une suppression, au kata 17.7, Laravel redirige vers la liste. À l'endroit marqué dans `<main>` :

```xml
@session('status')
    <p role="status">{{ $value }}</p>
@endsession
```

`role="status"` signale aux lecteurs d'écran un message d'état.

17.3 Valider le formulaire (25 minutes, installation comprise)

Installez les traductions françaises, comme au chapitre 15, section 4 :

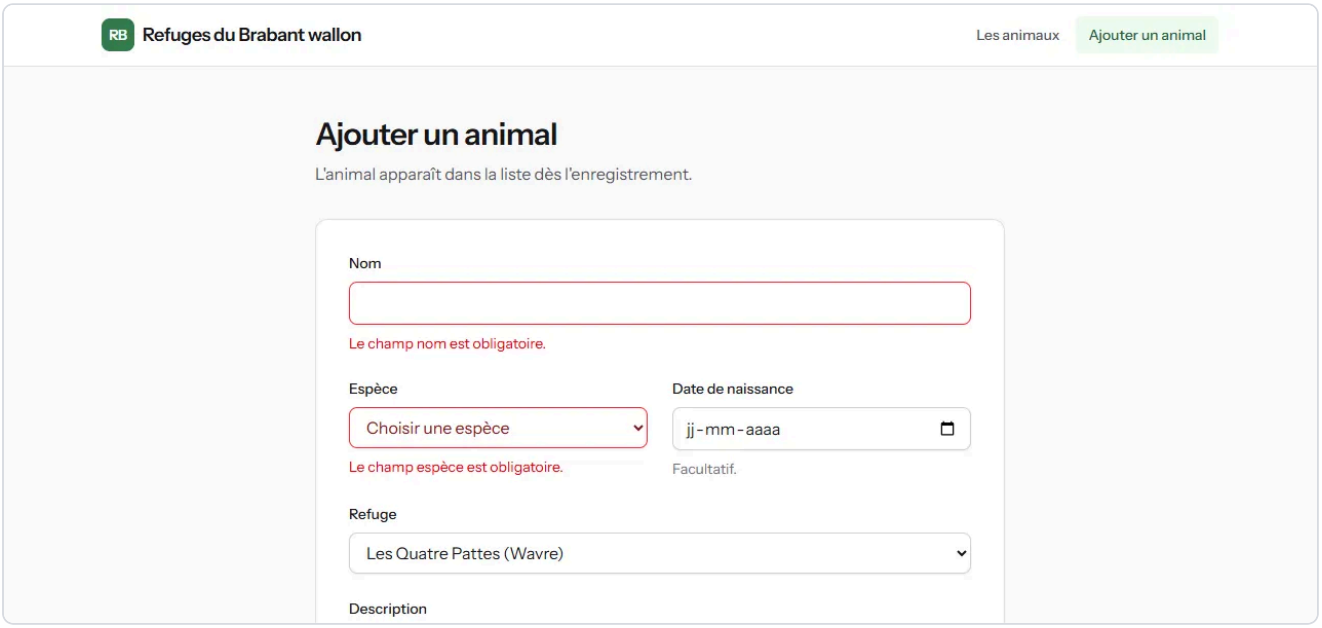
```bash
composer require laravel-lang/common --dev
php artisan lang:add fr
```

`APP_LOCALE=fr` est déjà dans `.env.example` et dans `phpunit.xml`. Enregistrez le dossier `lang/` dans un commit : la CI ne lance pas `lang:add`.

Dans `store()`, remplacez `$request->only()` par `$request->validate()`, avec les règles de la section 2 sauf celles de `tags`, comme au chapitre 15, section 3. `Rule::enum(Species::class)` refuse toute valeur qui n'est pas celle d'un cas de l'enum. Traduisez les noms des champs dans le tableau `attributes` de `lang/fr/validation.php`. Sous chaque champ, affichez le message avec `@error`, et rendez la valeur tapée avec `old()`.

```bash
php artisan test --group=valider
```

Test	Ce que vous écrivez
les traductions françaises sont installées	`composer require`, `lang:add fr`, et le dossier `lang/` enregistré dans un commit
un formulaire vide est refusé : erreurs sur name, species et shelter_id, rien en base	`required` sur les trois champs, et `Animal::create($validated)`
un nom de plus de 50 caractères est refusé	`max:50`
une espèce inconnue est refusée	`Rule::enum(Species::class)`, avec `use Illuminate\Validation\Rule;` et `use App\Enums\Species;`
un refuge qui n'existe pas est refusé	`exists:shelters,id`
une date de naissance dans le futur est refusée	`before_or_equal:today`
la date de naissance et la description sont facultatives	`nullable` sur les deux champs
le formulaire réaffiché garde les valeurs tapées et montre le message en français	`:value="old('name')"` dans le champ et `@error('name')` dessous, pour chaque champ



Critères

- ☐ Un champ refusé a une bordure rouge, et son message rouge s'affiche juste en dessous.
- ☐ Chaque champ refusé porte `aria-invalid="true"`, et son `aria-describedby` désigne son propre message : `aria-describedby="species-error"` sur l'espèce, et `id="species-error"` sur le message de l'espèce.
- ☐ Chaque message nomme le champ en français, comme « Le champ espèce est obligatoire. »
- ☐ Après un refus, les champs bien remplis gardent leur valeur, le refuge choisi compris.
- ☐ Aucun champ ne porte l'attribut `required` : envoyé vide, le formulaire part au serveur, et trois messages s'affichent.

Le tableau `attributes` de Laravel-Lang traduit déjà `name` en « nom ». `species` et `shelter_id` n'y figurent pas, et Laravel écrit alors le nom du champ, avec une espace à la place du trait de soulignement :

```
Le champ species est obligatoire.
Le champ shelter id est obligatoire.
```

Ajoutez `species`, `shelter_id` et `birth_date` au tableau. `tags` y est déjà, traduit par « mots clés » : vous le changerez au kata 17.5. Le message d'une date future finit par « au today » : le paquet ne traduit pas le mot qui suit les deux-points de `before_or_equal:today`, et aucun test ne lit ce message. Au kata 17.4, la Form Request le remplacera.

4. Modifier, adopter, supprimer

Les katas 17.4 à 17.7 modifient un animal, cochent ses traits, l'adoptent et le suppriment. Ils reprennent le chapitre 16. Le bonus 17.8 se fait à la maison, après la séance.

Kata	Groupe	À relire
17.4 Modifier un animal (25 minutes)	`modifier`	chapitre 16, sections 2 et 3, et section 5 pour `messages()`
17.5 Les traits de caractère (25 minutes)	`caracteres`	chapitre 16, section 5
17.6 Adopter (15 minutes)	`adopter`	chapitre 16, section 6
17.7 Supprimer (15 minutes)	`supprimer`	chapitre 16, section 7
17.8 Bonus : le refuge et le filtre (30 minutes, à la maison)	`bonus`	le composant du chapitre 16, section 5, `$request->query()` du chapitre 8, section 4, `tryFrom()` du chapitre 7 de la POO, section 5

17.4 Modifier un animal (25 minutes)

Ajoutez `edit()` et `update()`, et les routes `animals.edit` et `animals.update`. La vue `animals/edit.blade.php` reprend le formulaire d'ajout, envoyé vers `route('animals.update', $animal->id)` avec `@method('PUT')`, comme au chapitre 16, section 2. Chaque champ est pré-rempli avec `old()` à deux arguments. Pour les deux listes, `@selected()` compare chaque option à `old('species', $animal->species->value)`, puis à `old('shelter_id', $animal->shelter_id)`. `update()` valide avec les règles de 17.3, puis redirige vers la fiche avec « L'animal a été modifié. ». Ajoutez enfin un bouton « Modifier » sur la fiche, avec la prop `href` de `<x-button>`.

Rangez de préférence les règles dans une Form Request `AnimalRequest`, que `store()` et `update()` partagent. Faites renvoyer `true` à `authorize()`, comme au chapitre 16, section 3 : sinon, chaque envoi répond 403. Les tests acceptent aussi deux `validate()`, mais le message « au today » reste alors en place.

La Form Request corrige ce message du kata 17.3. Sa méthode facultative `messages()`, citée au chapitre 16, section 5, remplace le message d'une règle pour un champ, sous la clé `champ.règle` :

```php
/**
 * Les messages qui remplacent ceux de lang/fr/validation.php.
 *
 * @return array<string, string>
 */
public function messages(): array
{
    return [
        'birth_date.before_or_equal' => 'La date de naissance ne peut pas être dans le futur.',
    ];
}
```

Aucun test ne lit ce message : vérifiez-le dans le navigateur avec une date de l'an prochain. Les autres messages viennent toujours de `lang/fr/validation.php`, avec les noms que vous avez ajoutés à son tableau `attributes` au kata 17.3.

```bash
php artisan make:request AnimalRequest
php artisan test --group=modifier
```

Test	Ce que vous écrivez
GET /animaux/{id}/modifier affiche le formulaire pré-rempli, qui envoie en PUT	`@method('PUT')`, et `old('name', $animal->name)` dans chaque champ
envoyer le formulaire de modification met l'animal à jour et redirige vers sa fiche	$animal->update(\texttt{validated})$, puis la redirection avec le message
la modification est validée comme l'ajout	`AnimalRequest`, ou les règles de 17.3 dans `update()`
modifier un animal inconnu répond 404	`findOrFail()`, ou la liaison de modèle, dans `edit()` et dans `update()`

Critères

- [ ] Sur la fiche, « Modifier » est un lien en forme de bouton secondaire, blanc bordé de gris.
- [ ] Le formulaire de modification a la mise en forme du formulaire d'ajout, et son titre nomme l'animal : « Modifier Rex ».
- [ ] La date de naissance de Rex s'affiche dans son champ.
- [ ] Avec une date de naissance de l'an prochain, le formulaire affiche « La date de naissance ne peut pas être dans le futur. ».

Un `<input type="date">` n'accepte qu'une valeur de la forme `2021-03-14`. Affichée telle quelle, `birth_date` s'écrit `2021-03-14 00:00:00`, et le navigateur laisse le champ vide. Écrivez `$animal->birth_date?->format('Y-m-d')` : l'opérateur `?->` du chapitre 5 de la POO renvoie `null` quand la date est inconnue.

17.5 Les traits de caractère (25 minutes)

Dans les deux formulaires, ajoutez une case par trait de caractère, nommée `tags[]`. Pour ne pas écrire les cases deux fois, rangez tous les champs dans un composant `<x-animal-fields :animal="`$animal$`" :`$shelters$`="shelters" :tags="$tags" />`, comme au chapitre 16, section 5. `create()` lui passe alors `new Animal`, dont l'espèce vaut `null`. `$animal->species->value` s'arrête sur `Attempt to read property "value" on null` : écrivez `$animal->species?->value`. L'opérateur `?->` du chapitre 5 de la POO renvoie `null` au lieu de lire `value` quand l'espèce est absente.

Au formulaire de modification, les traits de l'animal sont pré-cochés avec `@checked`, comme les allergènes au chapitre 16, section 5. Ajoutez les règles `tags` et `tags.*` du tableau de la section 2, et enregistrez les traits cochés avec `sync()` dans `store()` et `update()`. Dans le tableau `attributes` de `lang/fr/validation.php`, remplacez « mots clés » par « traits de caractère », et ajoutez `'tags.*' => 'trait de caractère'`.

```bash
php artisan make:component animal-fields --view
php artisan test --group=caracteres
```

Test	Ce que vous écrivez
le formulaire d'ajout propose une case par trait de caractère	les traits passés par `create()` et `edit()`, une case `name="tags[]" value="{{ $tag->id }}"` par trait
à l'ajout, les traits cochés sont enregistrés dans animal_tag	`$animal->tags()->sync(...)` dans `store()`
le formulaire de modification pré-coche les traits de l'animal	`@checked(in_array($tag->id, $checkedTags))` sur chaque case
à la modification, la liste des traits est remplacée	`sync()` dans `update()`
décocher toutes les cases retire tous les traits	`sync($validated['tags'] ?? [])`
un trait qui n'existe pas est refusé	`'tags.*' => ['integer', 'exists:tags,id']`

Critères

- [ ] Les huit cases s'affichent sur deux colonnes à 375 pixels, et sur quatre à partir de 640.
- [ ] Les cases sont regroupées dans un `<fieldset>`, dont la `<legend>` dit « Traits de caractère ».
- [ ] Un clic sur le nom d'un trait coche ou décoche sa case.
- [ ] Après un refus sur le nom, les cases gardent l'état choisi avant l'envoi, toutes décochées comprises.

`@checked(…)` écrit l'attribut `checked` quand sa condition est vraie :

```XML
<input type="checkbox" name="tags[]" value="{{ tag->id }}" @checked(in_array(tag->id,
$checkedTags))>
```

En tête du composant, la liste des cases cochées vient de la session après un refus, et de la base sinon :

```XML
@php
    checkedTags=session()->hasOldInput()?(array)old('tags',[]):checkedTags = session()
->hasOldInput() ? (array) old('tags', []) : checkedTags=session()->hasOldInput()?
(array)old('tags',[]):animal->tags->pluck('id')->all();
@endphp
```

`hasOldInput()` dit si un envoi précédent a laissé des valeurs dans la session. Après un refus, une liste absente veut dire qu'aucune case n'était cochée. À l'ajout, `new Animal` n'a aucun trait. `(array)` protège d'une valeur seule envoyée hors du navigateur, comme au chapitre 16, section 5. Un navigateur n'envoie pas une case décochée : sans case cochée, `tags` est absent de la requête, et `$validated['tags'] ?? []` donne un tableau vide à `sync()`.

17.6 Adopter (15 minutes)

Ajoutez la route `animals.adopt` et la méthode `adopt()`. Sur la fiche d'un animal à adopter, un formulaire d'un bouton « Adopter » envoie la requête avec `@method('PATCH')`, comme le bouton « Marquer en rupture » de la friterie au chapitre 16, section 6. `adopt()` pose `adopted_at` à `today()` si l'animal n'est pas encore adopté, puis redirige vers la fiche avec « Rex a trouvé une famille. », écrit avec le nom de l'animal. `today()` renvoie la date du jour à minuit, comme `now()` au chapitre 16 mais sans l'heure.

```BASH
php artisan test --group=adopter
```

Test	Ce que vous écrivez
PATCH /animaux/{id}/adoption enregistre la date d'adoption et redirige vers la fiche	`$animal->update(['adopted_at' => today()])`, puis la redirection avec `"{$animal->name} a trouvé une famille."`
la fiche d'un animal à adopter contient le formulaire d'adoption, celle d'un animal adopté ne le contient plus	`@if (! $animal->isAdopted())` autour du formulaire
un animal adopté disparaît de la liste	rien de plus : le `whereNull('adopted_at')` du kata 17.1
adopter un animal déjà adopté ne change pas sa date d'adoption	un `if (! $animal->isAdopted())` dans `adopt()`

Test	Ce que vous écrivez
adopter un animal inconnu répond 404	`findOrFail()`, ou la liaison de modèle

Critères

- [ ] « Adopter » est un bouton principal, vert, de type `submit`.
- [ ] Après l'adoption, le message s'affiche dans l'encadré vert du kata 17.2, et le badge remplace le bouton.
- [ ] L'animal adopté n'apparaît plus sur `/animaux`.

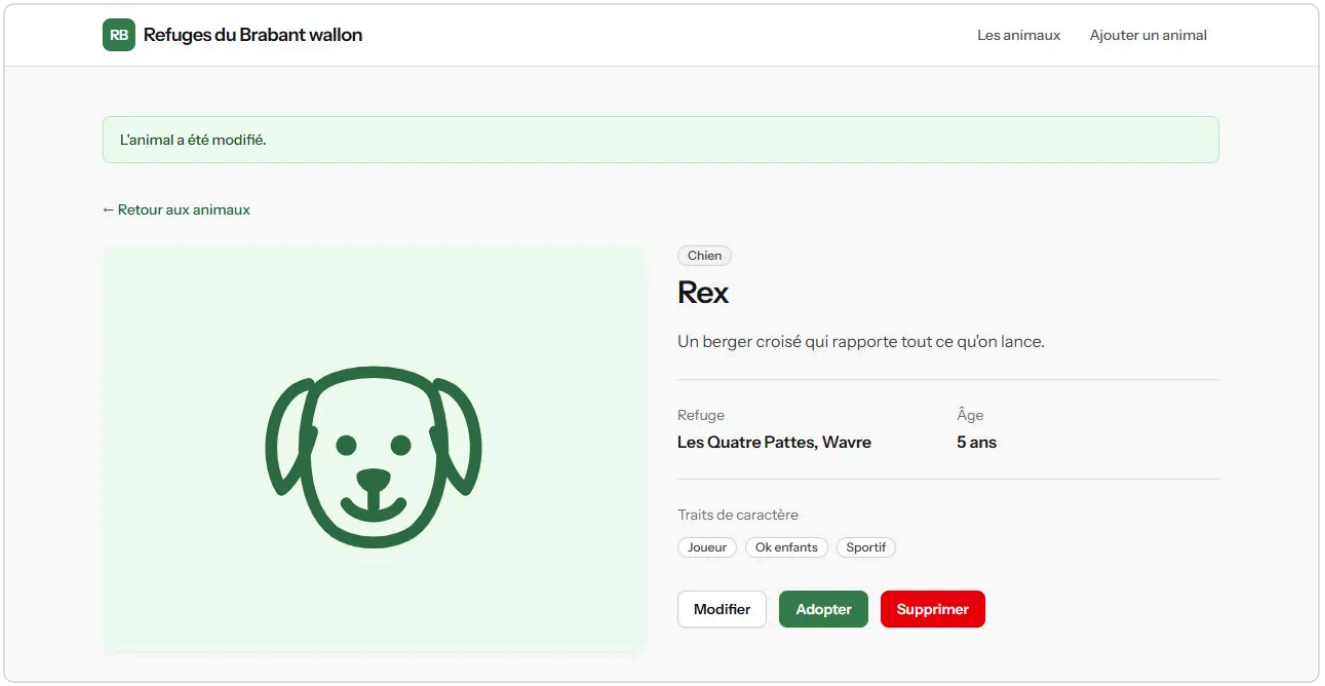
> Le test appelle `$animal->fresh()`, qui relit la ligne dans la base. Un `$animal->adopted_at = today()` sans `save()` ne change que l'objet en mémoire, et la base garde `null`. `update()` change la propriété et enregistre la ligne en une fois. Sans le `if`, le test « adopter un animal déjà adopté… », vert au départ, devient rouge.

17.7 Supprimer (15 minutes)

Ajoutez la route `animals.destroy` et la méthode `destroy()`. Sur la fiche, ajoutez un formulaire `POST` avec `@csrf`, `@method('DELETE')` et un bouton `<x-button type="submit" variant="danger">`. Il demande une confirmation avec `onsubmit="return confirm('Supprimer cet animal ?')"`, comme au chapitre 16, section 7. `<x-button>` écrit `type="button"` par défaut : sans `type="submit"`, le clic n'envoie rien. `destroy()` supprime l'animal, puis redirige vers la liste avec « L'animal a été supprimé. ».

```bash
php artisan test --group=supprimer
```

Test	Ce que vous écrivez
la fiche contient un formulaire de suppression qui envoie en DELETE	`<form method="POST" action="{{ route('animals.destroy', $animal->id) }}">` avec `@method('DELETE')`
DELETE /animaux/{id} supprime l'animal et redirige vers la liste	`$animal->delete()`, puis `redirect()->route('animals.index')->with('status', …)`
supprimer un animal supprime aussi ses lignes dans animal_tag	rien de plus : `cascadeOnDelete()` est dans la migration de `animal_tag`
supprimer un animal inconnu répond 404	`findOrFail()`, ou la liaison de modèle



Critères

- [ ] Les trois boutons de la fiche ont trois variantes : « Modifier » secondaire, « Adopter » principal, « Supprimer » rouge.
- [ ] Les trois boutons sont sur une ligne, et passent à la ligne quand la place manque.
- [ ] Un clic sur « Supprimer » ouvre une boîte de confirmation, et « Annuler » laisse l'animal en place.
- [ ] Un clic sur « Supprimer », puis sur OK, ramène à la liste sous « L'animal a été supprimé. ».
- [ ] Au clavier, chaque bouton reçoit un contour de sa couleur.

5. Bonus

17.8 Bonus : le refuge et le filtre (30 minutes, à la maison)

Créez `ShelterController`, la route `shelters.show` et la vue `shelters/show.blade.php` : le nom et la ville du refuge, puis ses animaux à adopter, dans la grille de la liste. Sur la fiche et sur les cartes, le nom du refuge devient un lien vers sa page.

Sur `/animaux`, ajoutez un filtre par espèce : trois liens vers `route('animals.index', ['espece' ⇒ 'chat'])` et ses voisines, qui ajoutent `?espece=chat` à l'adresse. Dans `index()`, `$request ->query('espece', '')` lit la valeur, ou une chaîne vide. `Species::tryFrom()`, vue au chapitre 7 de la POO, la change en cas de l'enum, ou renvoie `null` si elle est inconnue : ne filtrez que dans le premier cas. Pour aller plus loin, remplacez `confirm()` par une boîte `<dialog>`, ouverte par sa méthode `showModal()` et fermée par `close()`. Son script va dans `resources/js/app.js`, comme celui du menu à l'exercice 14.5.

```bash
php artisan make:controller ShelterController
php artisan test --group=bonus
```

Test	Ce que vous écrivez
GET /refuges/{id} affiche le refuge, sa ville et ses animaux à adopter	`$shelter->animals()->whereNull('adopted_at')->orderBy('name')->get()`
GET /refuges/999 répond 404	`findOrFail()`, ou la liaison de modèle
/animaux?espece=chat ne garde que les chats	`where('species', $species)` quand `tryFrom()` a trouvé un cas
une espèce inconnue dans le filtre est ignorée	aucun filtre quand `tryFrom()` renvoie `null`

Critères

- [ ] Le lien de l'espèce choisie se distingue des autres, et porte `aria-current="page"`.
- [ ] La page d'un refuge reprend les cartes de la liste sans les recopier, avec un composant `<x-animal-card :animal="$animal" />`, comme `<x-animal-fields>` au kata 17.5.
- [ ] La boîte `<dialog>` a les couleurs du site, et se ferme avec Échap.

À retenir

- Les tests d'un kata sont son énoncé, et le fichier du test montre comment chaque phrase est vérifiée.
- Une route imposée fixe le verbe, l'adresse et le nom, et laisse libres le contrôleur et la vue.
- Le message flash s'affiche dans le layout, sur la page où mène la redirection.
- Un navigateur n'envoie pas une case décochée, et `sync()` reçoit alors un tableau vide.
- Tout ce qui modifie la base passe par un formulaire `POST`, avec `@method()` pour `PUT`, `PATCH` ou `DELETE`.

Six cartes

Pourquoi le menu du layout est-il écrit en dur ?

Tant que la route `animals.index` n'existe pas, `route('animals.index')` lève une exception, et toutes les pages planteraient.

Pourquoi afficher le message flash dans le layout ?

Après une suppression, la redirection mène à la liste.

Pourquoi enregistrer le dossier `lang/` dans un commit ?

La CI ne lance pas `lang:add fr`.

Après un refus, pourquoi `session()->hasOldInput()` ?

Une case décochée n'est pas envoyée. Sans cette condition, une liste vide après un refus ferait revenir les traits enregistrés.

Une case décochée, que reçoit le serveur ?

Rien : le champ `tags` est absent, et `sync($validated['tags'] ?? [])` retire tous les traits.

Pourquoi le test d'adoption appelle-t-il `fresh()` ?

Il relit l'animal dans la base. Une propriété changée sans `save()` ne modifie que l'objet en mémoire.

Widget interactif — à voir en ligne : https://cours.tsix.be/cours/framework-cote-serveur/framework/le-refuge-exercices

3.18 Téléverser une image

Bloc Laravel · en autonomie · ~90 min

Sur le site du refuge, la fiche de Rex montre le même pictogramme de chien que celles de Nala, de Léon et de Nougat. Les bénévoles voudraient montrer une vraie photo de chaque animal. Dans ce chapitre, vous ajoutez un champ photo au formulaire d'un animal, vous rangez le fichier reçu sur le serveur, puis vous l'affichez sur la fiche et dans la liste. Envoyer un fichier de son ordinateur vers un serveur s'appelle **téléverser**. Vous faites ce chapitre seul, après le chapitre 17.

À la fin de ce chapitre, vous serez capables de :

1. **envoyer** un fichier depuis un formulaire, et **expliquer** le rôle de `enctype="multipart/form-data"`,
2. **valider** une image et sa taille, avec des messages en français,
3. **enregistrer** le fichier sur le disque `public`, et **l'afficher** grâce au lien `public/storage`,

4. **supprimer** l'ancien fichier quand la photo change ou que l'animal disparaît,
5. **vérifier** le tout avec des tests qui simulent le disque et le fichier.

1. Avant de commencer

Vous partez de votre fork de `refuge-26`, avec les huit katas du chapitre 17 faits. Dans son dossier, lancez les tests :

```bash
php artisan test
```

Les 41 tests doivent être verts. Les extraits de ce chapitre suivent le corrigé des katas : si votre code diffère, adaptez l'endroit où vous collez chaque morceau.

Les tests de ce chapitre sont donnés en entier à la section 8. Comme au chapitre 17, ils sont l'énoncé. Vous pouvez copier le fichier dès maintenant, et lancer `php artisan test --group=photo` après chaque section.

Préparez deux photos de moins de 2 Mo, en JPEG ou en PNG, nommées `rex.jpg` et `rex-ete.jpg`, et un fichier PDF quelconque.

2. Un fichier dans un formulaire

Les champs du dépôt passent par les composants `<x-input>`, `<x-select>` et `<x-textarea>`, qui lisent `$errors` eux-mêmes. Aucun ne gère un champ de fichier, et `<x-input type="file">` ne conviendrait pas : ses classes dessinent la bordure d'un champ texte. Créez un quatrième composant, `resources/views/components/file-input.blade.php`, sur le modèle d'`input.blade.php` :

```xml
{{-- Un champ de fichier. Comme <x-input>, il lit errors : en erreur, aria-invalid="true" et
aria-describedby="{name}-error". --}}
@props(['name'])

<input
    type="file"
    name="{{ name }}"
    @if (errors->has(errors→has(errors->has(name)) aria-invalid="true" aria-describedby="{{
name }}-error" @endif
    {{ attributes→merge(['id' ⇒ $name])→class('mt-2 block w-full text-sm text-zinc-700
file:mr-4 file:rounded-lg file:border-0 file:bg-brand-50 file:px-4 file:py-2 file:font-
semibold file:text-brand-700 hover:file:bg-brand-100 dark:text-zinc-300') }}
>
```

Un champ `type="file"` affiche un bouton « Choisir un fichier », qui ouvre la fenêtre de choix de votre système. Le modificateur Tailwind `file:` met en forme ce bouton. Dans `resources/views/components/animal-fields.blade.php`, ajoutez ce bloc entre la description et les traits de caractère :

```xml
<div>
    <x-label for="photo">Photo</x-label>
    <x-file-input name="photo" accept="image/*" />
</div>
```

`accept="image/*"` demande à la fenêtre de choix de ne proposer que des images. C'est un confort, pas un contrôle : la fenêtre permet de passer à « Tous les fichiers ».

Pour voir ce que le serveur reçoit, ajoutez cette ligne en tête de `store()`, dans `AnimalController` :

```php
dd(request->file('photo'),request→file('photo'), request->file('photo'),request
→input('photo'));
```

`$request→file('photo')` renvoie le fichier reçu dans le champ `photo`, ou `null`. Ouvrez `http://refuge-26.test/animaux/nouveau`, tapez `Rex`, choisissez Chien et un refuge, puis choisissez `rex.jpg`. Envoyez :

```
null // app\Http\Controllers\AnimalController.php:47

"rex.jpg" // app\Http\Controllers\AnimalController.php:47
```

Le serveur n'a reçu que le nom du fichier, comme un champ texte. Par défaut, un formulaire envoie ses champs les uns derrière les autres, comme dans une adresse : `name=Rex&species=chien&…&photo=rex.jpg`. Cette écriture ne transporte que du texte, et le navigateur y place le nom du fichier sans son contenu.

`enctype`

Pour envoyer le contenu, le formulaire doit choisir une autre écriture, `multipart/form-data`. Le corps de la requête est alors découpé en parties, une par champ. La partie du champ `photo` contient le nom du fichier, son type, puis tous ses octets. L'attribut **`enctype`** de la balise `<form>` choisit cette écriture. Ajoutez-le dans `animals/create.blade.php` :

```xml
<form method="POST" action="{{ route('animals.store') }}" enctype="multipart/form-data"
class="…">
```

Faites de même dans `animals/edit.blade.php`, sur le formulaire qui envoie vers `route('animals.update', $animal→id)`. Rechargez le formulaire d'ajout, remplissez-le de nouveau, choisissez `rex.jpg` et envoyez. Sortie raccourcie :

```
Illuminate\Http\UploadedFile {#508 ▼ // app\Http\Controllers\AnimalController.php:47
  -originalName: "rex.jpg"
  -mimeType: "image/jpeg"
  -error: 0
  …
  pathname: "C:\Users\admin\AppData\Local\Temp\php992B.tmp"
  …
  size: 7172
  …
}

null // app\Http\Controllers\AnimalController.php:47
```

Laravel range le fichier reçu dans un objet `UploadedFile`. `originalName` et `mimeType` viennent du navigateur. `pathname` montre où PHP a écrit le contenu : un fichier temporaire, `php992B.tmp`, que PHP efface à la fin de la requête. Tant que le code ne le copie pas ailleurs, la photo est perdue.

Retirez la ligne `dd()`. **Un formulaire qui envoie un fichier porte `enctype="multipart/form-data"`, sinon le serveur ne reçoit que le nom du fichier.**

3. La colonne `photo`

La base ne garde pas l'image elle-même. Elle garde le chemin du fichier, un texte comme `animals/khqE….jpg`. Il faut une colonne pour ce chemin :

```bash
php artisan make:migration add_photo_to_animals_table
```

```
   INFO  Migration [database/migrations/2026_10_10_144822_add_photo_to_animals_table.php]
created successfully.
```

Chez vous, Artisan affiche le chemin complet et la date du jour. Le nom finit par `to_animals_table` : Laravel en déduit que la migration modifie la table `animals`, et le fichier contient déjà `Schema::table('animals', …)`. Complétez `up()` et `down()` :

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::table('animals', function (Blueprint table) {
            $table->string('photo')->nullable();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::table('animals', function (Blueprint table) {
            $table->dropColumn('photo');
        });
    }
};
```

La colonne accepte `null` : les douze animaux du seeder n'ont pas de photo, et un bénévole peut ajouter un animal sans photo. `down()` retire la colonne si vous annulez la migration avec `php artisan migrate:rollback`.

```bash
php artisan migrate
```

```
   INFO  Running migrations.

  2026_10_10_144822_add_photo_to_animals_table ....................................
10.64ms DONE
```

Dans `app/Models/Animal.php`, ajoutez `photo` à la liste `#[Fillable]`, sinon `create()` et `update()` ignorent la colonne :

```php
#[Fillable(['shelter_id', 'name', 'species', 'birth_date', 'description', 'photo',
'adopted_at'])]
```

4. Valider le fichier

Dans `AnimalRequest`, ajoutez une règle pour `photo`, sous celles de `tags` :

```php
'photo' ⇒ ['nullable', 'image', 'max:2048'],
```

Règle	Ce qu'elle vérifie
`nullable`	le champ peut rester vide
`image`	le fichier est une image JPEG, PNG, GIF, BMP, WebP, AVIF ou HEIC
`max:2048`	pour un fichier, `max` compte en kilo-octets : 2 048 Ko, soit 2 Mo

Rien à ajouter pour le nom du champ : le tableau `attributes` de `lang/fr/validation.php`, venu de Laravel-Lang, contient déjà `'photo' ⇒ 'photo'`, et Laravel s'en sert pour les champs que la méthode `attributes()` d'`AnimalRequest` ne nomme pas.

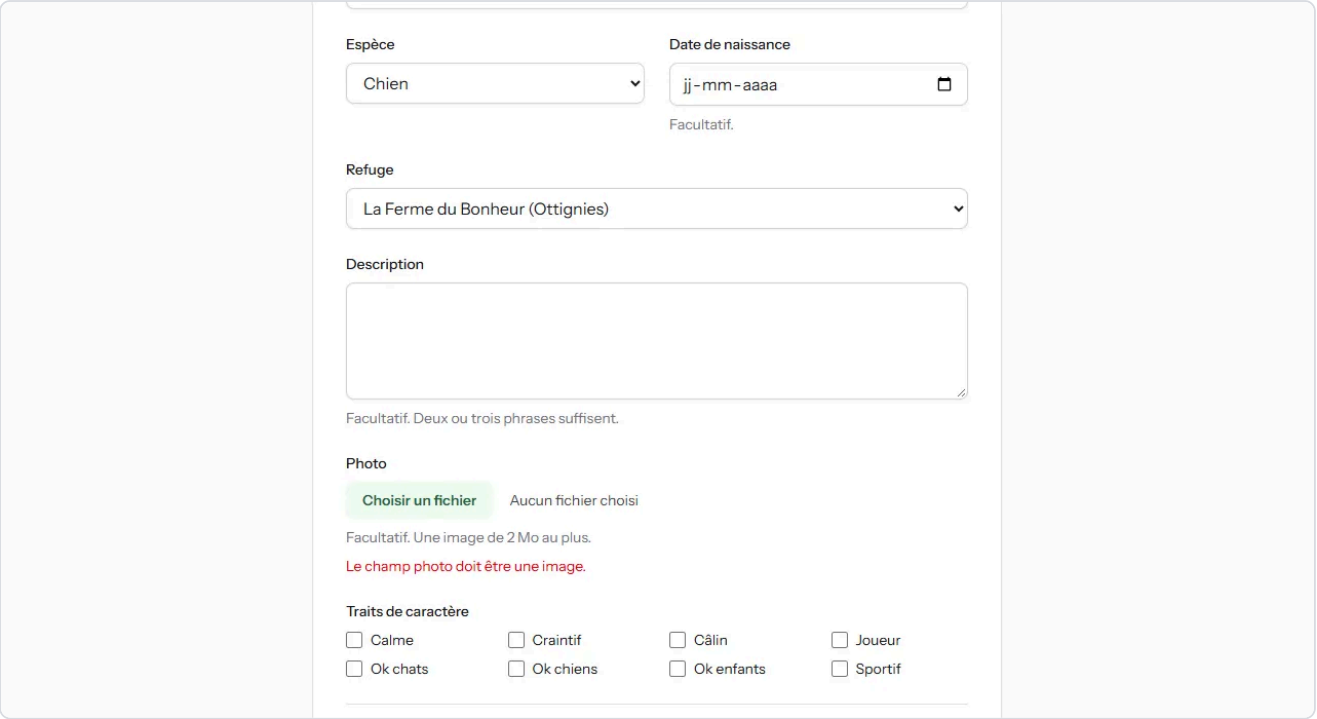
Le message d'erreur s'affiche sous le champ, comme pour les autres. Remplacez le bloc de la section 2 par sa version complète :

```xml
<div>
    <x-label for="photo">Photo</x-label>
    <x-file-input name="photo" accept="image/*" />
    <p class="mt-2 text-sm text-zinc-500 dark:text-zinc-400">Facultatif. Une image de 2 Mo
au plus.</p>
    @error('photo')
        <p id="photo-error" class="mt-2 text-sm text-red-600 dark:text-red-400">{{ $message
}}</p>
    @enderror
</div>
```

Le message garde l'identifiant `photo-error`, celui que le composant annonce dans `aria-describedby` quand le serveur a refusé le fichier, comme `<x-input>` pour les autres champs. Gardez `composer run dev` ouvert, ou lancez `npm run build`, pour que Vite produise les classes `file:`.

Sur le formulaire d'ajout, remplissez les champs, passez le filtre de la fenêtre sur « Tous les fichiers » et choisissez votre PDF. Envoyez :

```
Le champ photo doit être une image.
```



Le nom et l'espèce sont restés, grâce à `old()`. Le champ Photo, lui, est vide. **Un navigateur ne pré-remplit jamais un champ de fichier**, et `old()` ne peut rien y mettre : il faut choisir le fichier de nouveau.

Renommez une copie du PDF en `rex.jpg` et envoyez-la : le message est le même. La règle `image` ne se fie ni au nom ni au type annoncé par le navigateur. Laravel lit les premiers octets du fichier, y reconnaît un PDF, et refuse.

Avec une photo de 7 Mo, le message change :

```
La taille du fichier de photo ne peut pas dépasser 2048 kilo-octets.
```

ASTUCE

« Le fichier du champ photo n'a pu être téléversé. » veut dire que PHP a écarté le fichier avant Laravel, parce qu'il dépasse `upload_max_filesize`, la limite du fichier `php.ini`. Lisez-la avec `php -i | grep upload_max_filesize`, ou `php -i | Select-String upload_max_filesize` dans PowerShell. Si une photo de moins de 2 Mo reçoit ce message, augmentez la limite dans les réglages de PHP de Herd ou de Laragon, puis redémarrez.

5. Enregistrer le fichier

Dans `AnimalController`, `store()` range la photo avant de créer l'animal :

```php
public function store(AnimalRequest $request): RedirectResponse
{
    $validated = $request->validated();

    if ($request->hasFile('photo')) {
        $validated['photo'] = $request->file('photo')->store('animals', 'public');
    }

    $animal = Animal::create($validated);
    $animal->tags()->sync($validated['tags'] ?? []);

    return redirect()
        ->route('animals.show', $animal->id)
        ->with('status', 'L\'animal a été ajouté.');
}
```

`$request->hasFile('photo')` vaut `true` quand la requête contient un fichier reçu sans erreur. La méthode **`store()`** de l'objet `UploadedFile` copie le fichier temporaire dans le dossier `animals` du disque `public`, et renvoie son chemin. Ce chemin remplace l'objet `UploadedFile` dans `$validated`, et `create()` l'écrit dans la colonne `photo`. Ajoutez le même `if` dans `update()`, entre $validated = $request->validated();$ et $animal->update(validated);$.

Un **disque** est un emplacement nommé où Laravel range des fichiers. Les disques sont décrits dans `config/filesystems.php`. Les deux premiers s'appellent `local` et `public` :

```php
'local' => [
    'driver' => 'local',
    'root' => storage_path('app/private'),
    'serve' => true,
    'throw' => false,
    'report' => false,
],

'public' => [
    'driver' => 'local',
    'root' => storage_path('app/public'),
    'url' => rtrim(env('APP_URL', 'http://localhost'), '/').'/storage',
    'visibility' => 'public',
    'throw' => false,
    'report' => false,
],
```

Le mot `local` apparaît deux fois, pour deux choses différentes. Le **pilote**, `'driver' => 'local'`, dit comment les fichiers sont rangés : sur le disque dur du serveur, dans le dossier `root`. Les deux disques partagent ce pilote. Le troisième du fichier, `s3`, en utilise un autre, qui envoie les fichiers chez Amazon. Le **nom** du disque, `local` ou `public`, dit où et pour qui. Le disque `local` range dans `storage/app/private` ce que personne ne doit télécharger, comme une sauvegarde de la base. Le disque `public` range dans `storage/app/public` ce qui est fait pour être vu, et sa ligne `url` donne l'adresse sous laquelle le navigateur le demandera. Une photo d'animal est faite pour être vue : elle va sur le disque `public`.

Ouvrez `http://refuge-26.test/animaux/1/modifier`, le formulaire de Rex. Choisissez `rex.jpg` et enregistrez. Lisez la colonne :

```bash
php artisan tinker --execute="echo App\Models\Animal::find(1)->photo;"
```

```
animals/khqEGt4XYzzwclcPfTYNIR3V1rhPIloTsGT8mRcj.jpg
```

```bash
ls storage/app/public/animals
```

```
khqEGt4XYzzwclcPfTYNIR3V1rhPIloTsGT8mRcj.jpg
```

Chez vous, les quarante caractères sont différents, et PowerShell les affiche dans un tableau. `store()` tire ce nom au hasard, pour deux raisons. Deux bénévoles qui envoient chacun un `photo.jpg` n'écrasent pas le fichier l'un de l'autre. Et le nom choisi par l'expéditeur, qui peut contenir des espaces, des accents ou un `../`, n'arrive jamais sur le disque. L'extension `.jpg` vient du contenu, comme pour la règle `image`.

La fiche de Rex affiche toujours le pictogramme : rien ne lit encore la colonne.

6. Afficher la photo

Dans `animals/show.blade.php`, remplacez la balise `<img>` du pictogramme par ce bloc :

```xml
@if ($animal->photo !== null)
    <img src="{{ asset('storage/'.$animal->photo) }}" alt="Photo de {{ $animal->name }}"
class="aspect-4/3 w-full rounded-xl object-cover shadow-xs">
@else
    <img src="{{ asset($animal->species->image()) }}" alt="" width="200" height="200"
class="aspect-4/3 w-full rounded-xl bg-brand-50 object-contain p-8 shadow-xs dark:bg-brand-
100">
@endif
```

`asset('storage/'.$animal->photo)` donne l'adresse `http://refuge-26.test/storage/animals/khqE….jpg`. `object-cover` remplit le cadre sans déformer la photo, en coupant ce qui dépasse. Sans photo, la fiche garde le pictogramme. Son `alt` reste vide, car le pictogramme décore et le badge écrit déjà « Chien ». La photo, elle, montre un animal précis, et `alt="Photo de Rex"` le dit aux lecteurs d'écran.

Faites le même changement dans la carte de la liste, `animals/index.blade.php` ou `<x-animal-card>` selon votre code, avec `object-cover` pour la photo.

Rechargez la fiche de Rex. À la place de la photo, le navigateur montre une image cassée. Ouvrez F12, onglet Réseau, et rechargez : la requête vers `storage/animals/khqE….jpg` répond 403.

Au chapitre 6, vous avez vu que la racine web est `public/`, et le serveur n'y trouve aucun dossier `storage`. La requête arrive donc à `public/index.php`, et Laravel la confie à la route `storage/{path}`, que `route:list` vous a montrée au même chapitre. Cette route sert les fichiers du disque `local`, et seulement aux adresses signées que Laravel fabrique lui-même. L'adresse de la photo n'a pas de signature, et la route répond 403.

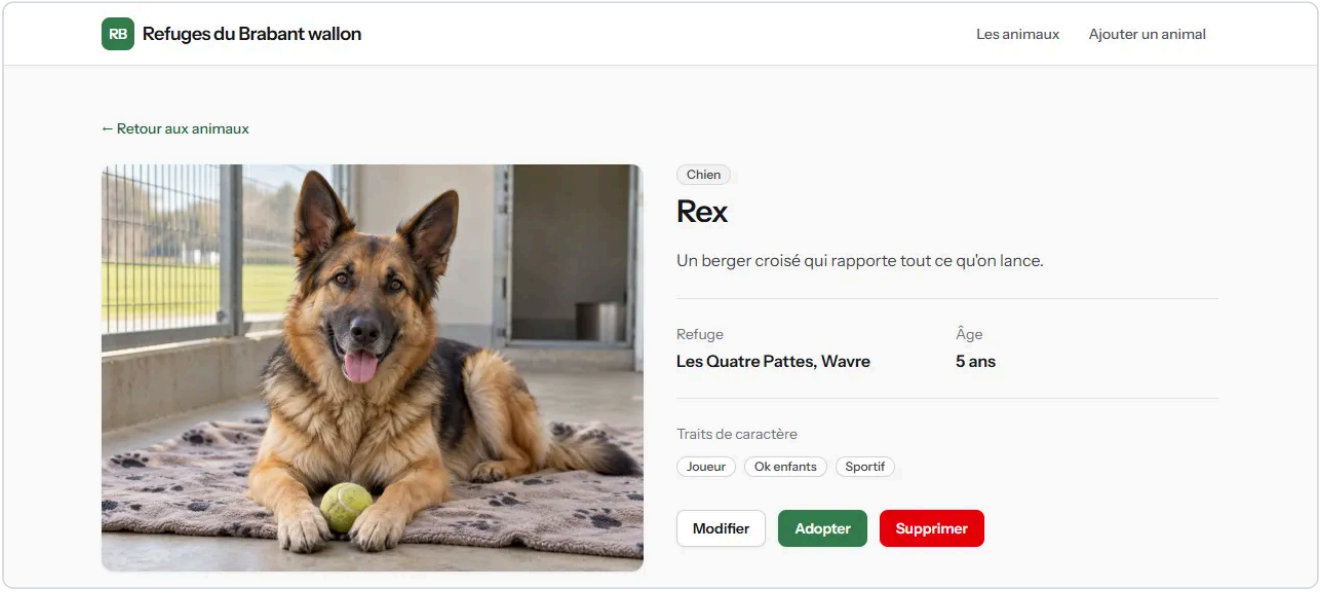
Le lien `public/storage`

Laravel prévoit une commande qui rend le dossier `storage/app/public` visible depuis `public/` :

```bash
php artisan storage:link
```

```
   INFO  The [C:\…\refuge-26\public\storage] link has been connected to [C:\…\refuge-26\storage\app/public].
```

Chez vous, Artisan affiche les chemins complets. La commande crée `public/storage`, qui n'est pas un vrai dossier. C'est un **lien symbolique** : une entrée de dossier qui renvoie vers un autre dossier. Ouvrir `public/storage/animals` revient à ouvrir `storage/app/public/animals`, et le serveur web trouve enfin le fichier. Sous Windows, Laravel crée une variante appelée jonction, qui ne demande pas de droits d'administrateur. Rechargez la fiche : la photo de Rex s'affiche.



La liste `/animaux` montre la photo dans la carte de Rex, et le pictogramme dans les autres.

> **ASTUCE**
>
> « The [... public\storage] link already exists. » veut dire que le lien est déjà là : il n'y a rien à refaire.

Ce que Git ne voit pas

Ouvrez `.gitignore`, à la racine du projet. Il contient cette ligne :

```
/public/storage
```

Le lien n'est pas versionné : il pointe vers un chemin propre à votre ordinateur. Les photos non plus, car `storage/app/public/.gitignore` contient `*`, qui exclut tout, puis `!.gitignore`, qui garde ce seul fichier. Quiconque clone le dépôt lance donc `php artisan storage:link` une fois, comme `php artisan key:generate`. Ajoutez la commande aux étapes d'installation de votre `README.md`.

Pourquoi `asset()` plutôt que `Storage::disk('public')->url($animal->photo)` ? Relisez la ligne `url` du disque `public`, à la section 5. Elle construit l'adresse avec `APP_URL`, qui vaut `http://localhost:8000` dans le `.env` du dépôt. `asset()`, lui, part de l'adresse de la page ouverte, `http://refuge-26.test`.

7. Remplacer et supprimer

Modifiez encore Rex, choisissez `rex-ete.jpg` et enregistrez. La fiche montre la nouvelle photo. Regardez le dossier :

```bash
ls storage/app/public/animals
```

```
F2GAvUCbwejurmSCCUeysaorEVI8eB9MBjNTz8jW.jpg
khqEGt4XYzzwclcPfTYNIR3V1rhPIloTsGT8mRcj.jpg
```

La colonne `photo` désigne le nouveau fichier, et l'ancien reste sur le disque sans qu'aucune ligne le cite. Ajoutez ensuite un animal, Bruno, avec une photo, puis supprimez-le depuis sa fiche. Sa ligne disparaît, mais son fichier reste dans `storage/app/public/animals`. Après un an d'adoptions, le dossier serait plein de photos que plus personne ne peut afficher.

La base ne connaît que le chemin : le fichier se supprime à part. `Storage::disk('public')->delete($path)` efface le fichier `$path` du disque `public`. Importez la façade en haut du contrôleur :

```php
use Illuminate\Support\Facades\Storage;
```

Dans `update()`, effacez l'ancienne photo quand une nouvelle arrive :

```php
public function update(AnimalRequest request,Animalrequest, Animal request,Animalanimal):
RedirectResponse
{
    validated=validated = validated=request→validated();

    if (request→hasFile('photo')) {
        // L'ancienne photo est remplacée : son fichier ne servira plus.
        if ($animal→photo !== null) {
            Storage::disk('public')→delete($animal→photo);
        }

        $validated['photo'] = $request→file('photo')→store('animals', 'public');
    }

    animal
→update(validated);//Aucunecasecoche´e:lechamptagsn'estpasenvoye´,sync([])retiretouslestrait
s.validated);

    // Aucune case cochée : le champ tags n'est pas envoyé, sync([]) retire tous les traits.

validated);//Aucunecasecoche´e:lechamptagsn'estpasenvoye´,sync([])retiretouslestraits.animal
→tags()→sync(validated['tags']??[]);returnredirect()
->route('animals.show',validated['tags'] ?? []);

    return redirect()
        →route('animals.show', validated['tags']??[]);returnredirect()
->route('animals.show',animal→id)
        →with('status', 'L\'animal a été modifié.');
}
```

Dans `destroy()`, effacez la photo avant de supprimer la ligne :

```php
public function destroy(Animal $animal): RedirectResponse
{
    // La base ne connaît que le chemin : le fichier se supprime à part.
    if ($animal→photo !== null) {
        Storage::disk('public')→delete($animal→photo);
    }

    // Les lignes de animal_tag suivent grâce à cascadeOnDelete().
    $animal→delete();

    return redirect()
        →route('animals.index')
        →with('status', 'L\'animal a été supprimé.');
}
```

Le `if` protège les animaux sans photo. `delete()` attend un chemin, et s'arrête sur `null` :

```
TypeError
League\Flysystem\Filesystem::delete(): Argument #1 ($location) must be of type string, null
given, called in vendor\laravel\framework\src\Illuminate\Filesystem\FilesystemAdapter.php on
line 610
```

Dans `storage/app/public/animals`, effacez à la main les fichiers de trop : gardez seulement celui que cite la colonne de Rex, avec la commande `tinker` de la section 5. Remplacez ensuite de nouveau la photo de Rex, puis ajoutez et supprimez Bruno. Le dossier ne garde que la photo actuelle de Rex :

```
eaKbHpXrccnKDSQ5qw4vN57RZileX3GX4KBYt6Rl.jpg
```

Modifiez enfin la description de Rex sans choisir de fichier : la photo reste. Laravel écarte le champ `photo` vide, `$validated` n'a pas de clé `photo`, et `update()` ne touche pas à la colonne.

8. Les tests

Créez `tests/Feature/PhotoTest.php` et copiez-y ce fichier en entier :

PHP

```php
<?php

use App\Models\Animal;
use App\Models\Shelter;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;

/*
| Chapitre 18 : une photo par animal.
| Storage::fake('public') remplace le disque public par un dossier temporaire, vidé à chaque test :
| vos fichiers de storage/app/public ne sont jamais touchés.
| UploadedFile::fake()->image('rex.jpg') fabrique une vraie image JPEG de 10 × 10 pixels.
*/

test('les deux formulaires envoient en multipart/form-data et proposent un champ photo', function (): void {
    animal=Animal::factory()->create();animal = Animal::factory()->create();

    animal=Animal::factory()->create();this->get('/animaux/nouveau')
        ->assertOk()
        ->assertSee('enctype="multipart/form-data"', false)
        ->assertSee('type="file"', false)
        ->assertSee('name="photo"', false);

    this->get("/animaux/{$animal->id}/modifier")
        ->assertOk()
        ->assertSee('enctype="multipart/form-data"', false)
        ->assertSee('name="photo"', false);
})->group('photo');

test('la photo envoyée à l\'ajout est rangée sur le disque public et son chemin en base', function (): void {
    Storage::fake('public');
    shelter = Shelter::factory()->create();

    this->post('/animaux',validAnimal(this->post('/animaux',
validAnimal(this->post('/animaux',validAnimal(shelter, [
        'photo' => UploadedFile::fake()->image('rex.jpg'),
    ]))->assertSessionHasNoErrors();

    animal=Animal::where('name','Rex')->firstOrFail();expect(animal = Animal::where('name',
'Rex')->firstOrFail();

    expect(animal=Animal::where('name','Rex')->firstOrFail();expect(animal->photo)
->toStartWith('animals/')->toEndWith('.jpg');
    Storage::disk('public')->assertExists(animal->photo);
})->group('photo');

test('la photo est facultative', function (): void {
    Storage::fake('public');
    shelter = Shelter::factory()->create();

    this->post('/animaux',validAnimal(this->post('/animaux',
validAnimal(this->post('/animaux',validAnimal(shelter))->assertSessionHasNoErrors();
```

```
    expect(Animal::where('name', 'Rex')->firstOrFail()->photo)->toBeNull();
})->group('photo');

test('un PDF est refusé, avec un message en français', function (): void {
    Storage::fake('public');
    shelter=Shelter::factory()->create();shelter = Shelter::factory()->create();

    shelter=Shelter::factory()->create();this->post('/animaux', validAnimal(shelter,
['photo'=>UploadedFile::fake()->create('fiche-rex.pdf',100,'application/pdf'),]))
->assertSessionHasErrors(['photo'=>'Lechampphotodoite^treuneimage.']);shelter, [
        'photo' => UploadedFile::fake()->create('fiche-rex.pdf', 100, 'application/pdf'),
    ]))->assertSessionHasErrors(['photo' => 'Le champ photo doit être une image.']);

    shelter,['photo'=>UploadedFile::fake()
->create('fiche-rex.pdf',100,'application/pdf'),]))
->assertSessionHasErrors(['photo'=>'Lechampphotodoite^treuneimage.']);this
->assertDatabaseCount('animals', 0);
    expect(Storage::disk('public')->allFiles())->toBeEmpty();
})->group('photo');

test('une image de plus de 2 Mo est refusée', function (): void {
    Storage::fake('public');
    shelter=Shelter::factory()->create();shelter = Shelter::factory()->create();

    shelter=Shelter::factory()->create();this->post('/animaux', validAnimal(shelter,
['photo'=>UploadedFile::fake()->image('rex.jpg')->size(3000),]))
->assertSessionHasErrors('photo');shelter, [
        'photo' => UploadedFile::fake()->image('rex.jpg')->size(3000),
    ]))->assertSessionHasErrors('photo');

    shelter,['photo'=>UploadedFile::fake()->image('rex.jpg')->size(3000),]))
->assertSessionHasErrors('photo');this->assertDatabaseCount('animals', 0);
})->group('photo');

test('la fiche et la liste affichent la photo, sinon le pictogramme de l\'espèce', function
(): void {
    rex=Animal::factory()->create(['name'=>'Rex','species'=>'chien',
'photo'=>'animals/rex.jpg']);rex = Animal::factory()->create(['name' => 'Rex', 'species' =>
'chien', 'photo' => 'animals/rex.jpg']);
    rex=Animal::factory()->create(['name'=>'Rex','species'=>'chien',
'photo'=>'animals/rex.jpg']);minette = Animal::factory()->create(['name' => 'Minette',
'species' => 'chat', 'photo' => null]);

    this->get("/animaux/{$rex->id}")
        ->assertOk()
        ->assertSee('storage/animals/rex.jpg', false)
        ->assertSee('alt="Photo de Rex"', false);

    this->get("/animaux/{minette->id}")
        ->assertOk()
        ->assertSee('images/species/chat.svg', false);

    this->get('/animaux')
        ->assertOk()
        ->assertSee('storage/animals/rex.jpg', false)
```

```
        →assertSee('images/species/chat.svg', false);
})→group('photo');

test('modifier sans choisir de fichier garde la photo', function (): void {
    Storage::fake('public');
    shelter=Shelter::factory()->create();shelter = Shelter::factory()→create();
    shelter=Shelter::factory()->create();old = UploadedFile::fake()→image('rex.jpg')
→store('animals', 'public');
    animal=Animal::factory()->for(animal = Animal::factory()→for(animal=Animal::factory()
->for(shelter)→create(['photo' ⇒ old]);old]);

    old]);this→put("/animaux/{animal→id}", validAnimal(shelter, ['description' ⇒ 'Un
nouveau texte.']))
        →assertSessionHasNoErrors();

    expect(animal->fresh()->photo)->toBe(animal→fresh()→photo)→toBe(animal->fresh()
->photo)->toBe(old);
    Storage::disk('public')→assertExists(old);
})→group('photo');

test('remplacer la photo supprime l\'ancien fichier', function (): void {
    Storage::fake('public');
    shelter = Shelter::factory()→create();
    old=UploadedFile::fake()->image('rex.jpg')->store('animals','public');old =
UploadedFile::fake()→image('rex.jpg')→store('animals', 'public');
    old=UploadedFile::fake()->image('rex.jpg')->store('animals','public');animal =
Animal::factory()→for(shelter)->create(['photo'⇒shelter)→create(['photo' ⇒ shelter)
->create(['photo'⇒old]);

    this→put("/animaux/{$animal→id}", validAnimal(shelter, [
        'photo' ⇒ UploadedFile::fake()→image('rex-ete.jpg'),
    ]))→assertSessionHasNoErrors();

    new=new = new=animal→fresh()→photo;

    expect(new)->not->toBe(new)→not→toBe(new)->not->toBe(old);
    Storage::disk('public')→assertExists(new);Storage::disk('public')->assertMissing(new);
    Storage::disk('public')→assertMissing(new);Storage::disk('public')->assertMissing(old);
})→group('photo');

test('supprimer un animal supprime sa photo', function (): void {
    Storage::fake('public');
    photo=UploadedFile::fake()->image('rex.jpg')->store('animals','public');photo =
UploadedFile::fake()→image('rex.jpg')→store('animals', 'public');
    photo=UploadedFile::fake()->image('rex.jpg')->store('animals','public');animal =
Animal::factory()→create(['photo' ⇒ photo]);photo]);

    photo]);this→delete("/animaux/{animal→id}")→assertRedirect(route('animals.index'));

    this→assertModelMissing(animal);Storage::disk('public')->assertMissing(animal);
    Storage::disk('public')→assertMissing(animal);Storage::disk('public')
->assertMissing(photo);
})→group('photo');
```

Trois outils de Laravel rendent ces tests possibles :

- `Storage::fake('public')` remplace le disque `public` par un dossier temporaire, vidé avant chaque test. Vos photos de `storage/app/public` ne sont jamais touchées.
- `UploadedFile::fake()->image('rex.jpg')` fabrique un objet `UploadedFile` qui contient une vraie image, dessinée par l'extension GD de PHP. `->size(3000)` lui fait annoncer 3 000 Ko. `create('fiche-rex.pdf', 100, 'application/pdf')` fabrique un fichier qui n'est pas une image.
- `assertExists()` et `assertMissing()` vérifient qu'un fichier est présent ou absent sur le disque simulé.

`validAnimal()` vient de `tests/Pest.php`, comme dans les tests du chapitre 17. Lancez le groupe :

```bash
php artisan test --group=photo
```

```
   PASS  Tests\Feature\PhotoTest
  ✓ les deux formulaires envoient en multipart/form-data et proposent un champ photo
0.60s
  ✓ la photo envoyée à l'ajout est rangée sur le disque public et son chemin en base
0.07s
  ✓ la photo est facultative
0.03s
  ✓ un PDF est refusé, avec un message en français
0.04s
  ✓ une image de plus de 2 Mo est refusée
0.03s
  ✓ la fiche et la liste affichent la photo, sinon le pictogramme de l'espèce
0.05s
  ✓ modifier sans choisir de fichier garde la photo
0.03s
  ✓ remplacer la photo supprime l'ancien fichier
0.04s
  ✓ supprimer un animal supprime sa photo
0.04s

  Tests:    9 passed (39 assertions)
```

Si un test est rouge, ce tableau dit ce qu'il attend de votre code :

Test	Ce que vous écrivez
les deux formulaires envoient en multipart/form-data…	`enctype="multipart/form-data"` sur les deux `<form>`, et `<input type="file" name="photo">` dans `<x-animal-fields>`
la photo envoyée à l'ajout est rangée sur le disque public…	la colonne `photo`, `photo` dans `#[Fillable]`, et `store('animals', 'public')` dans `store()`
la photo est facultative	`nullable` dans la règle et dans la migration
un PDF est refusé, avec un message en français	la règle `image`
une image de plus de 2 Mo est refusée	la règle `max:2048`

Test	Ce que vous écrivez
la fiche et la liste affichent la photo…	`@if ($animal->photo !== null)` dans la fiche et dans la carte, avec `alt="Photo de {{ $animal->name }}"`
modifier sans choisir de fichier garde la photo	le `if ($request->hasFile('photo'))` dans `update()`
remplacer la photo supprime l'ancien fichier	`Storage::disk('public')->delete($animal->photo)` dans `update()`
supprimer un animal supprime sa photo	le même `delete()` dans `destroy()`

Lancez enfin `php artisan test` sans groupe : les 41 tests du chapitre 17 doivent rester verts. Dans `SupprimerTest`, les animaux n'ont pas de photo : sans le `if` de `destroy()`, ils s'arrêtent sur le `TypeError` de la section 7. Faites un commit et poussez. La CI lance tous les fichiers de `tests/Feature/`, le nouveau compris.

9. Ce qu'il ne faut pas faire

ATTENTION

- **Faire confiance au nom du fichier.** `getClientOriginalName()` et `getClientOriginalExtension()` renvoient ce que l'expéditeur a écrit, et n'importe qui peut envoyer un `rex.jpg` qui n'est pas une image. La règle `image` et `store()` lisent le contenu : gardez-les.
- **Ranger les fichiers directement dans `public/`.** Tout ce qui s'y trouve est servi tel quel, sans passer par Laravel, et le dossier `public/` est versionné. Le disque `public` et le lien `public/storage` gardent les fichiers des visiteurs à part.
- **Versionner `public/storage` ou `storage/app/public`.** Le lien ne fonctionne que sur votre ordinateur, et les photos n'ont rien à faire dans le dépôt. Ne retirez pas les lignes des deux `.gitignore`.
- **Oublier `max`.** Sans cette règle, seule la limite `upload_max_filesize` de PHP arrête un fichier, et elle peut valoir plusieurs gigaoctets.

Les fichiers téléversés posent d'autres problèmes de sécurité. Ils reviennent au chapitre « Les failles typiques du web ».

À retenir

- Un formulaire qui envoie un fichier porte `enctype="multipart/form-data"`, sinon le serveur ne reçoit que le nom du fichier.
- La base garde le chemin de la photo, et le fichier va sur le disque `public`, dans `storage/app/public`.

- `store('animals', 'public')` donne au fichier un nom tiré au hasard, avec l'extension lue dans son contenu.
- `php artisan storage:link` crée le lien `public/storage`, que Git ignore et que chaque installation doit recréer.
- Quand la photo change ou que l'animal disparaît, le code efface l'ancien fichier avec `Storage::disk('public')->delete()`.

À vérifier

Vérification	Comment
La photo s'affiche sur la fiche et dans la liste	la fiche de Rex et `/animaux`, après une modification avec photo
Un animal sans photo garde son pictogramme	la fiche de Minette
Le PDF est refusé en français	« Le champ photo doit être une image. » sous le champ
L'ancien fichier disparaît	`ls storage/app/public/animals`, après un remplacement puis après une suppression
Les tests du groupe `photo` sont verts	`php artisan test --group=photo`, puis `php artisan test`
`public/storage` n'est pas dans Git	`git status` ne cite ni `public/storage` ni `storage/app/public/animals`

L'examen propose la photo d'un objet trouvé parmi ses fonctionnalités supplémentaires. Elle se construit avec le même mécanisme : un champ de fichier, une règle `image`, le disque `public` et le lien `public/storage`.

3.19 Projet d'examen : Perdu de vue

Bloc Laravel · projet d'examen · remise le 1er novembre 2026 à 23 h 59

La gare de Wavre a un bureau des objets trouvés, à côté des guichets. Chaque jour, le personnel y apporte ce que les voyageurs oublient dans le hall, sur les quais et dans les trains. Au comptoir, l'agent enregistre l'objet, note où et quand il a été trouvé, choisit sa couleur et lui colle des étiquettes comme « fragile » ou « vivant ». Quand le propriétaire se présente, l'agent marque l'objet restitué, et l'objet quitte la liste. Une fiche créée en double se supprime. Le site n'a pas de compte : l'authentification arrive au cours suivant. Vous construisez ce site seul, depuis un projet Laravel vide, avec tout ce que le bloc Laravel vous a appris.

À la fin de ce projet, vous aurez montré que vous savez :

1. **créer** un projet Laravel et le versionner sur GitHub, une étape après l'autre,
2. **construire** une base de trois tables et une table de liaison, avec ses migrations, ses modèles, ses enums et ses données,
3. **écrire** les routes, les contrôleurs et les vues d'un site, à partir d'un tableau de routes imposées,

4. **valider** des formulaires d'ajout et de modification, et **afficher** leurs erreurs en français,
5. **mettre en forme** un site avec Tailwind CSS et des composants Blade.

1. Le projet

Le domaine

Le site manipule trois choses. Un **lieu** est un endroit de la gare où l'on ramasse des objets : le hall des guichets, un quai, un train. Un **objet** a été trouvé dans un seul lieu, à une date donnée, et porte une couleur. Une **étiquette** décrit un objet en un mot ou deux, et un objet peut en porter plusieurs. Une même étiquette sert à beaucoup d'objets.

Table	Colonnes	Relation
`places`	`name`, `kind` (texte, enum `PlaceKind`)	un lieu a plusieurs objets
`items`	`place_id`, `name` (60 caractères), `description` (facultative), `color` (texte, enum `Color`), `found_on` (date), `returned_at` (date, facultative)	un objet appartient à un lieu et porte des étiquettes
`tags`	`name`	une étiquette est portée par plusieurs objets
`item_tag`	`item_id`, `tag_id`, clé primaire composée des deux	la table de liaison

Les trois premières tables ont aussi `id` et les deux colonnes de dates de `timestamps()`. Les trois clés étrangères s'écrivent `constrained()→cascadeOnDelete()`, comme au chapitre 12.

Les modèles

Le modèle des étiquettes s'appelle `Tag`, comme celui des traits de caractère du refuge. `Item` a une méthode `isReturned()`, qui renvoie `true` quand `returned_at` contient une date. La clé étrangère `place_id` est une composition : supprimer un lieu supprime ses objets. La table `item_tag` relie les objets et leurs étiquettes, comme `animal_tag` au refuge du chapitre 17. Le diagramme de classes n'est pas fourni : vous le dessinez vous-même pour le rapport, avec les classes, les enums, les attributs et les multiplicités.

Les deux enums

La base enregistre la valeur de l'enum en texte, `platform` ou `red`. Dans le modèle, `casts()` change ce texte en cas de l'enum, comme pour l'espèce du refuge. Chaque enum a une méthode `label()` qui donne le texte affiché :

```php
<?php

namespace App\Enums;

enum PlaceKind: string
{
    case Hall = 'hall';
    case Platform = 'platform';
    case Train = 'train';

    public function label(): string
    {
        return match ($this) {
            self::Hall ⇒ 'Hall',
            self::Platform ⇒ 'Quai',
            self::Train ⇒ 'Train',
        };
    }
}
```

`Color` suit la même forme, avec ces douze cas :

Cas	Valeur en base	`label()`
`Black`	`black`	Noir
`White`	`white`	Blanc
`Grey`	`grey`	Gris
`Red`	`red`	Rouge
`Blue`	`blue`	Bleu
`Green`	`green`	Vert
`Yellow`	`yellow`	Jaune
`Orange`	`orange`	Orange
`Brown`	`brown`	Brun
`Pink`	`pink`	Rose
`Purple`	`purple`	Mauve
`Multicolor`	`multicolor`	Multicolore

Les données

Le seeder crée **au moins** 4 lieux, 8 étiquettes et 15 objets, dont 3 restitués. Un lieu est une ligne de la table `places`, avec un nom et une sorte : la solution en a quatre, le hall des guichets, deux quais et un train, et les trois sortes de l'enum `PlaceKind` y sont représentées. Écrivez des données fixes, sans

`fake()` dans le seeder : après `php artisan migrate:fresh --seed`, la liste montre toujours les mêmes objets, dans le même ordre. Les dates de découverte se situent en octobre 2026, aucune dans le futur, et un objet est restitué après avoir été trouvé.

Les objets sont les vôtres. Inventez-les drôles et crédibles, avec une description qui aiderait leur propriétaire à les reconnaître. Pour donner le ton : un nain de jardin prénommé Gérard, trouvé sur un quai, un poisson rouge dans son sachet, oublié dans un train, ou une seule chaussure de sport, pointure 44. Les objets visibles sur les captures de cet énoncé viennent de la solution : ne les recopiez pas.

Chaque modèle a aussi sa fabrique, `PlaceFactory`, `ItemFactory` et `TagFactory`, comme au chapitre 11, section 4. Les fabriques peuvent utiliser `fake()`. Elles servent aux tests, et à remplir la base si vous voulez voir une longue liste.

2. Ce que vous créez vous-même

Le projet et le dépôt

Il n'y a pas de dépôt de départ, et aucun test n'est fourni : les écrire est la supplémentaire S11. Dans le dossier de Herd ou de Laragon, créez le projet comme au chapitre 4, avec PHP 8.4 :

```bash
laravel new perdu-de-vue --git --no-boost
```

Répondez « No » au starter kit, puis choisissez Blade. Gardez Pest, SQLite et npm, proposés par défaut. Sur GitHub, créez ensuite un dépôt **public** nommé `perdu-de-vue`, et poussez-y le projet depuis VS Code, comme pour les katas. Commitez régulièrement, à chaque étape franchie, et poussez : les migrations, les modèles et les enums, les seeders, la liste et la fiche, l'ajout, la validation, la modification, la restitution, la suppression, la mise en forme. Un message de commit dit ce que l'étape ajoute, par exemple « Ajoute la restitution d'un objet ». Le `.gitignore` de Laravel écarte déjà `.env`, `vendor/`, `node_modules/` et `database/database.sqlite` : ne le modifiez pas.

Quelques réglages se font au début du projet. Installez les traductions françaises, comme au chapitre 15, section 4, et enregistrez le dossier `lang/` dans un commit :

```bash
composer require laravel-lang/common --dev
php artisan lang:add fr
```

Mettez `APP_LOCALE=fr` et `APP_FAKER_LOCALE=fr_BE` dans `.env` et dans `.env.example` : la première pour les messages de validation, la seconde pour que `fake()` produise des noms et des adresses belges dans les fabriques. Dans `config/app.php`, réglez `'timezone' ⇒ 'Europe/Brussels'` : en UTC, `today()` donne encore la date de la veille pendant la première heure après minuit en hiver, et les deux premières en été.

Pint est l'outil de mise en forme du code PHP livré avec Laravel. Il n'a pas été vu au cours. `vendor/bin/pint` réécrit vos fichiers selon les conventions de Laravel, indentation, espaces, ordre des `use`, sans toucher à ce qu'ils font. `vendor/bin/pint --test` les contrôle sans rien modifier et liste ceux

qui s'écartent des conventions. Sous PowerShell, les deux commandes s'écrivent `.\vendor\bin\pint`. Lancez la mise en forme avant chaque commit. Créez à la racine un fichier `pint.json` qui écarte `lang/` du contrôle :

```json
{
    "preset": "laravel",
    "exclude": [
        "lang"
    ]
}
```

Sans ce fichier, Pint signale les six fichiers de `lang/fr/` produits par Laravel-Lang, que vous n'avez pas écrits.

Ce que vous pouvez recopier

Les quatre composants de champ de la friterie et du refuge, `<x-label>`, `<x-input>`, `<x-select>` et `<x-textarea>`, peuvent être recopiés tels quels. Prenez les quatre fichiers dans `resources/views/components/` de `friterie-26` ou de `refuge-26`. Le reste des deux dépôts, le layout, la navigation, les boutons, les cartes et les badges, peut vous servir de point de départ : reprenez ce qui vous est utile et modifiez-le pour qu'il colle à « Perdu de vue », ses pages, ses couleurs et son vocabulaire.

Ce que vous écrivez	Où
les quatre migrations	`database/migrations/`
les enums `PlaceKind` et `Color`	`app/Enums/`
les modèles `Place`, `Item` et `Tag`, avec leurs relations, `#[Fillable]` et `casts()`	`app/Models/`
les fabriques et les seeders	`database/factories/`, `database/seeders/`
les routes	`routes/web.php`
`ItemController` et `PlaceController`	`app/Http/Controllers/`
la Form Request `ItemRequest`	`app/Http/Requests/`
le layout, vos composants et les vues des pages	`resources/views/`

3. Les routes imposées

Le verbe, l'adresse, le nom de la route et la méthode du contrôleur sont imposés. Les adresses sont en français, les noms et les méthodes en anglais. Le paramètre s'écrit `{item}` ou `{place}`, avec la liaison de modèle du chapitre 15, section 6 : un objet inconnu répond 404 sans une ligne de plus. Posez `whereNumber` sur le paramètre, ou déclarez `/objets/nouveau` avant `/objets/{item}`, pour éviter le piège de l'ordre du chapitre 7.

Verbe	Adresse	Nom	Méthode
`ANY`	`/`		redirection vers `/objets`
`GET`	`/objets`	`items.index`	`ItemController@index`
`GET`	`/objets/nouveau`	`items.create`	`ItemController@create`
`POST`	`/objets`	`items.store`	`ItemController@store`
`GET`	`/objets/{item}`	`items.show`	`ItemController@show`
`GET`	`/objets/{item}/modifier`	`items.edit`	`ItemController@edit`
`PUT`	`/objets/{item}`	`items.update`	`ItemController@update`
`PATCH`	`/objets/{item}/restitution`	`items.return`	`ItemController@return`
`DELETE`	`/objets/{item}`	`items.destroy`	`ItemController@destroy`
`GET`	`/lieux/{place}`	`places.show`	`PlaceController@show`
`GET`	`/composants`	`styleguide`	la vue `styleguide`

La première et la dernière route n'ont pas de contrôleur à écrire. Vous les avez vues au chapitre 7. `Route::redirect()` répond à tous les verbes, et `route:list` affiche `ANY` :

```php
Route::redirect('/', '/objets');

Route::view('/composants', 'styleguide')->name('styleguide');
```

La page `/composants` montre chacun de vos composants dans ses variantes, comme celle du refuge. Elle vous sert de référence pendant le projet, et au correcteur pour voir vos composants d'un coup d'œil. `return` est un mot-clé de PHP, mais une méthode peut porter ce nom : `public function return(Item $item)` s'écrit sans erreur. Une fois les routes écrites, `php artisan route:list --except-vendor` affiche « Showing [11] routes ».

4. Les fonctionnalités obligatoires

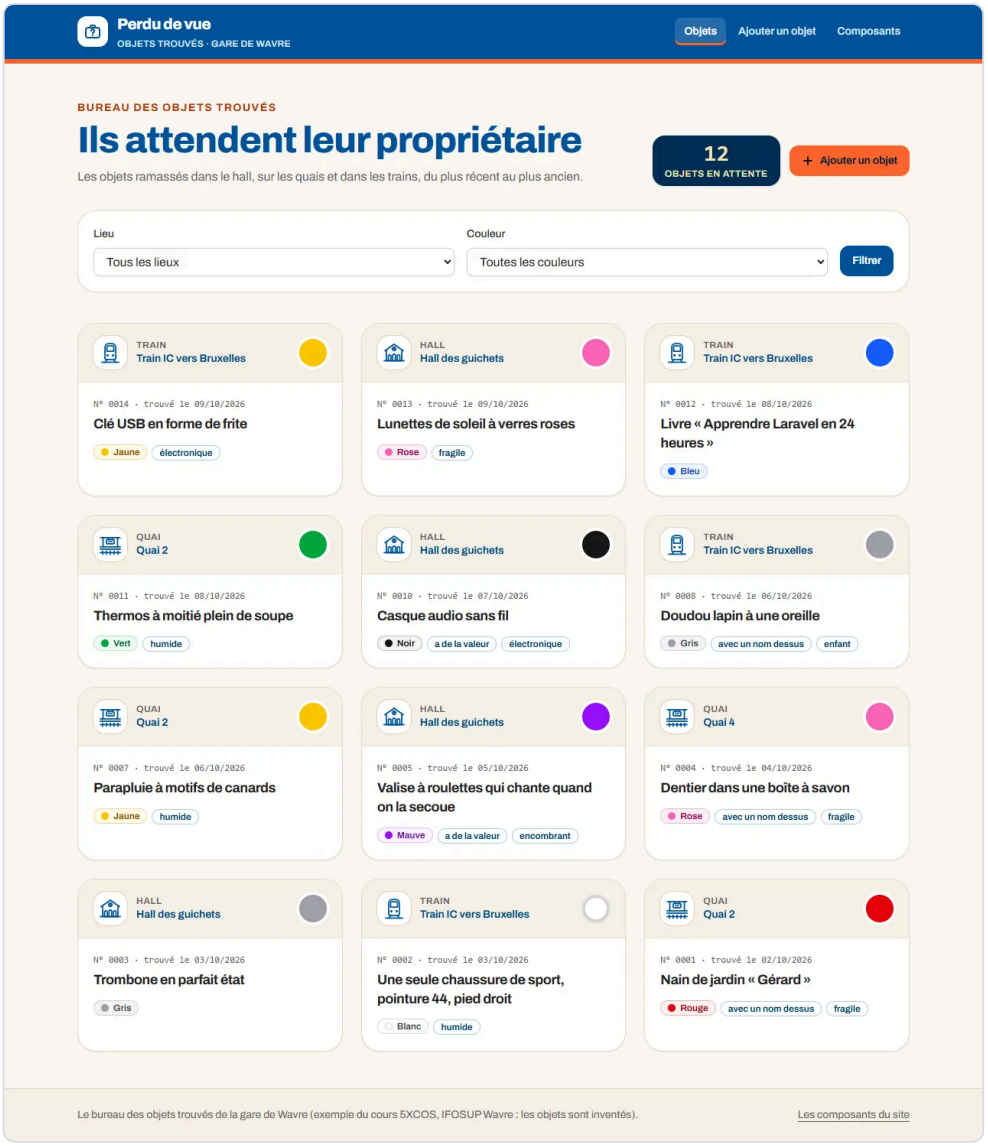
Tout le monde rend ce socle. Le tableau dit ce que le correcteur vérifie pour chaque fonctionnalité. Les numéros reviennent dans le canevas du rapport.

#	Fonctionnalité	Ce qui est vérifié
O1	Le projet	créé avec `laravel new perdu-de-vue --git --no-boost`, PHP 8.4, Blade, Pest, SQLite, dépôt GitHub public, des commits réguliers
O2	La base	les quatre tables, les clés étrangères en cascade, l'enum `PlaceKind` sur `kind`, l'enum `Color` sur `color`, `casts()` sur les enums et les dates
O3	Les données	les trois fabriques, un seeder fixe d'au moins 4 lieux, 8 étiquettes et 15 objets dont 3 restitués

#	Fonctionnalité	Ce qui est vérifié
O4	La liste	`GET /objets` montre les objets **non restitués**, du plus récent au plus ancien, avec le lieu, la date, la couleur et les étiquettes en badges, et les filtres `?lieu=` et `?couleur=`
O5	La fiche	`GET /objets/{item}` montre tout l'objet, son lieu en lien vers sa page, ses étiquettes et sa date de restitution s'il en a une
O6	Le lieu	`GET /lieux/{place}` montre le lieu et tous ses objets, restitués compris, chacun avec un badge « En attente » ou « Restitué »
O7	Ajouter	le formulaire avec `@csrf`, les quatre composants de champ et les cases des étiquettes, la validation, les erreurs en français sous chaque champ, `old()` sur tous les champs, un message flash
O8	Modifier	le formulaire pré-rempli, `@csrf` et `@method('PUT')`, la même Form Request `ItemRequest` que l'ajout, `sync()` des étiquettes, des cases qui gardent leur état après un refus
O9	Restituer	un formulaire avec `@csrf` et `@method('PATCH')` pose `returned_at` à `today()`, affiche un message flash, et l'objet quitte la liste. Un objet déjà restitué garde sa date
O10	Supprimer	un bouton `type="submit"` dans un formulaire avec `@csrf` et `@method('DELETE')`, une confirmation, un message flash, et les lignes de `item_tag` disparaissent avec l'objet
O11	La mise en forme	le layout `<x-layouts.app>` avec sa navigation, au moins trois composants réutilisés, les états `hover` et `focus`, des badges colorés selon l'enum
O12	Le code	identifiants en anglais, adresses et textes en français, contrôleurs courts, aucune requête dans les vues
O13	Le rapport et le carnet	le rapport selon le canevas, et le carnet métacognitif, en PDF (section 7)

La liste et ses filtres

Au-dessus de la liste, un formulaire en `GET` propose deux listes déroulantes, le lieu et la couleur. Envoyé, il ajoute `?lieu=2&couleur=red` à l'adresse : `lieu` porte l'`id` du lieu, `couleur` la valeur de l'enum. Les deux filtres se combinent. Une couleur inconnue, comme `?couleur=violet`, est ignorée : `Color::tryFrom()`, vue au chapitre 7 de la POO, renvoie alors `null`. Chargez les lieux et les étiquettes avec `with()`, pour éviter le piège N+1 du chapitre 12, section 6.



Sur chaque carte, la couleur apparaît deux fois : en pastille dans l'en-tête, et en badge avec son libellé. Seul le badge est demandé.

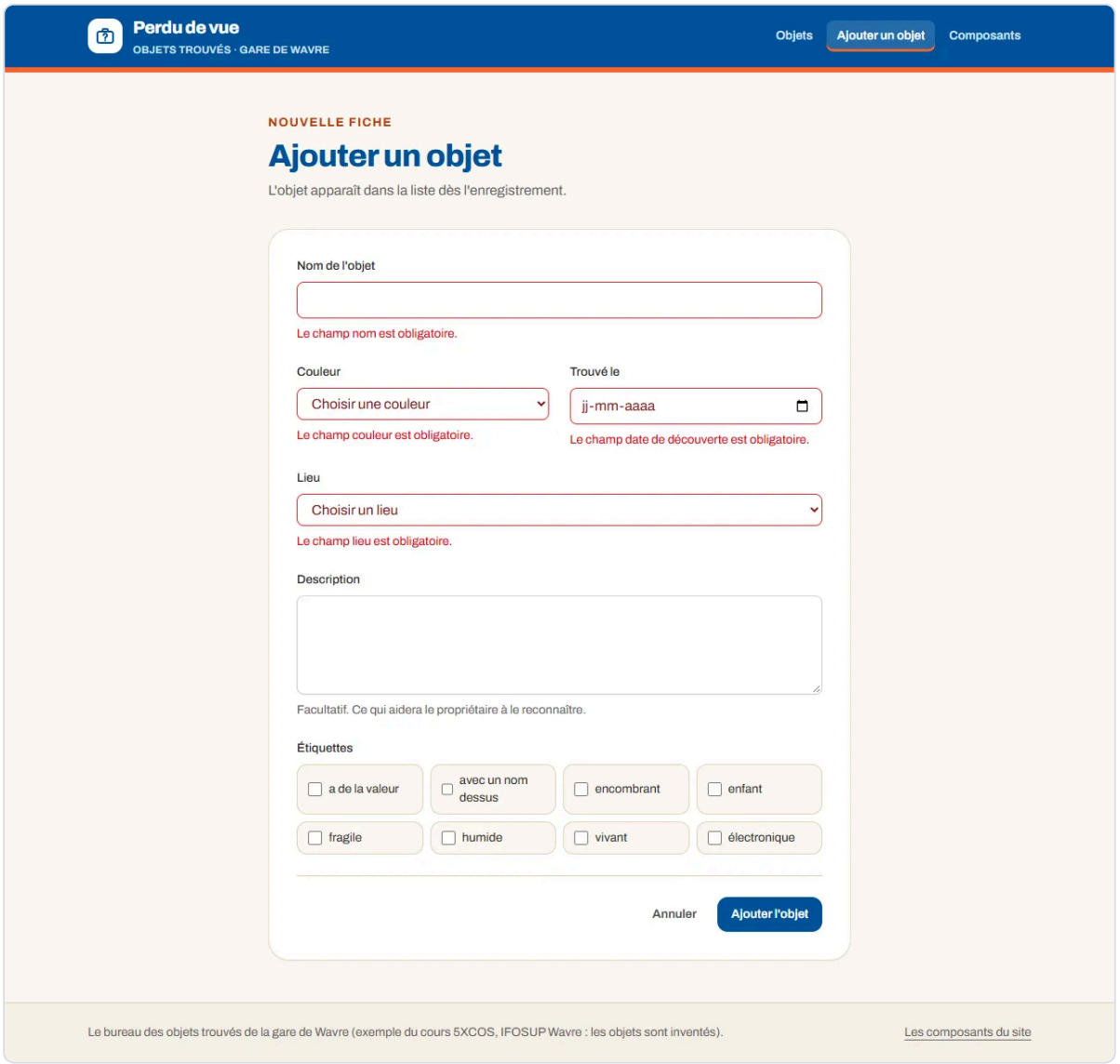
Les règles de validation

Les règles vivent dans `ItemRequest`, partagée par `store()` et `update()`, comme au chapitre 16, section 3. Les champs portent les mêmes noms à l'ajout et à la modification :

Champ	Élément	Règles	Nom dans les messages
`name`	`<input type="text">`	`required`, `string`, `min:2`, `max:60`	nom
`color`	`<select>` sur `Color::cases()`	`required`, `Rule::enum(Color::class)`	couleur
`found_on`	`<input type="date">`	`required`, `date`, `before_or_equal:today`	date de découverte

Champ	Élément	Règles	Nom dans les messages
`place_id`	`<select>` sur les lieux	`required`, `integer`, `exists:places,id`	lieu
`description`	`<textarea>`	`nullable`, `string`, `max:1000`	description
`tags`	cases `name="tags[]"` `value="{{ $tag->id }}"`	`nullable`, `array`	étiquettes
`tags.*`		`integer`, `exists:tags,id`	étiquette

Les noms français des champs vont dans le tableau `attributes` de `lang/fr/validation.php`, comme au chapitre 15, section 4, ou dans la méthode `attributes()` de la Form Request. Pour la date future, remplacez le message « ... au today » par le vôtre dans `messages()`, comme au kata 17.4. Aucun champ ne porte l'attribut HTML `required` : envoyé vide, le formulaire part au serveur, et quatre messages s'affichent.



Après un refus, les cases des étiquettes gardent l'état choisi avant l'envoi, toutes décochées comprises. Reprenez `session()->hasOldInput()` du kata 17.5 : sans lui, une liste vide après un refus ferait revenir les étiquettes enregistrées.

Restituer et supprimer

Sur la fiche d'un objet en attente, un formulaire d'un bouton envoie la restitution avec `@method('PATCH')`. Sur la fiche d'un objet restitué, ce bouton disparaît. Un bouton absent ne suffit pourtant pas : une fiche ouverte dans un deuxième onglet garde le sien. `return()` vérifie donc `isReturned()` avant d'écrire, et un objet déjà restitué garde sa première date, avec un message qui le dit.

La suppression demande une confirmation, avec `onsubmit="return confirm('…')"` comme au chapitre 16, section 7. Elle redirige vers la liste, et le message flash s'affiche dans le layout. Les textes des messages sont libres. Ceux de la solution donnent le ton :

Action	Message flash de la solution
ajout	L'objet a été ajouté.
modification	L'objet a été modifié.
restitution	« Parapluie à motifs de canards » a retrouvé son propriétaire.
restitution d'un objet déjà restitué	Cet objet a déjà été restitué le 06/10/2026.
suppression	L'objet a été supprimé.

5. Les fonctionnalités supplémentaires

Les fonctionnalités supplémentaires sont facultatives, et aucune n'est vue en séance. Chacune vous fait apprendre une notion seul, avec la documentation ou le chapitre indiqué. **Une fonctionnalité bien faite vaut mieux que trois bâclées.** Bien faite veut dire : dans le style du reste du site, validée quand elle reçoit une saisie, et présentée dans le rapport. Une supplémentaire ne compense jamais une obligatoire manquante. Gardez les numéros du tableau : le canevas du rapport les reprend. Les étoiles donnent l'effort estimé.

#	Fonctionnalité	Ce que vous apprenez	À lire	Effort
S1	Photo de l'objet	`enctype`, `$request->file()`, `store()`, le disque `public`, `storage:link`, la règle `image`, la suppression du fichier	le chapitre 18 « Téléverser une image », qui le fait sur le refuge	★
S2	Pagination	`paginate(9)`, `{{ $items->links() }}`, les filtres gardés avec `withQueryString()`	pagination	★
S3	Recherche par texte	un champ `?q=`, `where('name', 'like', …)`, combiné aux filtres	where	★
S4	Tri de la liste	`?tri=date` ou `?tri=nom`, une valeur comparée à une liste fixe avant `orderBy`	ordering	★

#	Fonctionnalité	Ce que vous apprenez	À lire	Effort
S5	Dates en français	« il y a 3 jours » avec `diffForHumans()`, « 2 octobre 2026 » avec `isoFormat('D MMMM YYYY')`	Carbon, localisation	★
S7	Déclaration de perte	une quatrième table `claims` : un voyageur décrit ce qu'il a perdu sur `/perdu`, l'agent relie la déclaration à un objet, `item_id` facultatif	one to many	★★
S9	Page de statistiques	les objets par lieu et par mois, `selectRaw`, `groupBy`, des barres en largeur Tailwind sans bibliothèque	aggregates, grouping	★★
S11	Tests Pest des routes	un test par route imposée, `assertSee`, `assertDatabaseHas`, la CI GitHub Actions de `refuge-26` (`.github/workflows/tests.yml`)	HTTP tests, database testing	★★
S13	Mode sombre	la variante `dark:` sur le layout et les composants, comme au bonus 14.8	dark mode	★
S14	Galerie de photos	plusieurs photos par objet, une table `photos`, une photo de couverture, la suppression d'une photo, à la suite de S1	file uploads	★★★
S15	Mise en ligne	le site sur un hébergement, l'adresse dans le rapport	les chapitres « Hébergement OVH » et « Hébergement O2Switch » du cours	★★★
S16	Interface en deux langues	`lang/en`, `__()` sur les textes affichés, une langue choisie et gardée en session	localization	★★

Faites chaque supplémentaire dans sa propre suite de commits, ou dans une branche fusionnée avant la remise. Si une supplémentaire casse une obligatoire, retirez-la : le socle passe avant.

6. La mise en forme

La mise en forme est libre, dans un cadre précis. Chaque page s'affiche dans le layout `<x-layouts.app>`, avec un en-tête, une navigation et le message flash. Le lien de la page ouverte se distingue des autres dans la navigation. Au moins trois composants sont écrits par vous et réutilisés sur plusieurs pages : un bouton `<x-button>` avec ses variantes, un badge `<x-badge>` et une carte `<x-card>`. Un composant `<x-item-card>`, partagé par la liste et la page d'un lieu, évite de recopier la carte d'un objet.

Les badges prennent leur couleur de l'enum. Une méthode de l'enum renvoie les classes Tailwind de chaque cas, écrites en entier, car Tailwind ne voit pas un nom de classe assemblé (chapitre 13) :

```php
public function badgeClasses(): string
{
    return match ($this) {
        self::Hall => 'bg-sky-50 text-sky-800',
        self::Platform => 'bg-amber-50 text-amber-900',
        self::Train => 'bg-orange-50 text-orange-800',
    };
}
```

Dans le composant, `@class` ajoute ces classes à celles du badge :

```xml
<span @class(['rounded-full px-2 py-0.5 text-xs font-semibold', $place->kind
->badgeClasses()])>
    {{ $place->kind->label() }}
</span>
```

Les liens et les boutons ont un état `hover` et un contour au clavier avec `focus-visible`, comme au chapitre 14. Un champ refusé a une bordure rouge et son message juste en dessous. Un site lisible sur un téléphone compte dans le degré de maîtrise : à 375 pixels de large, la liste montre une carte par ligne, les filtres s'empilent et la navigation reste utilisable. Vérifiez chaque page avec F12 puis Ctrl+Maj+M, comme au chapitre 14, section 1.

La palette est libre. La solution reprend les couleurs d'une signalétique de gare, et votre site peut prendre une tout autre direction, tant que les couleurs restent les mêmes sur toutes les pages et que le texte reste lisible.

7. Le rapport et le carnet

Téléchargez les deux documents Word, remplissez-les, puis enregistrez-les en PDF.

Canevas du rapport (Word) DOCX

Le canevas du rapport compte neuf sections : introduction, cahier des charges réalisé, base de données, fonctionnalités, validation et sécurité, mise en forme, difficultés rencontrées, autocritique et conclusion. Il contient le tableau O1 à O13 à cocher, le tableau des supplémentaires retenues, la place du diagramme de classes, qui est obligatoire, et la liste des captures attendues. Remplacez chaque consigne en italique par votre contenu. Tenez un journal de vos difficultés au fil du projet, avec la date, le problème et la solution : la section 7 se remplira presque toute seule. Pour cocher une case des deux documents, remplacez le carré par un « x », ou par ☑ si vous préférez.

Carnet métacognitif (Word) DOCX

Le carnet rappelle d'abord ce qui distingue un assistant d'un agent, et ce que chacun a le droit de faire. Il décrit ensuite votre usage de l'IA en trois parties : la déclaration d'usage, avec vos outils et votre niveau d'usage de 0 à 4, la démarche et les vérifications, puis un regard critique. Il demande un prompt qui a marché, un qui a raté, une vérification qui a révélé un problème, et cinq à dix lignes de votre projet expliquées avec vos mots. Il se termine par une attestation signée. Le carnet est individuel. Un usage intensif et bien vérifié, décrit honnêtement, vaut mieux qu'un usage caché.

Votre nom et votre prénom figurent sur la première page de chaque document et dans le nom de chaque fichier : `rapport-5xcos-NOM-Prenom.pdf` et `carnet-5xcos-NOM-Prenom.pdf`.

8. L'IA

Un **assistant** est une IA à qui vous parlez, dans un chat ou dans l'éditeur : vous lui posez une question, il répond, et c'est vous qui écrivez dans vos fichiers. Un **agent** est une IA qui agit à votre place : il lit vos fichiers, en crée, les modifie et lance des commandes. La règle du cours tient en deux phrases. L'assistant est autorisé pour le code : expliquer une notion ou une erreur, aider à déboguer, proposer une solution que vous recopiez et comprenez. L'agent est interdit pour écrire ou modifier du code, et autorisé pour lire votre projet et répondre à vos questions sur lui.

Pour le rapport et le carnet, l'assistant est autorisé en **coproduction encadrée** : les idées et le contenu sont les vôtres, l'IA corrige, reformule ou structure, et vous vérifiez que tout ce qui est écrit est vrai. Vous devez comprendre et pouvoir expliquer chaque ligne que vous remettez, du code comme du rapport. Le carnet le montre, avec ses lignes expliquées et ses vérifications.

Vérifiez ce qu'un assistant propose avec la documentation de Laravel 13 et avec le navigateur. Un assistant écrit souvent pour une ancienne version : un `app/Http/Kernel.php` qui n'existe plus depuis Laravel 11, `protected $fillable` à la place de `#[Fillable]`, une route en fonction anonyme quand le tableau impose un contrôleur. Déclarez votre usage dans le carnet.

9. La remise et la note

Le projet est individuel. Remettez trois pièces dans le devoir dédié du groupe Teams du cours, au plus tard le **1er novembre 2026 à 23 h 59** :

1. le lien de votre dépôt GitHub public,
2. le rapport en PDF,
3. le carnet métacognitif en PDF.

Il n'y a pas de défense orale. Le dépôt est lu tel qu'il est à l'heure de la remise : ne poussez plus rien ensuite. Avant de remettre, ouvrez le lien du dépôt dans une fenêtre de navigation privée, pour vérifier qu'il est bien public.

ATTENTION

Les trois pièces sont indispensables à la recevabilité du projet. Sans rapport en PDF portant votre nom, l'acquis d'apprentissage 1.4 est non acquis. Sans carnet en PDF portant votre nom, l'acquis 1.5 est non acquis. Sans dépôt accessible, l'acquis 1.3 est non acquis. Un seul acquis non acquis suffit pour l'ajournement.

La grille d'évaluation compte deux parties de 50 points. La première vérifie que les cinq acquis d'apprentissage sont atteints : les fonctionnalités, la qualité du code, Git, le rapport et le carnet. Chaque acquis est décrit en quatre niveaux. Quand les cinq sont atteints, la seconde partie mesure le degré de maîtrise sur quatre critères : l'interface, la robustesse, l'architecture et l'analyse du rapport. Lisez la grille avant de commencer, et relisez-la avant de remettre.

Grille d'évaluation 2026-2027 (PDF) PDF

À vérifier avant de remettre

Cette liste reprend ce que les acquis d'apprentissage exigent. Le reste, un site lisible sur un téléphone, Pint sans remarque, un historique Git qui raconte le projet, compte dans le degré de maîtrise, pas dans la réussite.

Vérification	Comment
La base se reconstruit	`php artisan migrate:fresh --seed` passe sans erreur, et la liste montre au moins 12 objets en attente
Les onze routes existent	`php artisan route:list --except-vendor` affiche « Showing [11] routes », avec les noms du tableau
Le formulaire vide est refusé	quatre messages en français, chacun sous son champ
La modification est pré-remplie	la date s'affiche dans son champ, les étiquettes de l'objet sont cochées
La restitution retire l'objet de la liste	restituez un objet depuis sa fiche : la liste ne le montre plus, la page de son lieu le montre avec le badge « Restitué »
La suppression vide la liaison	après la suppression de l'objet 5, `DB::table('item_tag')->where('item_id', 5)->count()` renvoie `0` dans `php artisan tinker`
Le dépôt est public	ouvert dans une fenêtre de navigation privée, il s'affiche sans connexion
Les PDF sont nommés	votre nom sur la première page et dans le nom des deux fichiers, `rapport-5xcos-NOM-Prenom.pdf` et `carnet-5xcos-NOM-Prenom.pdf`

3.20 Comment installer un projet Laravel existant sur un nouvel ordinateur

Ce chapitre vous guidera à travers les étapes nécessaires pour installer un projet Laravel existant sur un nouvel ordinateur. Nous supposerons que vous avez déjà installé les outils requis et que vous utilisez un serveur de développement local comme Laragon ou Laravel Valet/Herd.

Prérequis

Avant de commencer, assurez-vous que les éléments suivants sont installés sur votre ordinateur :

- **PHP** : Le langage de programmation sur lequel Laravel est construit, avec les extensions requises.
- **Composer** : Un gestionnaire de dépendances pour PHP, utilisé pour installer les packages nécessaires.
- **Node.js et npm** : Node.js est un environnement d'exécution JavaScript, et npm est son gestionnaire de packages. Ils sont utilisés pour compiler les assets front-end de Laravel.
- **Un serveur de développement local** : Outils comme **Laragon** ou **Laravel Valet/Herd** facilitent le développement local en gérant les configurations de serveur.

Étapes d'installation

1. Cloner le dépôt du projet Laravel

La première étape consiste à obtenir une copie du code source du projet. Vous pouvez le faire en clonant le dépôt Git du projet :

```bash
git clone https://github.com/nom-utilisateur/nom-du-projet.git
```

Cette action télécharge le code source du projet sur votre machine locale, vous permettant de travailler avec le même code que celui du dépôt distant.

2. Naviguer vers le répertoire du projet

Après avoir cloné le dépôt, déplacez-vous dans le répertoire du projet :

```bash
cd nom-du-projet
```

Cela vous positionne dans le dossier du projet où vous exécuterez les commandes suivantes.

3. Installer les dépendances PHP avec Composer

Installez les packages PHP nécessaires en exécutant :

```bash
composer install
```

Cette commande lit le fichier `composer.json` et installe toutes les bibliothèques et dépendances PHP dont le projet a besoin pour fonctionner.

4. Copier le fichier `.env.example` en `.env`

Le fichier `.env` contient les variables d'environnement spécifiques à votre application, comme les informations de base de données. Créez-en une copie à partir de l'exemple fourni :

```bash
cp .env.example .env
```

Cette action crée un fichier `.env` que vous pourrez personnaliser pour votre environnement local.

5. Générer une nouvelle clé d'application

Générez une clé unique pour votre application Laravel :

```bash
php artisan key:generate
```

La clé d'application est utilisée pour sécuriser les données chiffrées et les sessions. Cette commande met à jour la variable `APP_KEY` dans votre fichier `.env`.

6. Configurer les paramètres de base de données

Ouvrez le fichier `.env` avec un éditeur de texte et configurez les informations de connexion à votre base de données :

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=nom_de_votre_base_de_donnees
DB_USERNAME=votre_nom_d_utilisateur
DB_PASSWORD=votre_mot_de_passe
```

Ces paramètres permettent à Laravel de se connecter à votre base de données pour effectuer des opérations comme les migrations et les requêtes.

7. Migrer la base de données

Créez les tables nécessaires dans votre base de données en exécutant :

```bash
php artisan migrate
```

Les migrations sont des scripts qui créent ou modifient la structure de la base de données. Si la base de données spécifiée n'existe pas, la commande vous proposera de la créer.

8. Peupler la base de données avec des seeders (si disponibles)

Si le projet contient des seeders pour insérer des données initiales, exécutez :

```bash
php artisan db:seed
```

Les seeders remplissent la base de données avec des données fictives ou initiales, utiles pour les tests ou pour démarrer l'application avec des données par défaut.

9. Créer un lien symbolique pour le stockage (si nécessaire)

Si le projet utilise le stockage de fichiers (par exemple, pour les uploads), créez un lien symbolique vers le répertoire public :

```bash
php artisan storage:link
```

Cette commande crée un lien symbolique entre `storage/app/public` et `public/storage`, permettant un accès web aux fichiers stockés.

10. Installer les dépendances Node.js

Installez les packages front-end nécessaires avec npm :

```bash
npm install
```

Cette commande lit le fichier `package.json` et installe toutes les dépendances JavaScript requises pour le front-end de l'application.

11. Compiler les assets front-end

Compilez les fichiers CSS et JavaScript du projet :

- Pour le développement (avec source maps et sans minification) :

  ```bash
  npm run dev
  ```

- Pour la production (optimisé et minifié) :

  ```bash
  npm run build
  ```

La compilation des assets génère les fichiers optimisés nécessaires pour le front-end de l'application, en combinant et en minifiant le code pour de meilleures performances.

12. Démarrer le serveur local

Avec votre serveur de développement local (Laragon, Laravel Valet/Herd), votre application devrait être automatiquement reconnue et servie. Assurez-vous que votre serveur est en cours d'exécution et accédez à l'application via votre navigateur web à l'adresse appropriée, par exemple :

```bash
http://nom-du-projet.test
```

Cette étape vous permet de visualiser et de tester votre application Laravel dans un environnement local.

Conclusion

Vous avez maintenant installé avec succès un projet Laravel existant sur votre nouvel ordinateur. Ce processus vous permet de configurer rapidement un environnement de développement local pour travailler sur le projet. N'oubliez pas de consulter la documentation du projet pour d'éventuelles instructions supplémentaires spécifiques à celui-ci.

3.21 Hébergement OVH

L'hébergement d'une application web est une étape cruciale pour sa mise en production. OVH, en tant que fournisseur d'hébergement, offre des services robustes pour déployer des applications Laravel. Ce chapitre guide à travers les différentes étapes nécessaires pour préparer, uploader et configurer une application Laravel sur un serveur OVH.

Étape 1 : Créer et importer la base de données

La gestion de la base de données est une étape cruciale pour le bon fonctionnement de votre application Laravel en production. Cette étape concerne la préparation de la base de données locale, son exportation, et son importation dans l'environnement de production OVH.

Si vous avez utilisé sqlite, il suffit d'uploader le fichier database.sqlite avec le reste du code.

1.1. Faire un dump de la base de données

Pour préparer votre base de données à être transférée sur le serveur OVH, suivez ces étapes :

1. **Ajouter un préfixe aux noms des tables** : Dans le fichier de configuration de la base de données de Laravel (généralement situé dans `config/database.php`), vous pouvez spécifier un préfixe pour les noms de tables pour éviter les conflits, surtout si vous partagez votre base de données avec d'autres applications. Par exemple, vous pourriez utiliser `mini_instagram_` comme préfixe.

config > database.php

```
5   return [
32      'connections' => [
44
45          'mysql' => [
46              'driver' => 'mysql',
47              'url' => env('DB_URL'),
48              'host' => env('DB_HOST', '127.0.0.1'),
49              'port' => env('DB_PORT', '3306'),
50              'database' => env('DB_DATABASE', 'laravel'),
51              'username' => env('DB_USERNAME', 'root'),
52              'password' => env('DB_PASSWORD', ''),
53              'unix_socket' => env('DB_SOCKET', ''),
54              'charset' => env('DB_CHARSET', 'utf8mb4'),
55              'collation' => env('DB_COLLATION', 'utf8mb4_unicode_ci'),
56              'prefix' => 'mini_instagram_',
57              'prefix_indexes' => true,
58              'strict' => true,
```

2. **Exécuter les migrations et les seeders** : Assurez-vous que votre base de données est à jour avec toutes les migrations et que les données initiales nécessaires sont chargées. Utilisez les commandes suivantes :

```bash
php artisan migrate:fresh --seed
```

3. **Créer un dump de la base de données** : Utilisez la commande `mysqldump` pour créer un dump de votre base de données MySQL. Assurez-vous d'inclure les paramètres nécessaires pour capturer l'ensemble de la structure et des données :

```bash
mysqldump -u username -p database_name > database_dump.sql
```

Remplacez `username` par votre nom d'utilisateur MySQL et `database_name` par le nom de votre base de données.

1.2. Importer le dump dans OVH via phpMyAdmin

Une fois le dump de votre base de données créé, vous devez l'importer dans votre base de données sur OVH :

1. **Accéder à phpMyAdmin** : Connectez-vous à phpMyAdmin via l'interface OVH. Vous trouverez généralement un lien direct dans le tableau de bord de votre hébergement.

2. **Importer le fichier** `.sql` : Dans phpMyAdmin, sélectionnez la base de données que vous venez de créer, puis utilisez l'onglet "Importer" pour charger le fichier `database_dump.sql`. Suivez les instructions pour compléter l'importation.

Étape 2 : Préparer le projet Laravel

2.1. Ouvrir le terminal et se positionner dans le dossier du projet

Avant de commencer à travailler sur votre projet Laravel, il est essentiel de savoir naviguer dans les dossiers à l'aide du terminal. Ouvrez votre terminal et utilisez les commandes `cd` (change directory) pour vous positionner dans le dossier de votre projet Laravel. Cette étape est fondamentale car toutes les commandes suivantes seront exécutées dans ce contexte.

2.2. Installer les dépendances avec Composer

Pour que votre application Laravel fonctionne correctement, elle doit avoir accès à certaines bibliothèques PHP, qui sont gérées par Composer. Exécutez la commande suivante dans votre terminal :

```bash
composer install --no-dev
```

Cette commande installe les dépendances nécessaires pour le fonctionnement de l'application en excluant les packages uniquement nécessaires en développement. C'est une pratique courante pour optimiser les performances et la sécurité de l'application en production.

2.3. Compiler les assets frontend

Laravel mix est un outil puissant qui vous permet de compiler et de minimiser les assets CSS et JavaScript. Pour préparer vos assets pour la production, exécutez :

```bash
npm run build
```

Il est important de ne pas uploader le dossier `node_modules` sur votre serveur de production car il contient des dépendances uniquement nécessaires en développement. Uploader ce dossier serait non seulement superflu mais aussi potentiellement risqué en termes de sécurité et de performance.

Nous avons maintenant préparé notre projet Laravel pour le déploiement. Passons à l'étape suivante qui consiste à uploader les fichiers sur le serveur OVH.

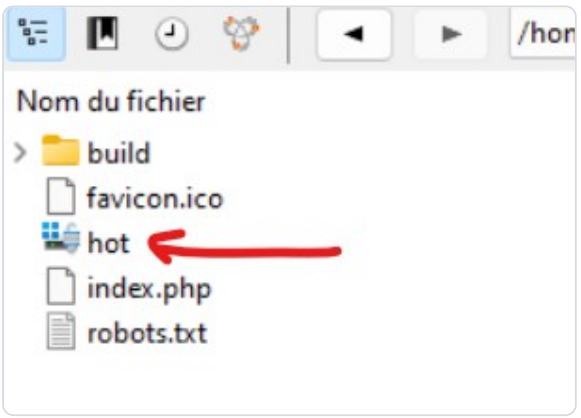
Étape 3 : Uploader les fichiers sur OVH

3.1. Utiliser un client FTP

Pour transférer votre projet Laravel sur le serveur OVH, vous aurez besoin d'un client FTP. Des clients tels que FileZilla ou Cyberduck sont recommandés pour leur facilité d'utilisation et leur fiabilité. Voici les étapes à suivre :

1. Ouvrez votre client FTP et configurez-le pour se connecter à votre espace d'hébergement OVH. Vous aurez besoin de l'adresse du serveur FTP, de votre nom d'utilisateur et de votre mot de passe, qui sont généralement fournis par OVH lors de la configuration de votre hébergement.
2. Une fois connecté, naviguez dans l'arborescence de votre espace local pour localiser le dossier de votre projet Laravel.
3. Transférez tous les fichiers de votre projet vers le serveur, à l'exception du dossier `node_modules`. Assurez-vous que tous les fichiers et dossiers nécessaires sont correctement uploadés, y compris le dossier `.env` et les dossiers `vendor` et `storage`.

Dans le dossier public, vous devez supprimer le fichier `hot` qui empecherait le front de fonctionner correctement :



3.2. Vérifier l'arborescence des fichiers

Après le transfert, il est crucial de vérifier que tous les fichiers et dossiers nécessaires sont correctement positionnés sur votre espace d'hébergement OVH. Voici les points clés à vérifier :

- Assurez-vous que le fichier `public/index.php` est bien présent et correctement placé car il est le point d'entrée de votre application Laravel.
- Vérifiez que les dossiers `bootstrap`, `config`, et `storage` sont bien présents et contiennent les fichiers attendus.

Une fois ces vérifications effectuées, votre application est prête à être configurée sur le serveur OVH.

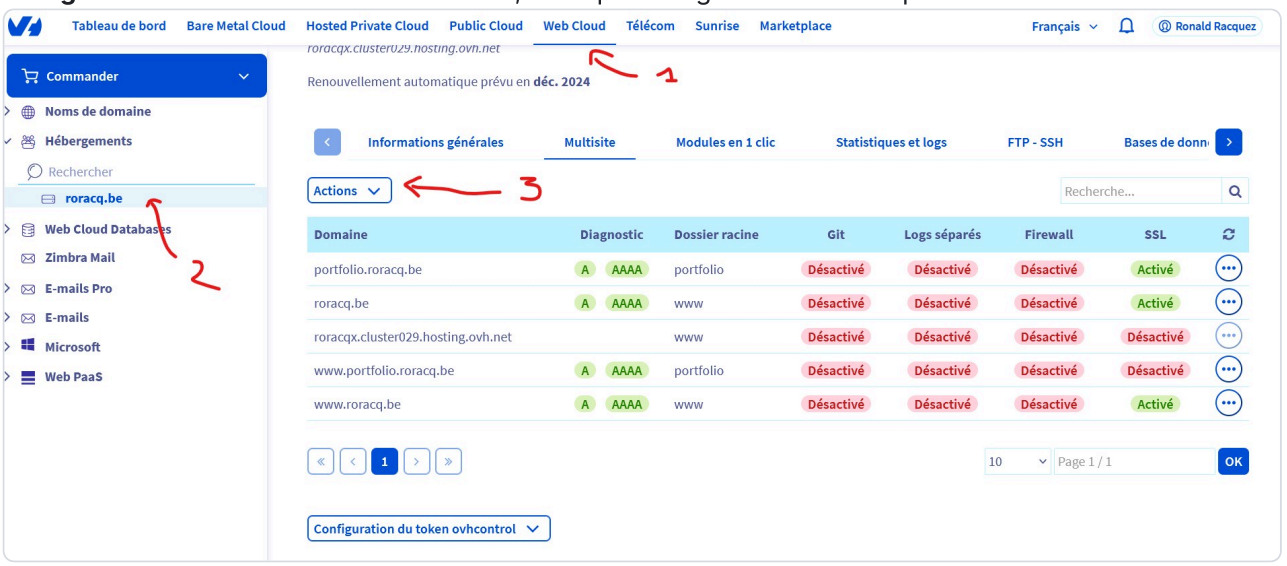
Étape 4 : Configurer l'hébergement sur OVH

Une fois que vos fichiers et votre base de données sont en place, il est temps de configurer l'hébergement pour s'assurer que votre application Laravel fonctionne correctement sur OVH.

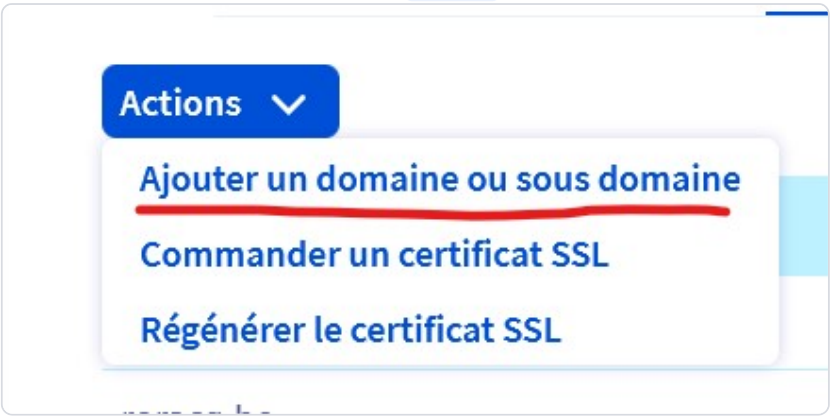
4.1. Ajouter un domaine ou sous-domaine

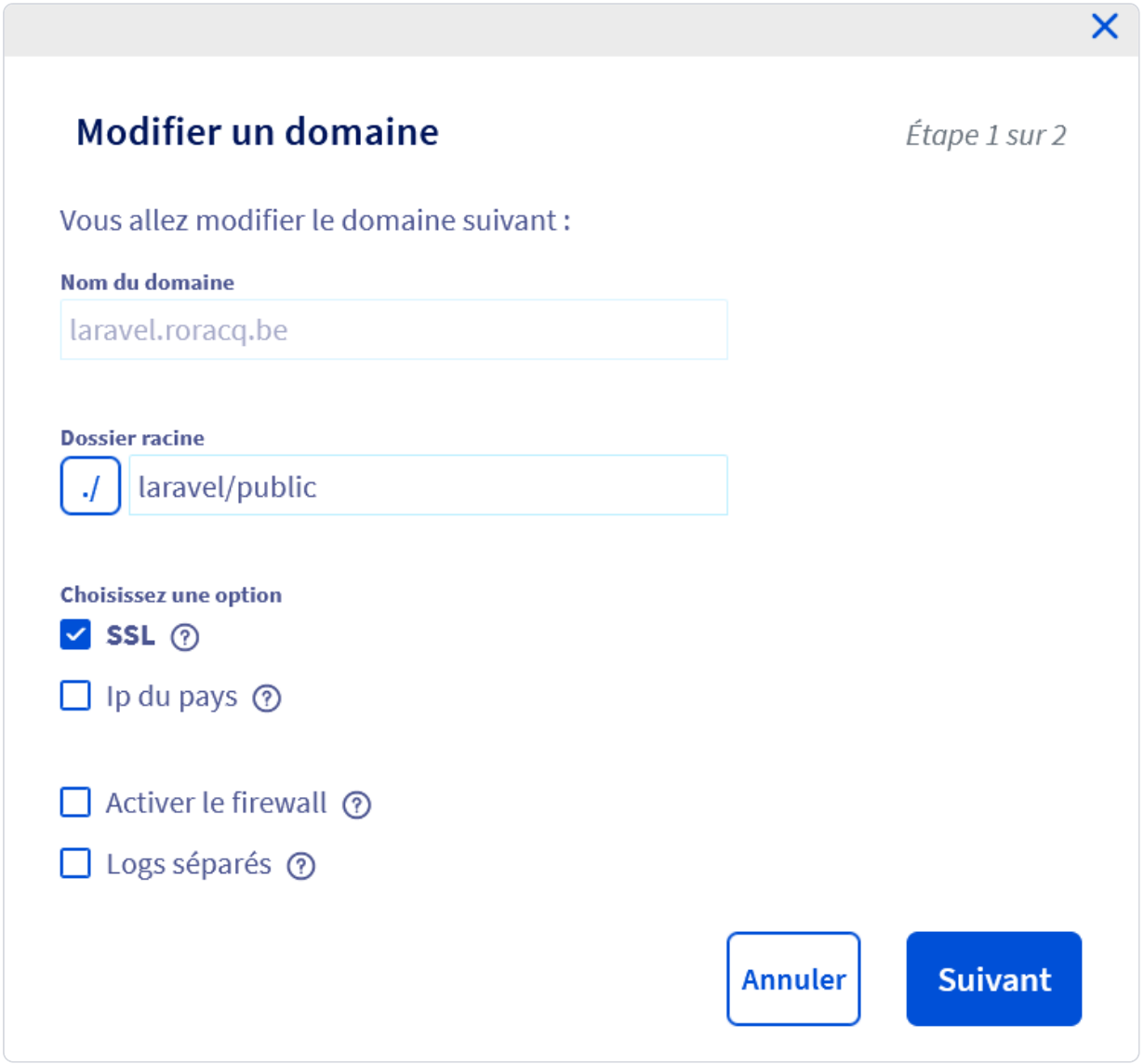
Pour que votre application Laravel soit accessible via Internet, vous devez configurer un domaine ou un sous-domaine. Voici comment procéder :

1. **Connectez-vous à l'interface OVH** : Utilisez vos identifiants pour accéder au tableau de bord OVH.
2. **Naviguez vers la section "Multisite"** : Ici, vous pouvez gérer tous les aspects liés à vos domaines.



3. **Ajouter un domaine ou sous-domaine** : Suivez les instructions pour ajouter un nouveau domaine ou configurer un sous-domaine existant. Assurez-vous que le domaine pointe vers le dossier racine de votre application Laravel, généralement le dossier `public`.





4.2. Modifier la configuration PHP

OVH permet de configurer la version de PHP et d'autres paramètres directement depuis son interface. Laravel 11 requiert PHP 8.2 ou supérieur.

Après avoir configuré votre domaine et la version de PHP, votre application devrait être presque prête à être utilisée. Il reste quelques étapes de configuration et de vérification avant de conclure le déploiement. Nous allons maintenant passer à la configuration du fichier `.env` pour assurer la connexion à la base de données.

Étape 5 : Configurer le fichier `.env`

Le fichier `.env` est crucial dans les applications Laravel car il contient toutes les configurations spécifiques à l'environnement dans lequel votre application s'exécute. Cette étape est essentielle pour s'assurer que votre application peut se connecter correctement à la base de données et fonctionner correctement en production.

5.1. Modifier les paramètres de connexion à la base de données

Pour que votre application Laravel puisse interagir avec la base de données sur OVH, vous devez configurer les paramètres de connexion dans le fichier `.env`. Voici les paramètres principaux à configurer :

- `DB_CONNECTION` : le type de votre base de données (par exemple, `mysql`).
- `DB_HOST` : l'adresse du serveur de base de données. Sur OVH, cela sera généralement une adresse locale comme `localhost` ou une adresse spécifique fournie par OVH.
- `DB_PORT` : le port de connexion à la base de données, généralement `3306` pour MySQL.
- `DB_DATABASE` : le nom de votre base de données sur OVH.
- `DB_USERNAME` : votre nom d'utilisateur pour la base de données.
- `DB_PASSWORD` : le mot de passe associé à l'utilisateur de la base de données.

Voici un exemple de configuration :

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_PORT=3306
DB_DATABASE=nom_de_votre_base
DB_USERNAME=utilisateur
DB_PASSWORD=mot_de_passe
```

Assurez-vous que ces informations sont correctes et correspondent à celles que vous avez configurées sur OVH. Une fois modifié, sauvegardez le fichier `.env`.

5.2. Vérifier les autres configurations

En plus des configurations de la base de données, vérifiez et ajustez les autres paramètres dans le fichier `.env` qui pourraient être nécessaires pour votre environnement de production, tels que :

- `APP_ENV` : assurez-vous que cela est défini sur `production`.
- `APP_KEY` : une clé unique qui sécurise les sessions et les données cryptées de votre application. Vous pouvez la générer avec la commande `php artisan key:generate`.
- `APP_DEBUG` : définissez-le à `false` pour désactiver le mode débogage en production.

Après avoir configuré correctement votre fichier `.env`, il est temps de passer à la régénération du certificat SSL pour sécuriser les connexions à votre site.

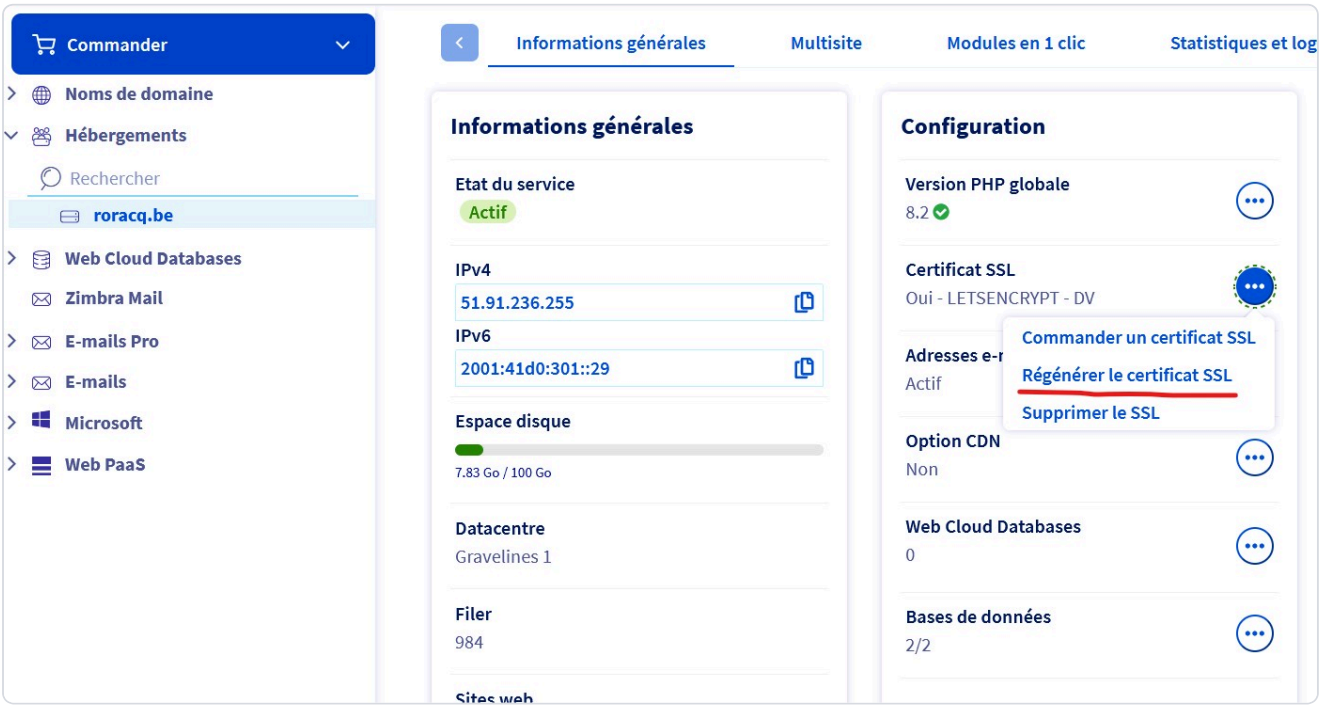
Étape 6 : Régénérer le certificat SSL

La sécurité est un aspect crucial de tout site web, et l'utilisation d'un certificat SSL (Secure Sockets Layer) est essentielle pour garantir que les données transmises entre le serveur et les utilisateurs sont chiffrées et sécurisées. OVH offre des options pour gérer facilement les certificats SSL de vos domaines.

6.1. Activer et régénérer le SSL

Pour activer ou régénérer un certificat SSL sur OVH, suivez ces étapes :

1. **Accéder à l'interface OVH** : Connectez-vous à votre compte sur le site d'OVH.
2. **Naviguez vers la section "Hébergements"** : Sélectionnez l'hébergement associé à votre domaine.
3. **Gérer le SSL** : Trouvez l'option pour gérer le SSL dans le tableau de bord de votre hébergement. OVH peut offrir un certificat SSL gratuit via Let's Encrypt.
4. **Activer/Régénérer le certificat** : Suivez les instructions pour activer ou régénérer le certificat SSL. Cela peut prendre quelques minutes à quelques heures pour que le certificat soit complètement installé et actif.



Assurez-vous que votre domaine est configuré pour utiliser HTTPS en modifiant les réglages DNS si nécessaire et en ajustant les configurations de votre application Laravel pour rediriger toutes les requêtes vers HTTPS.

Étape 7 : Tests de vérification

Après avoir configuré tous les aspects de votre application, il est important de réaliser une série de tests pour s'assurer que tout fonctionne comme prévu.

7.1. Vérifier les routes et pages du site

Testez toutes les routes principales de votre application pour vous assurer qu'elles chargent correctement sans erreurs. Cela inclut les pages d'accueil, les formulaires de connexion, les dashboards utilisateurs, et toute autre fonctionnalité clé de votre application.

7.2. Vérifier la connexion HTTPS

Accédez à votre site en utilisant `https://` pour confirmer que le SSL est actif et fonctionne correctement. Vérifiez que le navigateur affiche un cadenas vert ou une indication que la connexion est sécurisée.

7.3. Vérifier l'accès à la base de données

Testez les fonctionnalités de votre application qui nécessitent des interactions avec la base de données, comme les inscriptions, les connexions, et les mises à jour de données. Assurez-vous que toutes les opérations se déroulent sans erreurs et que les données sont correctement récupérées et stockées.

Ces tests sont essentiels pour s'assurer que votre application est prête pour le lancement et fonctionnera correctement pour vos utilisateurs finaux. Une fois ces tests passés avec succès, votre application est prête à être lancée en production sur OVH.

Étape 8 : Configurer le disque public pour utiliser le dossier public/storage

Pour que votre application Laravel gère correctement les fichiers statiques et les uploads sur l'environnement de production OVH, il est important de configurer correctement le disque public. Cette configuration permettra de stocker et de servir les fichiers directement depuis le dossier `public/storage` sans avoir besoin de créer des liens symboliques. Nous devons faire ça à cause d'une limitation de l'hébergement OVH qui n'autorise pas l'accès aux fichiers en utilisant des liens symboliques.

8.1. Supprimer le dossier public/storage

Avant de modifier la configuration, assurez-vous que le dossier `public/storage` qui pourrait avoir été créé automatiquement lors de l'upload des fichiers est supprimé. Cela évitera toute confusion ou conflit avec la configuration que vous allez établir. Vous pouvez supprimer ce dossier via un client FTP ou en utilisant une commande SSH si vous avez accès au terminal du serveur.

8.2. Modifier la configuration du disque public

Pour ajuster la configuration du disque public afin qu'il utilise le dossier `public/storage`, suivez ces étapes :

1. **Ouvrir le fichier de configuration** : Accédez au fichier `config/filesystems.php` dans votre projet Laravel.
2. **Modifier la configuration du disque public** : Localisez la section `disks` et modifiez la configuration du disque `public` pour qu'elle ressemble à ce qui suit :

```php
'disks' => [

    // ...

    'public' => [
        'driver' => 'local',
        'root' => public_path('storage'),
        'url' => env('APP_URL') . '/storage',
        'visibility' => 'public',
        'throw' => false,
    ],

    // ...
]
```

3. **Explications des modifications** :
   - `root` : Définit le chemin d'accès au dossier où les fichiers seront stockés, ici `public/storage`.

8.3. Sauvegarder et tester la configuration

Après avoir modifié la configuration, sauvegardez le fichier et testez pour vous assurer que les fichiers sont correctement stockés et accessibles. Vous pouvez le faire en uploadant un fichier via votre application et en vérifiant ensuite qu'il est accessible via l'URL spécifiée.

Cette configuration vous permet de gérer les fichiers statiques et les uploads de manière efficace et sécurisée, en utilisant le système de fichiers intégré de Laravel sans dépendre de liens symboliques, ce qui est particulièrement utile dans des environnements d'hébergement avec des restrictions spécifiques comme OVH.

3.22 Hébergement O2SWITCH

Étape 1 : Créer et importer la base de données

> ***Astuce SQLite***
> *Si tu es un **noob** (ou simplement malin) qui utilise encore **SQLite**, tu peux ignorer toutes ces histoires de dump : tu balances **directement** ton fichier `database.sqlite` sur le serveur, bim, c'est torché. Plus simple, plus rapide.*

1.1. Exporter ta base MySQL locale (optionnel si SQLite)

1. **Mets tes migrations à jour** localement :

```bash
php artisan migrate:fresh --seed
```

2. **Fais un dump** de ta base :

```bash
mysqldump -u username -p my_database > database_dump.sql
```

1.2. Importer sur o2switch via phpMyAdmin

1. Dans le **cPanel** d'o2switch, repère l'icône **phpMyAdmin** et connecte-toi.
2. Crée une nouvelle base (si ce n'est pas déjà fait).
3. **Importe** ton fichier `.sql`.

Étape 2 : Préparer le projet Laravel

1. **Sélectionner la version PHP** : Dans la partie "Logiciel" du cPanel, clique sur "Sélectionner une version de PHP". Choisis **PHP 8.2+** (ou 8.3).
2. **Installer tes dépendances** (en local) :

```bash
composer install --no-dev
```

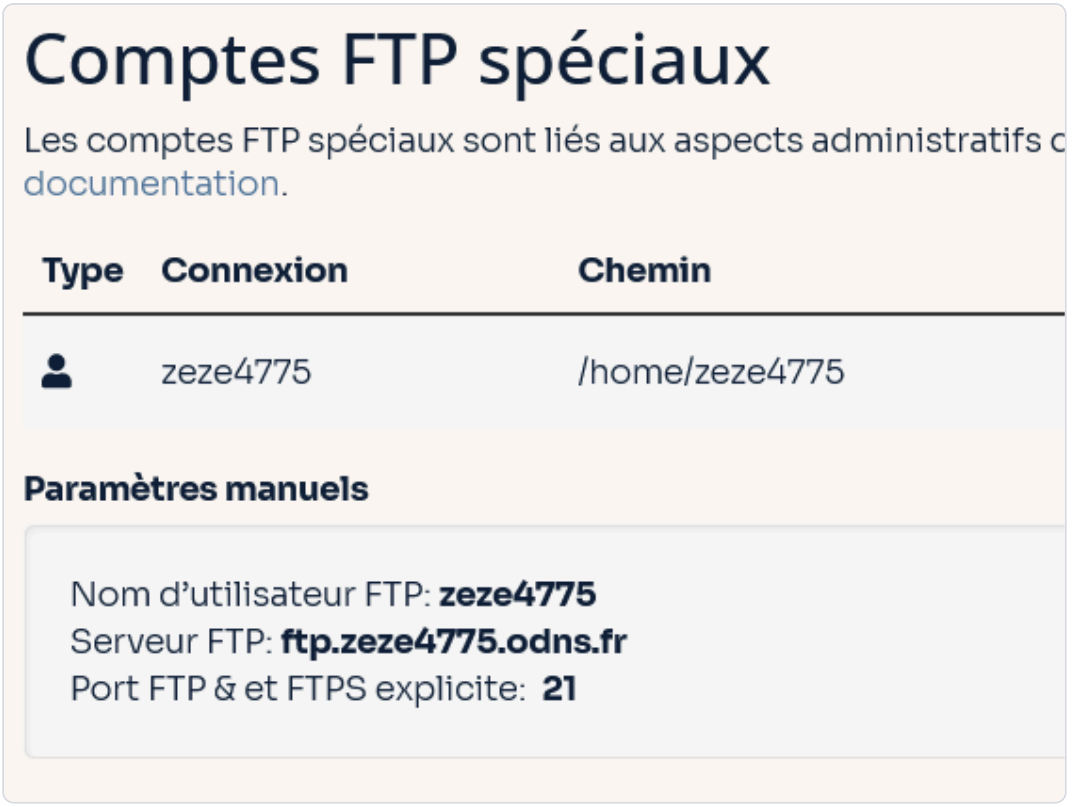
3. **Compiler tes assets** (avec npm/Vite) :

```bash
npm run build
```

4. **Supprimer** `node_modules` qui pèse une tonne.

Étape 3 : Uploader les fichiers

1. Dans cPanel > **Fichiers** > **Comptes FTP spéciaux**, retrouve tes identifiants.
2. Connecte-toi avec FileZilla (ou autre) en utilisant **le même mot de passe** que cPanel.
3. **Transfère** tous tes fichiers Laravel, **sauf** `node_modules`.
4. Dans `public/`, supprime le fichier `hot` si tu l'aperçois (sinon, meltdown de ton front).



Étape 4 : Créer un (sous-)domaine

1. Va dans cPanel > **Domaines** > **Sous-domaines**.
2. Tapes-y “test” (ou un nom plus stylé), sélectionne ton domaine principal, et assure-toi que le répertoire pointe vers `test/public`.
3. Clique sur “Créer”.

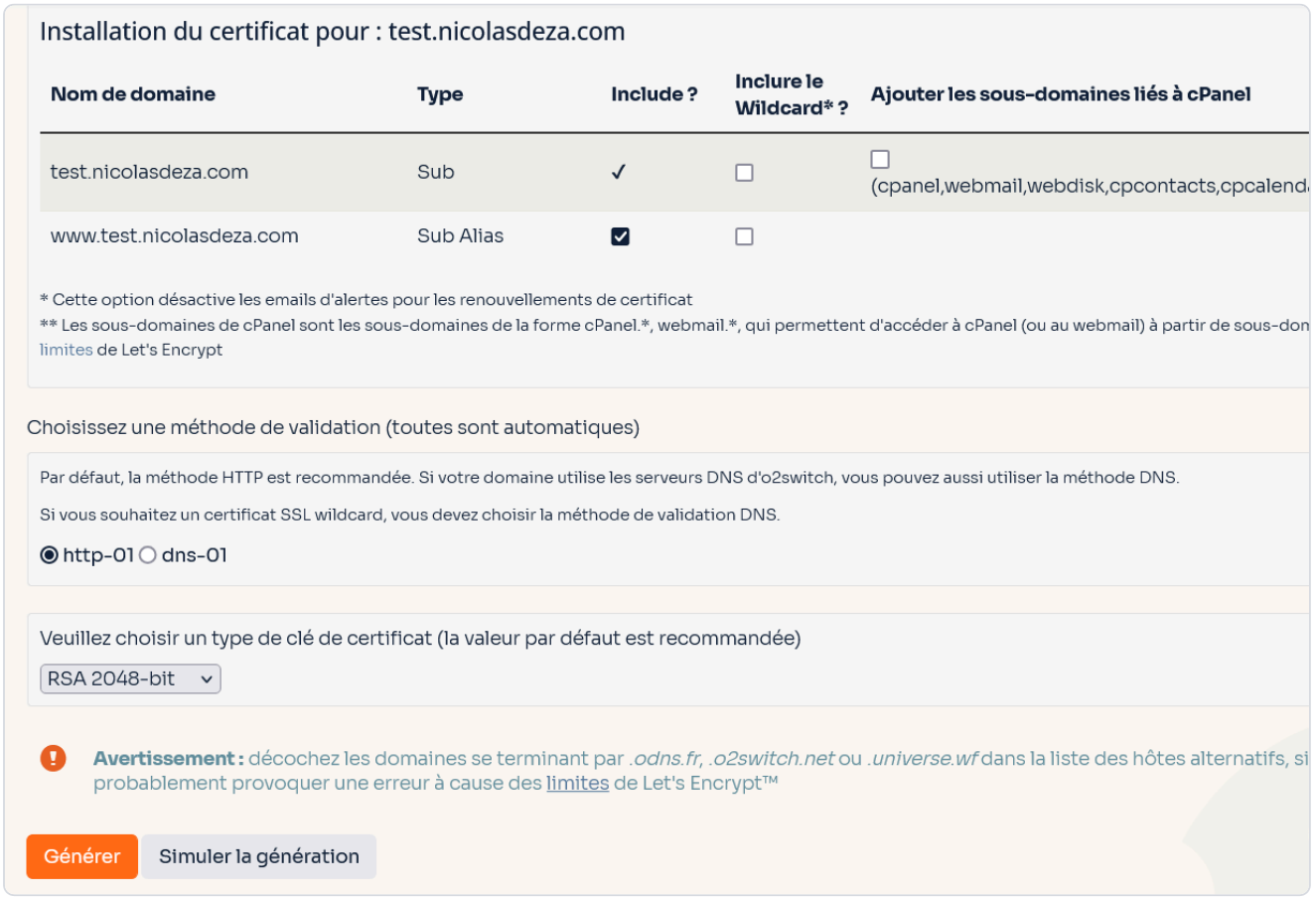
Étape 5 : Configurer ton `.env`

1. **DB_CONNECTION** : `mysql` (ou `sqlite` si t'as choisi la voie facile).
2. **DB_HOST** : souvent `localhost` sur o2switch.
3. **DB_PORT** : 3306 par défaut.
4. **DB_DATABASE**, **DB_USERNAME**, **DB_PASSWORD** : ceux créés dans le cPanel.
5. **APP_ENV=production**, **APP_DEBUG=false**
6. **APP_URL** : `https://test.mondomaine.com` ou ce que tu utilises.

Sauvegarde et mets le `.env` sur le serveur (ou modifie-le directement là-bas).

Étape 6 : Activer le HTTPS via Let's Encrypt

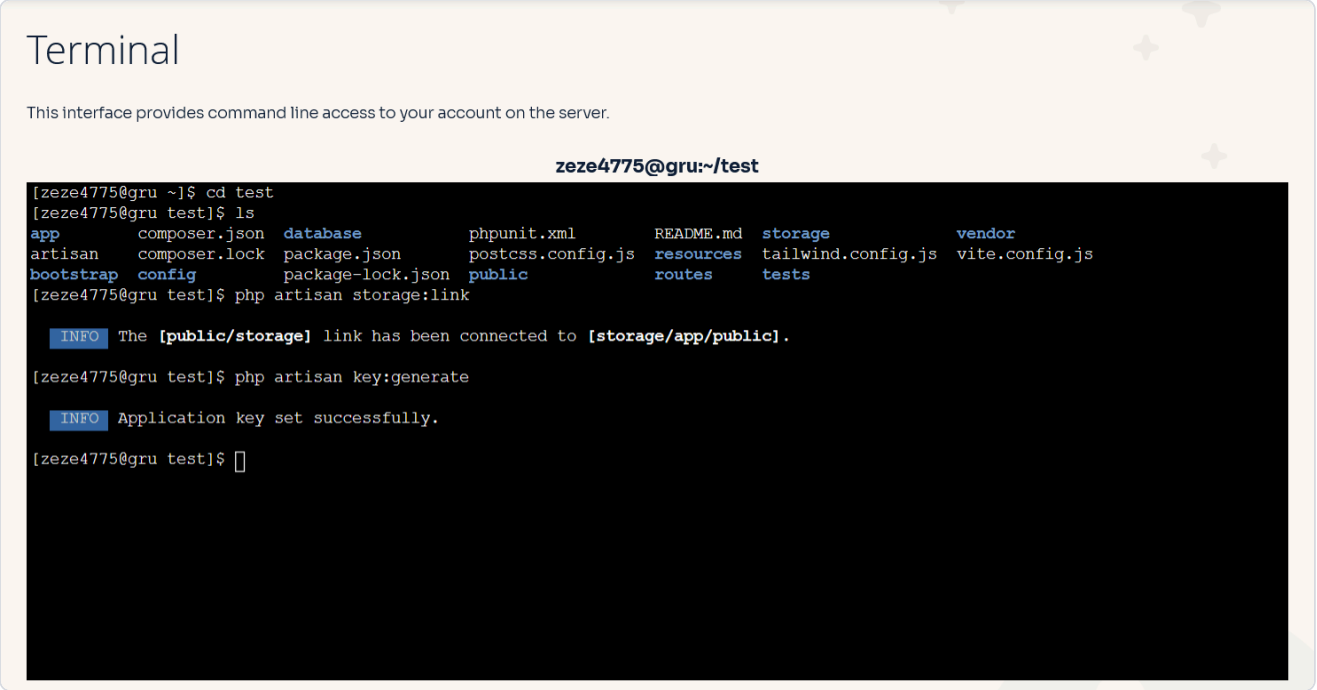
1. Dans cPanel, cherche le module **SSL Let's Encrypt**.
2. Sélectionne ton (sous-)domaine, clique sur "Générer" (méthode HTTP par défaut).
3. Patiente, puis vérifie que ça passe en vert sur `https://…`.



Étape 7 : Commandes via le terminal cPanel

o2switch te file un **Terminal** (menu "Terminal" dans le cPanel) pour lancer quelques trucs **depuis le serveur** :

```bash
# 1. Aller dans le dossier de ton projet
cd test

# 2. Générer la clé d'appli Laravel
php artisan key:generate

# 3. Créer le lien storage ⇒ public/storage
# (si tu veux gérer les uploads directement)
php artisan storage:link
```



Étape 8 : Vérifications finales

1. **Visite** ton site en HTTPS : `https://test.mondomaine.com`.
2. **Teste** toutes tes pages et routes.
3. **Regarde** s'il n'y a pas d'erreur 500 (check `storage/logs/laravel.log` si souci).

(▀̿_▀̿) Voilà, t'as (normalement) un beau site Laravel en ligne. L'objectif, c'est que ça marche… mais si tu te plantes, c'est pas la faute du tuto, hein. Bonne chance, petit Padawan de la prod ! (▀̿'̿-'̿)▀̿