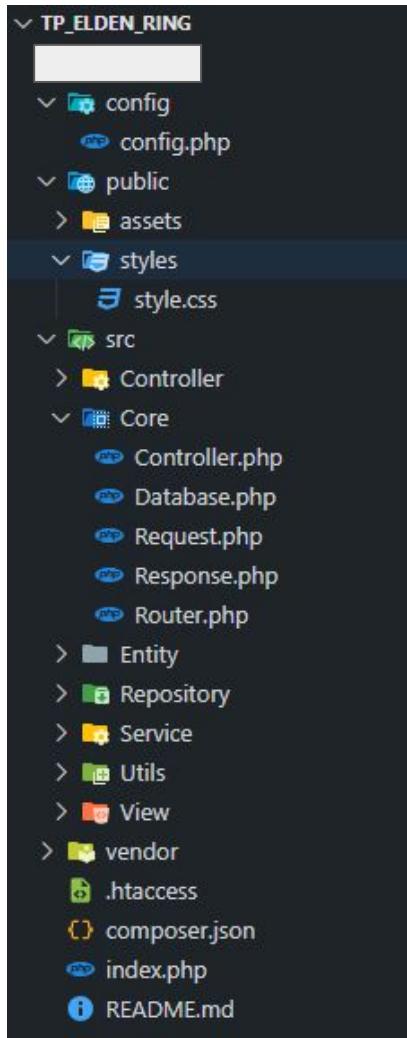


Programmation Orientée Objet (POO) et Modèle MVC en PHP

Valentin RIBEZZI



Nous allons créer un **MINI** Elden Ring
en PHP en utilisant la **Programmation**
Orientée Objet.



- config
 - config.php
- public
 - assets
 - styles
 - style.css
- src
 - Controller
 - Core
 - Controller.php
 - Database.php
 - Request.php
 - Response.php
 - Router.php
 - Entity
 - Repository
 - Service
 - Utils
 - View
- index.php
- README.md

Sommaire

- I. Programmation Orientée Objet (POO)
 - A. Introduction à la POO
 - B. Concepts fondamentaux de la POO
 - C. Encapsulation et visibilité
 - D. Héritage et polymorphisme
 - E. Abstraction et interfaces
 - F. Les namespaces
- II. Le modèle Modèle Vue Contrôleur (MVC)
 - A. Introduction au MVC
 - B. Routeur (ou Router)
 - C. Contrôleur (ou Controller)
 - D. Modèle (ou Repository)
 - E. Service
 - F. Vue (ou View)

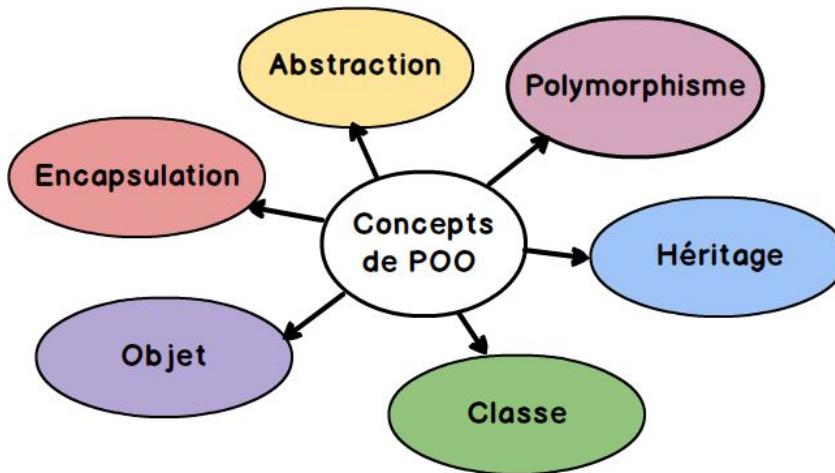


01

Programmation Orientée Objet (POO)

Introduction à la POO

Introduction à la POO



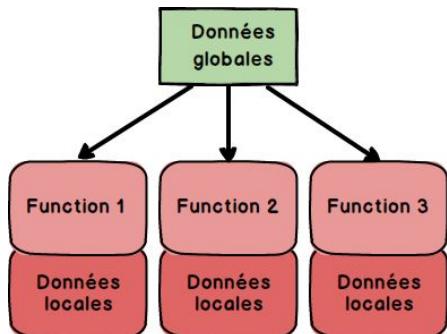
La **Programmation orientée objet (POO)** est une manière de résoudre un problème. C'est un paradigme, modèle de référence, une façon d'aborder un problème, une manière de penser et une concrétisation d'une philosophie de programmation.

Introduction à la POO

Jusqu'ici comment avons-nous l'habitude de programmer ? Et quelle est la structure de nos codes ?

Introduction à la POO

Procédure procédurale



La programmation procédurale est un paradigme de programmation organisé autour de procédures ou de sous-programmes. Le paradigme se concentre sur l'exécution séquentielle et la logique de programmation linéaire.

La programmation procédurale définit des procédures ou des fonctions pour des tâches spécifiques. Il utilise des variables, des boucles et des instructions conditionnelles pour contrôler le flux d'exécution.

Introduction à la POO

Les différences entre le procédurale et la POO

Aspect	Programmation Orientée Objet	Programmation procédurale
Organisation	Code structuré en objets regroupant données et comportements.	Code organisé en fonctions/procédures distinctes, avec une séparation nette entre fonctions et données.
Gestion des données	Données encapsulées dans des objets, accessibles via des méthodes (getters/setters).	Données souvent globales ou passées en paramètres entre fonctions.
Réutilisabilité	Réutilisation facilitée grâce à l'héritage et au polymorphisme ; possibilité d'extension sans modifier le code existant.	Réutilisation par appel de fonctions ; l'extension peut devenir complexe dans de gros projets.
Modélisation	Permet de modéliser des entités concrètes et leurs interactions (ex : utilisateurs, produits, personnages).	Moins intuitive pour représenter des entités complexes et leurs interactions.
Maintenance et évolutivité	Architecture modulaire facilitant la maintenance, les tests et l'évolution du code dans des projets de grande envergure.	Adaptée aux petits projets ou scripts simples, mais peut devenir difficile à maintenir sur le long terme.

Introduction à la POO

```
<?php
// Fonction qui calcule l'aire d'un rectangle
function rectangleArea($width, $height) {
    return $width * $height;
}

// Définition des dimensions du rectangle
$width = 5;
$height = 10;

// Calcul de l'aire
$area = rectangleArea($width, $height);

// Affichage du résultat
echo "L'aire du rectangle est de : $area";
?>
```

Introduction à la POO

```
<?php
// Déclaration de la classe Rectangle
class Rectangle {
    // Attributs privés pour encapsuler les données
    private $width;
    private $height;

    // Constructeur pour initialiser l'objet
    public function __construct($width, $height) {
        $this→width = $width;
        $this→height = $height;
    }

    // Méthode pour calculer l'aire du rectangle
    public function getArea() {
        return $this→width * $this→height;
    }
}

// Instanciation de l'objet Rectangle
$rectangle = new Rectangle(5, 10);

// Utilisation de la méthode de l'objet pour obtenir l'aire
echo "L'aire du rectangle est de : " . $rectangle→getArea();
?>
```

Introduction à la POO

Que faut-il retenir sur l'intérêt de la
Programmation **O**rientée **O**bjet ?

Utilisation de la notion d'objet (et d'instance)

Amélioration de la réutilisabilité et l'organisation du code

02

Programmation Orientée Objet (POO)

Concepts fondamentaux de la POO

Concepts fondamentaux de la POO

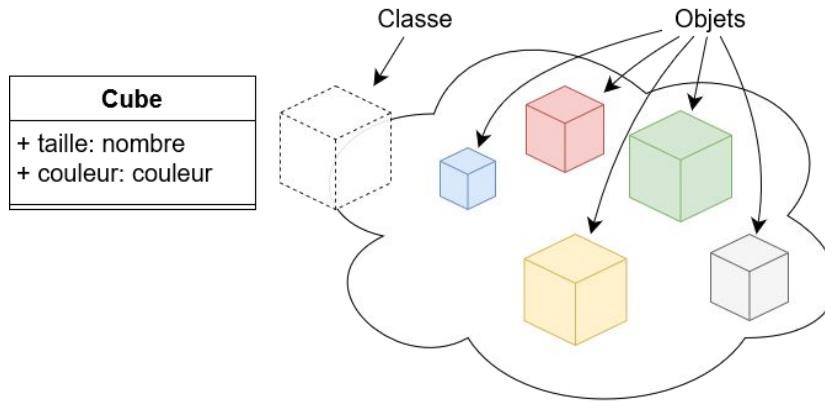
En POO, il existe **deux notions** indispensables pour comprendre son fonctionnement.



Concepts fondamentaux de la POO

Concrètement,
qu'est-ce qu'une **classe** ?

Concepts fondamentaux de la POO



Les classes sont des **moules**, des **patrons** qui permettent de créer des objets en série sur le même modèle. On peut se représenter une classe comme le schéma de construction ainsi que la liste des fonctionnalités d'un ensemble d'objets.

Concepts fondamentaux de la POO



```
● ● ●

class Character {
    // Attributs, constructeur et méthodes
}

$hero = new Character();
```

Pour cette classe, on créera le fichier
Character.php dans le dossier **src/Entity**

Concepts fondamentaux de la POO

Vu que l'on manipule des objets, des entités, chacun d'eux possède **des attributs spécifiques.**



A screenshot of a macOS terminal window. The window has a dark theme with red, yellow, and green title bar buttons. The code inside the terminal is as follows:

```
class Character {  
    protected $name; // Attribut 1  
    protected $health; // Attribut 2  
    protected $attackPower; // Attribut 3  
}  
  
$hero = new Character();
```

Concepts fondamentaux de la POO



```
class Character {  
    // Attributs, constructeur et méthodes  
}  
  
$hero = new Character();
```

Pour créer un ou des objets à partir d'un classe, on a besoin de le(s) construire.

Pour cela, nous allons créer une méthode à l'intérieur de notre classe pour effectuer cela, qui se nomme **Le constructeur**.

Concepts fondamentaux de la POO

```
● ● ●  
class Character {  
    // Attributs, constructeur et méthodes  
}  
  
$hero = new Character();
```



```
● ● ●  
class Character {  
protected $name; // Attribut 1  
protected $health; // Attribut 2  
protected $attackPower; // Attribut 3  
  
public function __construct(string $name, int $health, int $attackPower) {  
    $this->name = $name;  
    $this->health = $health;  
    $this->attackPower = $attackPower;  
}  
  
$hero = new Character("Chevalier Errant", 150, 25);
```

Notre méthode pour le constructeur (qui doit s'appeler comme cela) va prendre en paramètre les valeurs à définir pour les attributs de la classe.

Concepts fondamentaux de la POO

Comment faire pour que notre objet effectue des actions spécifiques ?

```
class Character {  
    protected $name; // Attribut 1  
    protected $health; // Attribut 2  
    protected protected $attackPower; // Attribut 3  
  
    public function __construct(string $name, int $health, int $attackPower) {  
        $this->name = $name;  
        $this->health = $health;  
        $this->attackPower = $attackPower;  
    }  
}  
  
$hero = new Character("Chevalier Errant", 150, 25);
```

Concepts fondamentaux de la POO

En POO, pour effectuer un ensemble d'actions ou d'instructions, nous avons **des méthodes**.

Si fonctions et méthodes peuvent prendre des entrées et renvoyer des sorties, les fonctions ne sont pas relatives à un objet. En fait, en programmation orientée objet pure, les fonctions n'existent pas puisque tout est objet, c'est-à-dire instance de classe.

Grâce à une méthode, on va pouvoir réaliser des **opérations** qui sont **spécifiques à un objet** : modifier ses attributs, les afficher, les retourner (ou les initialiser dans le cas de la méthode `__init__`), etc.

Concepts fondamentaux de la POO

Le premier genre de méthode que nous allons voir pour nos classes sont **les getters** et **les setters**.



```
// Class definition
class Character {
    protected $name; // Attribut 1
    protected $health; // Attribut 2
    protected protected $attackPower; // Attribut 3

    // Constructeur
    public function __construct(string $name, int $health, int $attackPower) {
        $this->name = $name;
        $this->health = $health;
        $this->attackPower = $attackPower;
    }

    // Getters
    public function getName(): string {
        return $this->name;
    }

    public function getHealth(): int {
        return $this->health;
    }

    public function getAttackPower(): int {
        return $this->attackPower;
    }

    // Setters
    public function setHealth(int $health): void {
        $this->health = $health;
    }
}

// On crée nos objets
$character1 = new Character("Chevalier Errant", 150, 25);
$character2 = new Character("Seigneur des Ténèbres", 200, 30);
```

Il joue un rôle très important car ce sont eux qui vont permettre de modifier (proprement) les attributs de notre classe, mais également de les récupérer.

*On reparlera de leur importance dans l'**encapsulation**.*

Concepts fondamentaux de la POO

Nous allons maintenant créer une “vrai” méthode, qui va permettre à un personnage de lancer des attaques à d’autres.

Pour cela, nous allons créer la méthode **attack**.

PS : On a aussi ajouté la méthode **displayStatus**, que l’on peut utiliser pour voir les points de vie et la puissance d’attaque de notre personnage.

```
class Character {
    protected $name; // Attribut 1
    protected $health; // Attribut 2
    protected $attackPower; // Attribut 3

    // Constructeur
    public function __construct(string $name, int $health, int $attackPower) {
        $this->name = $name;
        $this->health = $health;
        $this->attackPower = $attackPower;
    }

    // Getters
    public function getName(): string {
        return $this->name;
    }

    public function getHealth(): int {
        return $this->health;
    }
    public function getAttackPower(): int {
        return $this->attackPower;
    }

    // Setters
    public function setHealth(int $health): void {
        $this->health = $health;
    }

    // Méthodes
    public function attack(Character $target): void {
        $target->setHealth($target->getHealth() - $this->attackPower);
    }

    public function displayStatus(): void {
        echo "{$this->name} - Points de vie: {$this->health}, Attaque: {$this->attackPower}\n";
    }
}

// On crée nos objets
$character1 = new Character("Chevalier Errant", 150, 25);
$character2 = new Character("Seigneur des Ténèbres", 200, 30);
```

Concepts fondamentaux de la POO

En POO, il est possible de définir des objets comme des types pour des variables.

En l'occurrence ici, notre cible sera de type **Character** puisque c'est le type d'objet que l'on veut cibler.

```
// Méthodes
public function attack(Character $target): void {
    $target→setHealth($target→getHealth() - $this→attackPower);
}
```

1. Nous allons passer en paramètre de notre méthode une cible, qui sera un objet **Character**
2. Nous allons définir une nouvelle santé pour notre cible : Cette santé sera égale à sa santé actuelle moins la puissance d'attaque du personnage qui attaque la cible.

Concepts fondamentaux de la POO

```
● ● ●

class Character {
    protected $name; // Attribut 1
    protected $health; // Attribut 2
    protected $attackPower; // Attribut 3

    // Constructeur
    public function __construct(string $name, int $health, int $attackPower) {
        $this->name = $name;
        $this->health = $health;
        $this->attackPower = $attackPower;
    }

    // Getters
    public function getName(): string {
        return $this->name;
    }

    public function getHealth(): int {
        return $this->health;
    }
    public function getAttackPower(): int {
        return $this->attackPower;
    }

    // Setters
    public function setHealth(int $health): void {
        $this->health = $health;
    }

    // Méthodes
    public function attack(Character $target): void {
        $target->setHealth($target->getHealth() - $this->attackPower);
    }

    public function displayStatus(): void {
        echo "{$this->name} - Points de vie: {$this->health}, Attaque: {$this->attackPower}\n";
    }
}

// On crée nos objets
$character1 = new Character("Chevalier Errant", 150, 25);
$character2 = new Character("Seigneur des Ténèbres", 200, 30);

// On appelle la méthode "attack" de notre classe
$character1->attack($character2);
```

03

Programmation Orientée Objet (POO)

Encapsulation et visibilité

Encapsulation et visibilité

On a vu que nos attributs, getters, setters et nos méthodes pouvaient être en **protected** ou en **public**.

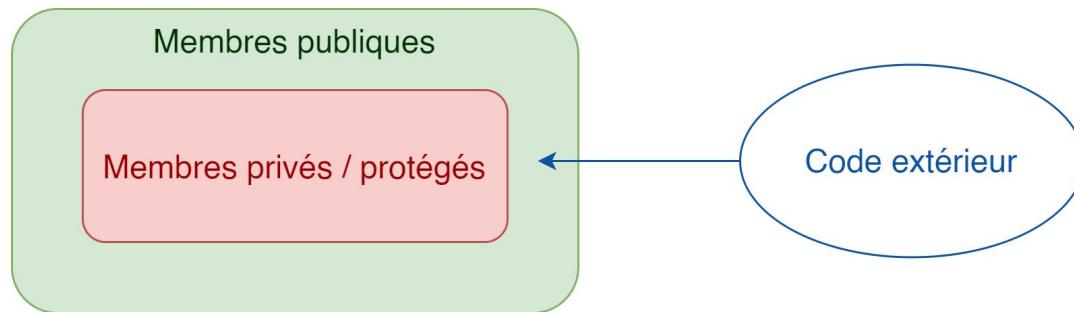
Mais qu'est-ce que cela signifie ?

Encapsulation et visibilité

En POO, il existe le principe d'**encapsulation**.

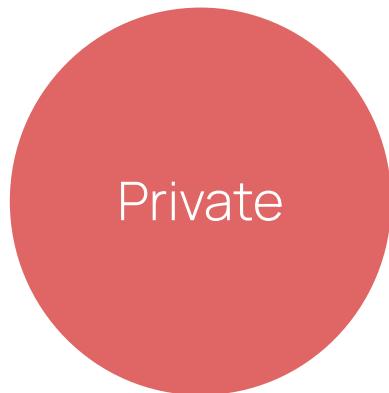
Le but est de cacher les détails de mise en oeuvre
d'un objet.

Autrement dit, on veut réguler l'accès direct aux
champs d'un objet depuis l'extérieur de celui-ci.



Encapsulation et visibilité

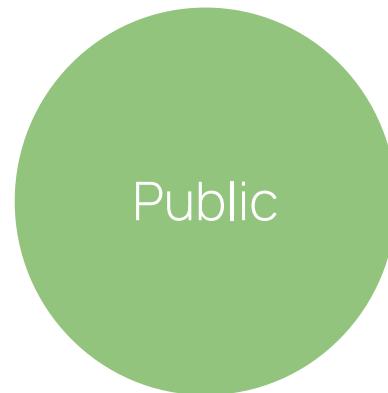
Il y a **3 niveaux** de visibilité pour une entité en POO.



Invisible en dehors
de la classe

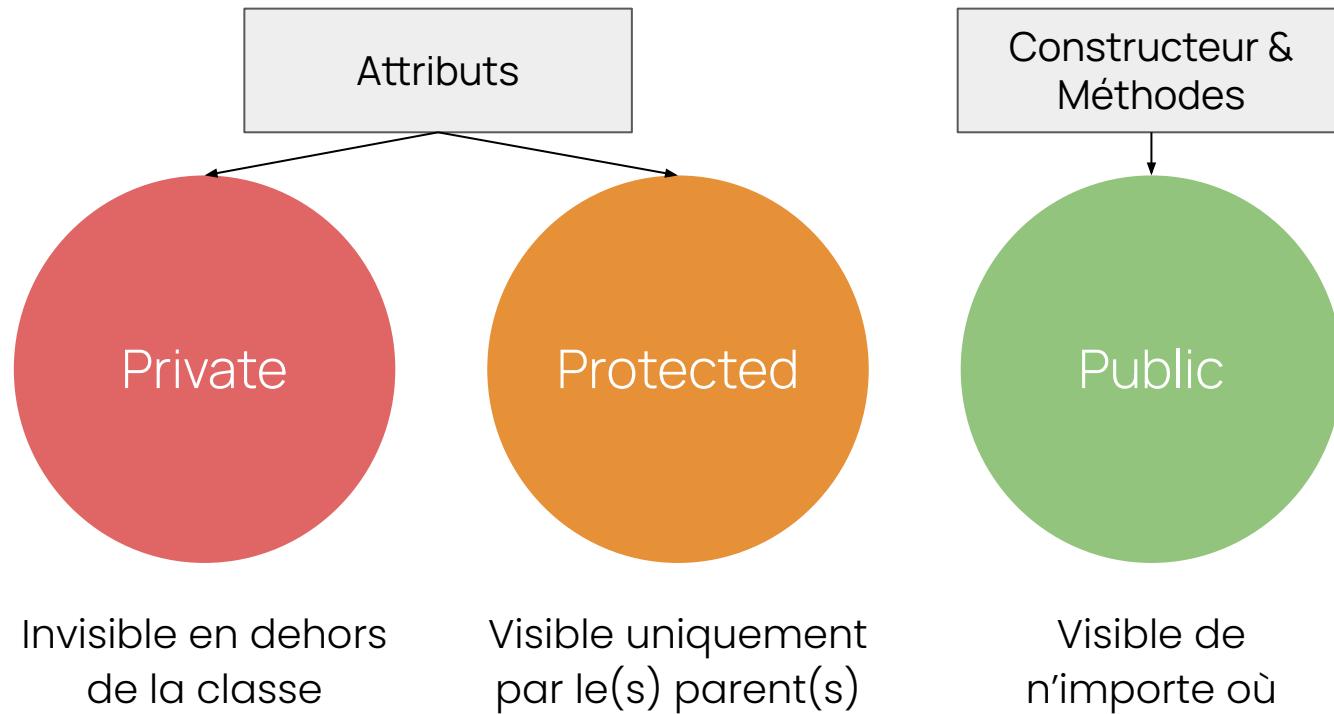


Visible uniquement
par le(s) parent(s)



Visible de
n'importe où

Encapsulation et visibilité



Encapsulation et visibilité

```
class Character {  
    // Attributs, méthodes, ....  
}
```

```
class Character {  
    protected $name; // Attribut 1  
    protected $health; // Attribut 2  
    protected $attackPower; // Attribut 3  
}
```

```
class Character {  
    protected $name; // Attribut 1  
    protected $health; // Attribut 2  
    protected $attackPower; // Attribut 3  
  
    // Constructeur  
    public function __construct(string $name, int $health, int $attackPower) {  
        $this->name = $name;  
        $this->health = $health;  
        $this->attackPower = $attackPower;  
    }  
}
```

```
class Character {  
    // Getters  
    public function getName(): string {  
        return $this->name;  
    }  
  
    public function getHealth(): int {  
        return $this->health;  
    }  
    public function getAttackPower(): int {  
        return $this->attackPower;  
    }  
  
    // Setters  
    public function setHealth(int $health): void {  
        $this->health = $health;  
    }  
  
    // Méthodes  
    public function attack(Character $target): void {  
        $target->setHealth($target->getHealth() - $this->attackPower);  
    }  
  
    public function displayStatus(): void {  
        echo "{$this->name} - Points de vie: {$this->health}, Attaque: {$this->attackPower}\n";  
    }  
}
```

```
// On crée nos objets  
$character1 = new Character("Chevalier Errant", 150, 25);  
$character2 = new Character("Seigneur des Ténèbres", 200, 30);  
  
// On appelle la méthode "attack" de notre classe  
$character1->attack($character2);
```

04

Programmation Orientée Objet (POO)

Héritage et polymorphisme

Héritage et polymorphisme

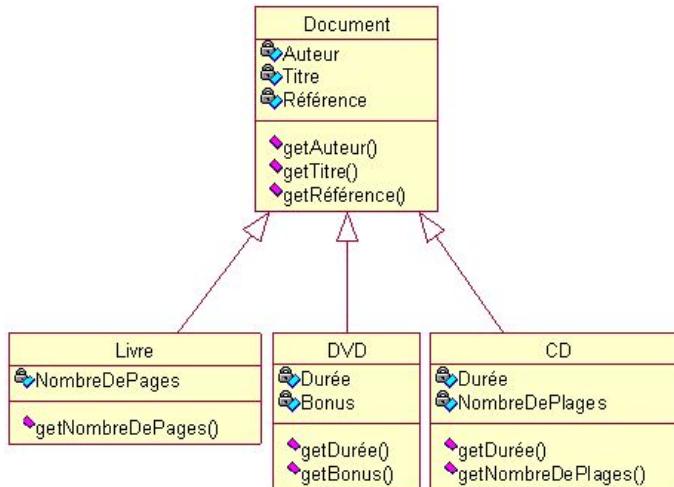
Jusqu'à maintenant, nous n'avons crée qu'une classe générique permettant de créer des personnages.

Nous allons à présent créer des "sous-classes" à partir de notre classe générique.

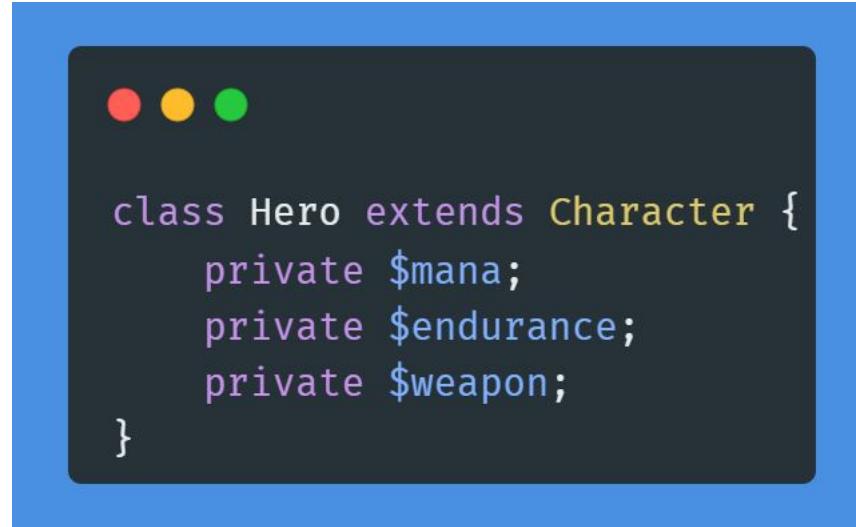
Cela se nomme de l'**héritage**.

Héritage et polymorphisme

L'héritage en programmation orientée objet permet de créer facilement des classes similaires à partir d'une autre classe. On parle alors de faire hériter une classe fille d'une classe mère.



Héritage et polymorphisme



```
class Hero extends Character {  
    private $mana;  
    private $endurance;  
    private $weapon;  
}
```

Pour cette classe, on créera le fichier
Hero.php dans le dossier **src/Entity**

Héritage et polymorphisme

```
● ● ●

class Hero extends Character {
    private $mana;
    private $endurance;
    private $weapon;

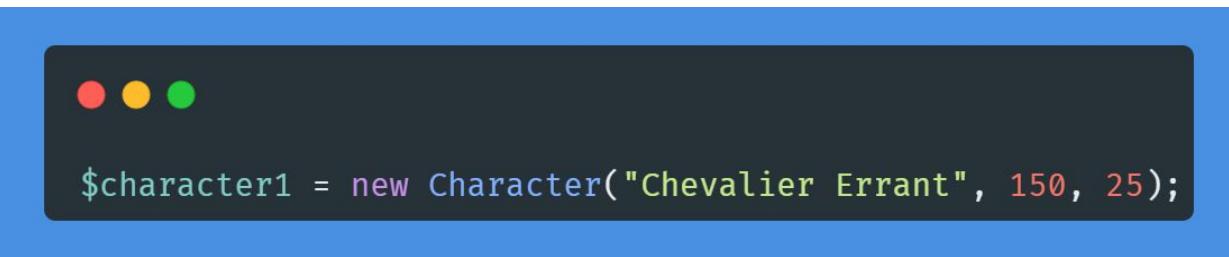
    public function __construct(string $name, int $health, int $attackPower, int $mana = 100, int $endurance =
        100) {
        parent::__construct($name, $health, $attackPower);
        $this->mana = $mana;
        $this->endurance = $endurance;
    }
}
```

Notre constructeur va combiner le constructeur de notre **parent**, ainsi que la définition des valeurs des attributs de la classe **Hero**.

Pour le constructeur du parent, on va écrire **parent::__construct** (Comme si nous voulions créer un objet **Character**)

Héritage et polymorphisme

Lorsque l'on va créer un objet à l'aide de notre classe **Hero**, nous définirons à la fois les attributs de la classe **Hero**, mais aussi les attributs de la classe **Character**.



A screenshot of a code editor window. At the top left, there are three colored circles: red, yellow, and green. Below them, the code `$character1 = new Character("Chevalier Errant", 150, 25);` is displayed in a monospaced font.



A screenshot of a code editor window, identical in layout to the one above it. It shows the same three colored circles at the top left and the same code: `$hero = new Hero("Chevalier Errant", 150, 25, 100, 50); // 100 mana, 50 endurance`. This illustrates that the constructor for the derived class (`Character`) is called, which in turn calls the constructor for the base class (`Hero`).

Héritage et polymorphisme

L'utilisation de **max** pour les fonctions **decreaseMana** et **decreaseEndurance** permet de s'assurer que la valeur d'endurance ne descend jamais en dessous de 0.

```
●●●

class Hero extends Character {
    private $mana;
    private $endurance;
    private $weapon; // Arme équipée, instance de Weapon (optionnelle)

    public function __construct(string $name, int $health, int $attackPower, int $mana = 100, int $endurance =
100) {
        parent::__construct($name, $health, $attackPower);
        $this->mana = $mana;
        $this->endurance = $endurance;
    }

    // Getters
    public function getMana(): int {
        return $this->mana;
    }

    public function getEndurance(): int {
        return $this->endurance;
    }

    // Méthodes de modification
    public function increaseMana(int $amount): void {
        $this->mana += $amount;
    }

    public function decreaseMana(int $amount): void {
        $this->mana = max(0, $this->mana - $amount);
    }

    public function increaseEndurance(int $amount): void {
        $this->endurance += $amount;
    }

    public function decreaseEndurance(int $amount): void {
        $this->endurance = max(0, $this->endurance - $amount);
    }

    // Méthode pour équiper une arme
    public function equipWeapon(Weapon $weapon): void {
        $this->weapon = $weapon;
        echo "{$this->name} a équipé {$weapon->getName()} qui ajoute {$weapon->getDamageBonus()} points
d'attaque.\n";
    }

    // La gestion des sorts sera réalisée dans les classes de sort (voir FireSpell)
}
```

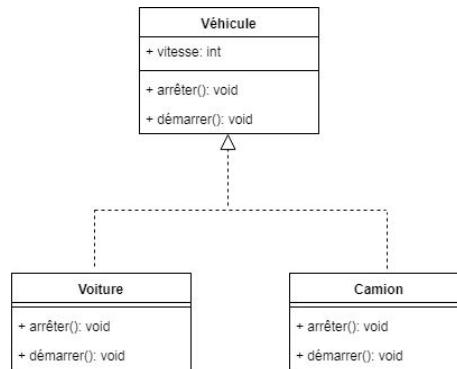
Héritage et polymorphisme

Une question peut alors se poser :

Que se passe t-il si deux Character
n'attaquent pas de la même manière ?

Héritage et polymorphisme

C'est là qu'intervient le **polymorphisme**.



Le polymorphisme est un concept clé en programmation orientée objet (POO) qui permet aux objets de différentes classes d'utiliser des méthodes portant le même nom mais de se comporter de manière différente selon la classe à laquelle ils appartiennent.

Héritage et polymorphisme

Pour un **Character**, la méthode *attack* était ainsi :

```
● ● ●

public function attack(Character $target): void {
    echo "{$this->name} attaque {$target->getName()} et inflige {$this->attackPower} points de dégâts.\n";
    $target->setHealth($target->getHealth() - $this->attackPower);
}
```

Héritage et polymorphisme

Dans **Hero**, nous redéfinissons la méthode comme ceci :

```
public function attack(Character $target): void {
    $attackCost = 10; // Coût en endurance pour une attaque

    if ($this->endurance < $attackCost) {
        echo "{$this->name} n'a pas assez d'endurance pour attaquer.\n";
        return;
    }

    $this->decreaseEndurance($attackCost);

    $totalDamage = $this->attackPower;
    if ($this->weapon) {
        $totalDamage += $this->weapon->getDamageBonus();
    }

    echo "{$this->name} attaque {$target->getName()} et inflige {$totalDamage} points de dégâts (coût
endurance: {$attackCost}).\n";
    $target->setHealth($target->getHealth() - $totalDamage);
}
```

Héritage et polymorphisme

```
class Hero extends Character {
    private $name;
    private $endurance;
    private $weapon; // Arme équipée, instance de Weapon (optionnelle)

    public function __construct(string $name, int $health, int $attackPower, int $mana = 100, int $endurance =
100) {
        parent::__construct($name, $health, $attackPower);
        $this->mana = $mana;
        $this->endurance = $endurance;
    }

    // Getters
    public function getMana(): int {
        return $this->mana;
    }

    public function getEndurance(): int {
        return $this->endurance;
    }

    // Méthodes de modification
    public function increaseMana(int $amount): void {
        $this->mana += $amount;
    }

    public function decreaseMana(int $amount): void {
        $this->mana = max(0, $this->mana - $amount);
    }

    public function increaseEndurance(int $amount): void {
        $this->endurance += $amount;
    }

    public function decreaseEndurance(int $amount): void {
        $this->endurance = max(0, $this->endurance - $amount);
    }

    // Méthode pour équiper une arme
    public function equipWeapon(Weapon $weapon): void {
        $this->weapon = $weapon;
        echo "{$this->name} a équipé {$weapon->getName()} qui ajoute {$weapon->getDamageBonus()} points
d'attaque.\n";
    }

    // Redéfinition de attack() : vérifie l'endurance
    public function attack(Character $target): void {
        $attackCost = 10; // Coût en endurance pour une attaque

        if ($this->endurance < $attackCost) {
            echo "{$this->name} n'a pas assez d'endurance pour attaquer.\n";
            return;
        }

        $this->decreaseEndurance($attackCost);

        $totalDamage = $this->attackPower;
        if ($this->weapon) {
            $totalDamage += $this->weapon->getDamageBonus();
        }

        echo "{$this->name} attaque {$target->getName()} et inflige {$totalDamage} points de dégâts (coût
endurance: {$attackCost}).\n";
        $target->setHealth($target->getHealth() - $totalDamage);
    }
}
```

Héritage et polymorphisme

Nous allons aussi créer la classe **Boss** qui hérite de **Character**.

```
class Boss extends Character {
    // Attaque spéciale : double les dégâts
    public function attack(Character $target): void {
        $damage = $this->attackPower * 2;
        echo "Le Boss {$this->name} attaque {$target->getName()} et inflige {$damage} points de dégâts (attaque
spéciale) !\n";
        $target->setHealth($target->getHealth() - $damage);
    }
}
```

Pour cette classe, on créera le fichier
Boss.php dans le dossier **src/Entity**

Héritage et polymorphisme

Au passage, nous allons créer la classe **Weapon** pour attribuer une arme à la classe **Hero**.

```
class Weapon {  
    private $name;  
    private $damageBonus;  
  
    public function __construct(string $name, int $damageBonus) {  
        $this->name = $name;  
        $this->damageBonus = $damageBonus;  
    }  
  
    public function getName(): string {  
        return $this->name;  
    }  
  
    public function getDamageBonus(): int {  
        return $this->damageBonus;  
    }  
}
```

*Pour cette classe, on créera le fichier
Weapon.php dans le dossier **src/Entity***

Héritage et polymorphisme

Maintenant, nous allons utiliser l'**abstraction** pour gérer les sorts, et les **interfaces** pour définir le comportement des consommables.

05

Programmation Orientée Objet (POO)

Abstraction et interfaces

Abstraction et interfaces

Abstraction

Abstract signifie “non complète”.

L'abstraction consiste à masquer à l'utilisateur les détails inutiles.

Autrement dit, il est possible de construire des classes où des méthodes ne possèdent pas de code.

MAIS, une classe ou méthode abstraite doit être obligatoirement redéfinie. **Une classe est abstraite si elle contient au moins une méthode abstraite.**

Abstraction et interfaces

Dans Elden Ring, tous les sorts partagent certaines caractéristiques communes (nom, coût en mana, dégâts), mais leur manière de lancer le sort peut varier.

Pour cette classe, on créera le fichier **Spell.php** dans le dossier **src/Entity**

```
abstract class Spell {
    protected $name;
    protected $manaCost;
    protected $damage;

    public function __construct(string $name, int $manaCost, int $damage) {
        $this->name = $name;
        $this->manaCost = $manaCost;
        $this->damage = $damage;
    }

    // Méthode abstraite à implémenter par chaque sort concret
    abstract public function cast(Hero $caster, Character $target): void;

    // Getter pour le nom du sort
    public function getName(): string {
        return $this->name;
    }
}
```

Du coup, on crée une méthode **cast**, qui devra être réimplémenter par les enfants de cette classe (devenu *abstraite* à cause de la méthode). On ne définit que sa "signature".

Abstraction et interfaces

La particularité d'une classe abstraite, c'est qu'elle ne peut être instanciée !

On ne peut pas faire :

```
● ● ●  
$normalSpell = new Spell("Eclair", 15, 30);
```

Cela donnera l'erreur suivante :

(!) Fatal error: Uncaught Error: Cannot instantiate abstract class App\Entity\Spell in C:\wamp64\www\COURS\TP_ELDEN_RING\src\Service\CombatService.php on line 69

Abstraction et interfaces

Voici une classe qui hérite de notre classe abstraite **Spell**

```
class FireSpell extends Spell {
    public function __construct(string $name, int $manaCost, int $damage) {
        parent::__construct($name, $manaCost, $damage);
    }

    public function cast(Hero $caster, Character $target): string {
        if ($caster->getMana() < $this->manaCost) {
            return 'Pas assez de mana pour lancer le sort.';
        }
        // Déduire le mana du héros
        $caster->decreaseMana($this->manaCost);
        $target->setHealth($target->getHealth() - $this->damage);

        return $caster->getName() . ' lance un sort de feu et inflige ' . $this->damage . ' points de dégâts à
        ' . $target->getName() . '.';
    }
}
```

Cette classe doit OBLIGATOIUREMENT implémenter la méthode **cast**, vu qu'elle hérite de la classe **Spell**, qui est *abstraite*.

*Pour cette classe, on créera le fichier **FireSpell.php** dans le dossier **src/Entity***

Abstraction et interfaces

Et du coup, c'est cette classe que l'on utilisera pour créer des objets **FireSpell**.



```
$fireSpell = new FireSpell("Boule de Feu", 20, 40);
```

Abstraction et interfaces

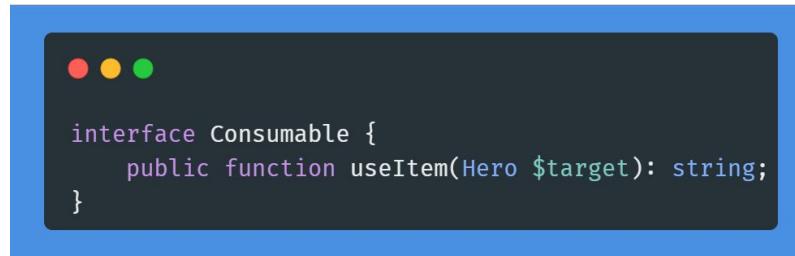
Interfaces

Une interface permet de regrouper des comportements (méthodes) et non pas des attributs. Elle peut contenir des constantes.

C'est très similaire à une classe abstraite, mais cela permet de ne pas imposer un comportement commun ou encore de ne pas avoir besoin de partager des attributs.

Abstraction et interfaces

Dans Elden Ring, les fioles d'Estus et les fioles de mana n'ont rien en commun.



A screenshot of a code editor window with a dark theme. At the top left are three small colored dots (red, yellow, green). The main area contains the following PHP code:

```
interface Consumable {
    public function useItem(Hero $target): string;
}
```

On crée une interface **Consumable** qui va définir une méthode **useItem** à OBLIGATOIREMENT implémenter (comme une méthode abstraite) si on utilise l'interface **Consumable**.

Comme une classe abstraite, **on ne peut pas instancier une interface**. En plus, ce n'est pas une classe (donc pas un moule), donc pour le coup, vous aurez juste une belle erreur !

Pour cette interface, on créera le fichier **Consumable.php** dans le dossier **src/Entity**

Abstraction et interfaces

Pour cette classe, on créera les fichiers **EstusFlask.php** et **ManaFlask.php** dans le dossier **src/Entity**

```
class EstusFlask implements Consumable {
    private $healAmount;

    public function __construct(int $healAmount = 50) {
        $this->healAmount = $healAmount;
    }

    public function useItem(Hero $target): string {
        $target->setHealth($target->getHealth() + $this->healAmount);
        return 'Le héros a utilisé une Estus Flask et a récupéré ' . $this->healAmount . ' points de vie.';
    }
}
```

```
class ManaFlask implements Consumable {
    private $manaAmount;

    public function __construct(int $manaAmount = 30) {
        $this->manaAmount = $manaAmount;
    }

    public function useItem(Hero $target): string {
        $target->increaseMana($this->manaAmount);
        return 'Le héros a utilisé une Mana Flask et a récupéré ' . $this->manaAmount . ' points de mana.';
    }
}
```

06

Programmation Orientée Objet (POO)

Statiques et finales

Statiques et finales

En POO, pour améliorer la sûreté de notre code, on peut définir une classe comme "finale".

Comme son nom l'indique, cela signifie qu'elle ne pourra plus être dérivée par une sous-classe. Cela implique également que ses attributs et méthodes ne pourront plus être redéfinis.

Concrètement, comment cela se met en place dans le code ?

Prenons l'exemple de notre classe **FireSpell**.

Statiques et finales

En ajoutant le mot “**final**” juste avant le mot clé **class**, notre classe ne pourra jamais être dérivée.

```
final class FireSpell extends Spell {
    public function __construct(string $name, int $manaCost, int $damage) {
        parent::__construct($name, $manaCost, $damage);
    }

    public function cast(Hero $caster, Character $target): string {
        if ($caster->getMana() < $this->manaCost) {
            return 'Pas assez de mana pour lancer le sort.';
        }
        // Déduire le mana du héros
        $caster->decreaseMana($this->manaCost);
        $target->setHealth($target->getHealth() - $this->damage);

        return $caster->getName() . ' lance un sort de feu et inflige ' . $this->damage . ' points de dégâts à
        ' . $target->getName() . '.';
    }
}
```

Autre exemple : Si **Spell** était une classe “**final**”, **FireSpell** ne pourrait pas être son dérivée.

Statiques et finales

Cela fonctionne aussi avec des méthodes !

Statiques et finales

Dans la classe **Character**, nous avions défini une méthode de base.

```
public function attack(Character $target): void {
    echo "{$this->name} attaque {$target->getName()} et inflige {$this->attackPower} points de dégâts.\n";
    $target->setHealth($target->getHealth() - $this->attackPower);
}
```

Statiques et finales

Dans **Hero**, nous avions redéfini la méthode comme ceci :

```
● ● ●

public function attack(Character $target): void {
    $attackCost = 10; // Coût en endurance pour une attaque

    if ($this->endurance < $attackCost) {
        echo "{$this->name} n'a pas assez d'endurance pour attaquer.\n";
        return;
    }

    $this->decreaseEndurance($attackCost);

    $totalDamage = $this->attackPower;
    if ($this->weapon) {
        $totalDamage += $this->weapon->getDamageBonus();
    }

    echo "{$this->name} attaque {$target->getName()} et inflige {$totalDamage} points de dégâts (coût
endurance: {$attackCost}).\n";
    $target->setHealth($target->getHealth() - $totalDamage);
}
```

C'était le **Polymorphisme**.

Statiques et finales

Maintenant, si nous décidions de passer **attack** en méthode “final”, il ne serait plus possible de faire du polymorphisme et de redéfinir la méthode.

```
final public function attack(Character $target): string {
    echo "{$this→name} attaque {$target→getName()} et inflige {$this→attackPower} points de dégâts. \n";
    $target→setHealth($target→getHealth() - $this→attackPower);
}
```

La méthode serait “figée” par rapport à sa classe actuelle.

Statiques et finales

Pour finir avec la **Programmation Orientée Objet**, il existe des propriétés et des méthodes qui peuvent être utilisées sans que la classe soit instancié.

On appelle cela des propriétés ou méthodes **statiques**.

Statiques et finales

```
● ● ●

class CombatCalculator {
    public static function calculateDamage(int $baseDamage, int $weaponBonus = 0, float $criticalChance = 0.2):
array {
    $totalDamage = $baseDamage + $weaponBonus;
    $isCritical = false;
    // Tirage aléatoire pour le coup critique
    if (mt_rand(0, 100) / 100 < $criticalChance) {
        $totalDamage *= 2;
        $isCritical = true;
    }
    return [
        'damage'  => $totalDamage,
        'critical' => $isCritical
    ];
}
```

Ici, nous allons créer une classe **CombatCalculator** qui va contenir une méthode permettant de calculer les dégâts (et de mettre un coup critique).

*Pour cette classe, on créera le fichier **CombatCalculator.php** dans le dossier **src/Utils***

07

Programmation Orientée Objet (POO)

Les namespaces

Les namespaces

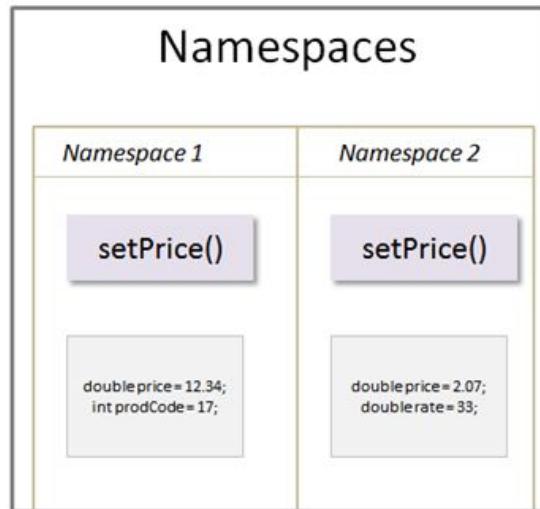
En POO, il existe un moyen de séparer ses éléments au sein du code de telle sorte à éviter les conflits (ou collisions). Ces collisions sont dues à des duplications de noms (ou identifiants) d'éléments comme les fonctions, les constantes ou les classes.

On appelle ça les **namespaces**.

Les namespaces

Les namespaces font alors en sorte de créer comme des répertoires abstraits qui permettent d'encapsuler les éléments et prévenir ainsi tout risque de collision.

Toutefois, ils ont aussi un autre rôle qui consiste à créer des alias (des noms alternatifs) plus simples pour les éléments qui ont des noms trop longs ou trop compliqués.



Les namespaces

Quand est ce qu'aura-t-on vraiment besoin des namespaces?



Dans un gros projet

où le nombre de fonctions,
de constantes et de classes
mises en jeu est énorme,
donc le risque de collision
est élevé.



Dans des projets qui
utilisent des librairies
externes

dans ce cas, il se peut que
l'on utilise des éléments qui
peuvent avoir le même nom
que leur homologues qui
sont embarquées dans la
librairie.

Les namespaces



```
<?php
namespace App\Entity;

class Character {
    // ...
}
```

Ici, nous avons dit à notre classe **Character** qu'elle fait partie du namespace **App/Entity**.

Les namespaces



Si on veut utiliser notre classe **Character** autre part, nous n'aurons pas besoin de l'importer, mais juste de spécifier son namespace.

Les namespaces



```
<?php
namespace App\Entity;

class Character {
    // ...
}
```

A screenshot of a terminal window with a dark background and light-colored text. It shows a snippet of PHP code starting with a shebang (<?php), followed by a namespace declaration 'namespace App\Entity;', and a class definition 'class Character { ... }'. The code is syntax-highlighted, with 'App\Entity' in purple and the class name in white.

Comment trouvons-nous le mot clé App
(*dans App\Entity*) ?

Les namespaces

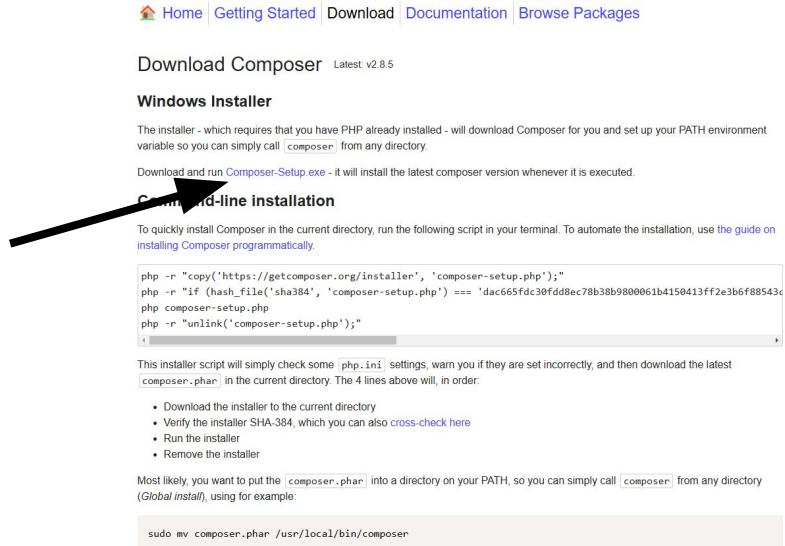
Pour cela, nous allons utiliser le **protocole PSR-4** avec le système **Composer**.



Les namespaces

Étape 1 : Installer Composer sur le PC

<https://getcomposer.org/download/>



The screenshot shows the official Composer download page. A large black arrow points from the text "Command-line installation" to the code block below it.

[Home](#) | [Getting Started](#) | [Download](#) | [Documentation](#) | [Browse Packages](#)

Download Composer Latest: v2.8.5

Windows Installer

The installer - which requires that you have PHP already installed - will download Composer for you and set up your PATH environment variable so you can simply call `composer` from any directory.

Download and run [Composer-Setup.exe](#) - it will install the latest composer version whenever it is executed.

Command-line installation

To quickly install Composer in the current directory, run the following script in your terminal. To automate the installation, use the guide on [installing Composer programmatically](#).

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') === 'dac665fdc30fd8ec78b38b9800061b4150413ff2e3b6f88543c
php -r 'unlink('composer-setup.php')';"
```

This installer script will simply check some `.php.ini` settings, warn you if they are set incorrectly, and then download the latest `composer.phar` in the current directory. The 4 lines above will, in order:

- Download the installer to the current directory
- Verify the installer SHA-384, which you can also [cross-check here](#)
- Run the installer
- Remove the installer

Most likely, you want to put the `composer.phar` into a directory on your PATH, so you can simply call `composer` from any directory (*Global install*), using for example:

```
sudo mv composer.phar /usr/local/bin/composer
```

Les namespaces

Étape 2 : Créer un fichier composer.json dans le projet.



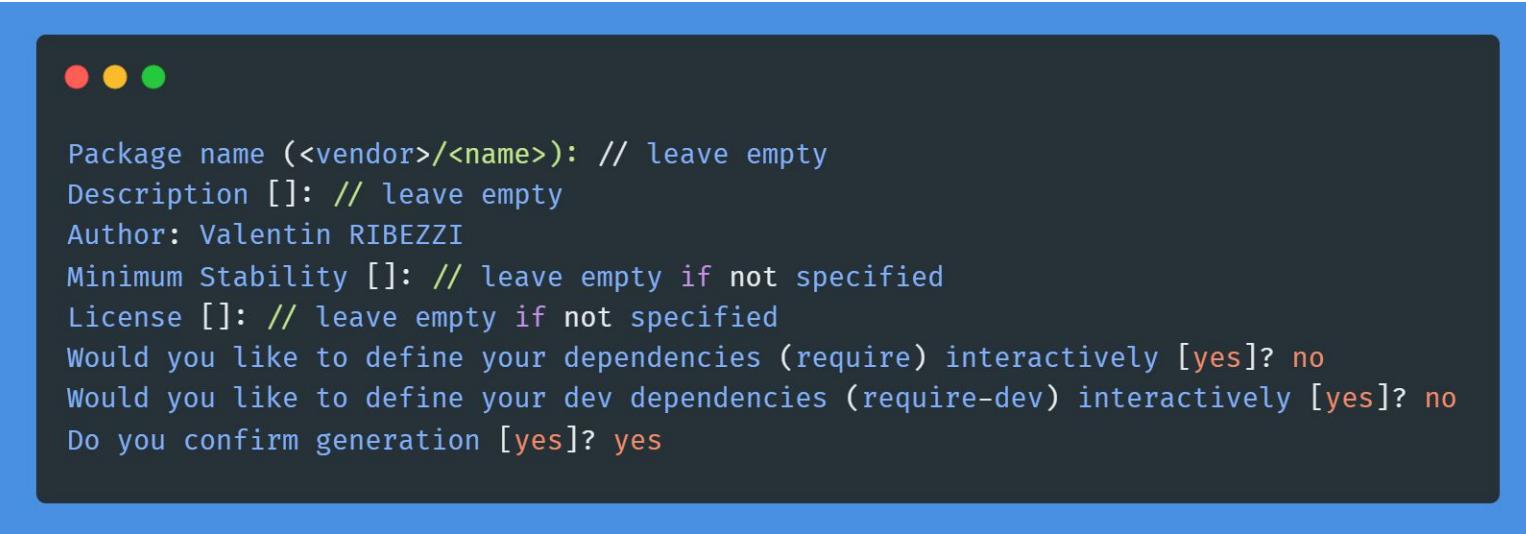
```
{  
    "name": "valen/tp_elden_ring",  
    "authors": [  
        {  
            "name": "Valentin RIBEZZI",  
            "email": "valentin.ribezzi@pyramide-agency.com"  
        }  
    ],  
    "require": {}  
}
```

Vous pouvez soit le créer à la main, soit utiliser la commande `composer init`

On stockera le fichier **composer.json** à la **racine de notre projet** (au même niveau qu'**index.php**)

Les namespaces

Étape 2 BIS : Si on crée le fichier composer.json avec
composer init



```
Package name (<vendor>/<name>): // leave empty
Description []: // leave empty
Author: Valentin RIBEZZI
Minimum Stability []: // leave empty if not specified
License []: // leave empty if not specified
Would you like to define your dependencies (require) interactively [yes]? no
Would you like to define your dev dependencies (require-dev) interactively [yes]? no
Do you confirm generation [yes]? yes
```

Le paramétrage sera le suivant lorsque l'on exécute la commande.

Les namespaces

Étape 2 BIS : Si on crée le fichier composer.json avec
composer init

Une fois le composer.json créé, il faut rajouter le paramétrage pour le protocole psr-4.
Il est possible de le générer avec la commande `composer dump-autoload`.

The diagram illustrates the generation of a full `composer.json` file from a partial configuration. On the left, a dark blue rounded rectangle represents a terminal window containing a partial `composer.json` snippet:

```
"autoload": {  
    "psr-4": {  
        "App\\": "app/"  
    }  
}
```

An arrow points from this partial configuration to a larger dark blue rounded rectangle on the right, representing the full generated `composer.json` file:

```
{  
    "name": "valen/tp_elden_ring",  
    "autoload": {  
        "psr-4": {  
            "App\\": "app/"  
        }  
    },  
    "authors": [  
        {  
            "name": "Valentin RIBEZZI",  
            "email": "valentin.ribezzi@pyramide-agency.com"  
        }  
    ],  
    "require": {}  
}
```

Les namespaces

Étape 3 : Mettre à jour le protocole pour prendre en compte notre arborescence

Dans notre projet, nous voulons que App corresponde à notre dossier src.

Pour cela, nous allons légèrement modifier (à la main) le psr-4.

The diagram illustrates a file comparison between a local configuration and a remote repository. On the left, a dark blue box represents a local file containing the following code:

```
"autoload": {  
    "psr-4": {  
        "App\\": "src/"  
    }  
}
```

An arrow points from this local file to a larger dark blue box representing a remote repository. This repository contains the following code:

```
{  
    "name": "valen/tp_elden_ring",  
    "autoload": {  
        "psr-4": {  
            "App\\": "src/"  
        }  
    },  
    "authors": [  
        {  
            "name": "Valentin RIBEZZI",  
            "email": "valentin.ribezzi@pyramide-agency.com"  
        }  
    ],  
    "require": {}  
}
```

Les namespaces

Étape 4 : Mettre à jour l'autoloader avec la configuration que nous avons créé

Pour cela, il faut juste executer la commande :



À la racine de notre projet, cela va nous créer un dossier **Vendor** avec un fichier **autoload.php**. C'est ce fichier qui va nous permettre d'utiliser les namespaces avec App.

Les namespaces

Étape 5 : Ajouter notre autoloader dans notre page index.php

```
<?php  
require_once __DIR__ . '/vendor/autoload.php';
```

Le **__DIR__** contient le chemin dans lequel est notre projet, nous avons juste à lui concaténer le chemin menant vers notre fichier **autoload.php** dans le dossier **Vendor**.

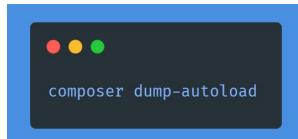
Les namespaces

Si nous voulons créer une nouvelle "base" de namespace, il suffit de rajouter le répertoire du projet concerné, et lui donner son nom de namespace.



```
"autoload": {  
    "psr-4": {  
        "App\\": "src/",  
        "Tests\\": "tests"  
    }  
}
```

On relancera ensuite la commande composer dump-autoload pour importer tous cela dans le fichier autoload.php



```
composer dump-autoload
```

Les namespaces

```
<?php  
namespace App\Entity;  
  
class Character {  
    // ....  
}
```

Grâce à cette configuration, nous pouvons maintenant créer et utiliser **des namespaces** pour les différents niveaux de notre architecture !

Les namespaces

Nous avons mis en place nos différentes entités en intégrant la **P**rogrammation **O**rientée **O**bjet.

Nous allons maintenant utiliser notre architecture avec de construire une interface web pour afficher un combat.

01

Le modèle Modèle Vue Contrôleur (MVC)

Introduction au MVC

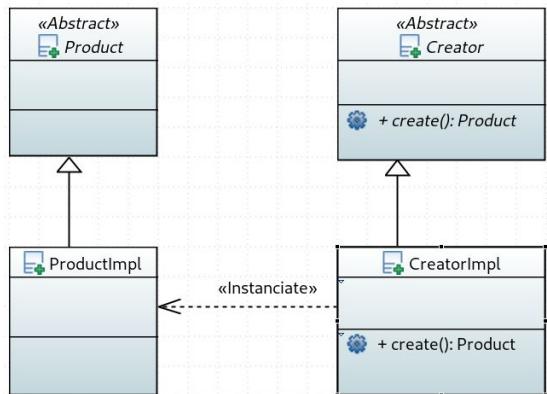
Introduction au MVC

Avant tout, il faut savoir que l'architecture MVC est ce qu'on appelle un "**Design pattern**".

Mais qu'est-ce qu'un **Design pattern** ?

Introduction au MVC

Un **design pattern**, ou **patron de conception**, en français, est une solution type à un problème spécifique en programmation. Il représente un modèle de solution à implémenter.



C'est un peu comme un patron de couture : vous avez une série d'étapes à suivre pour créer votre pantalon ou votre robe. Si vous suivez mal les étapes, c'est la catastrophe et votre pantalon est raté. Si vous les suivez bien, votre pantalon est réussi.

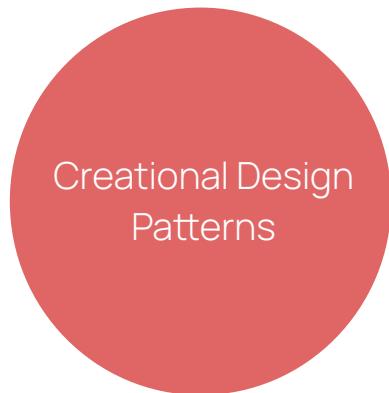
Introduction au MVC

En une phrase :

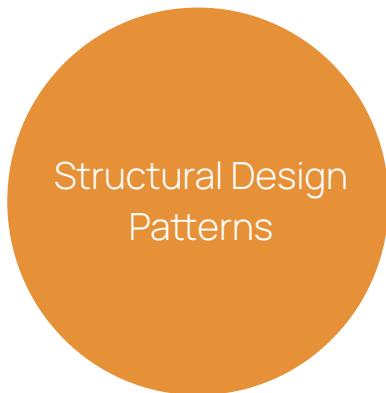
Les **design patterns** fournissent une proposition de solution face à un problème donné.

Introduction au MVC

Il existe **trois** grands types de **design patterns** :



qui représentent tous les design patterns dédiés à la création d'objets



qui permettent de gérer et d'assembler des objets dans des structures plus grandes



qui correspondent à la communication entre les objets

Introduction au MVC

Du coup oui, le **modèle MVC** est un **Design Pattern**.

Il définit la segmentation de notre application en plusieurs “modules”.

Modèle

Vue

Contrôleur

Introduction au MVC

Modèle

Cette partie gère ce qu'on appelle la logique métier de votre site. Elle comprend notamment la gestion des données qui sont stockées, mais aussi tout le code qui prend des décisions autour de ces données.

Vue

Cette partie se concentre sur l'affichage. Elle ne fait presque aucun calcul et se contente de récupérer des variables pour savoir ce qu'elle doit afficher.

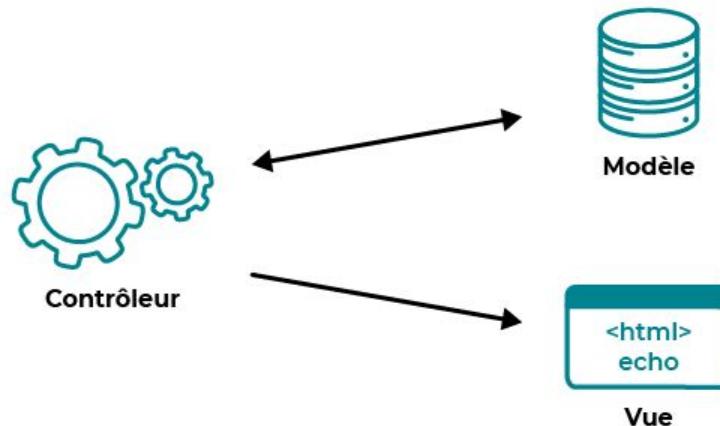
Contrôleur

Cette partie gère les échanges avec l'utilisateur. C'est en quelque sorte l'intermédiaire entre l'utilisateur, le modèle et la vue. Le contrôleur va recevoir des requêtes de l'utilisateur.

Voici comment cela se retranstcrit en termes de communication entre nos modules.

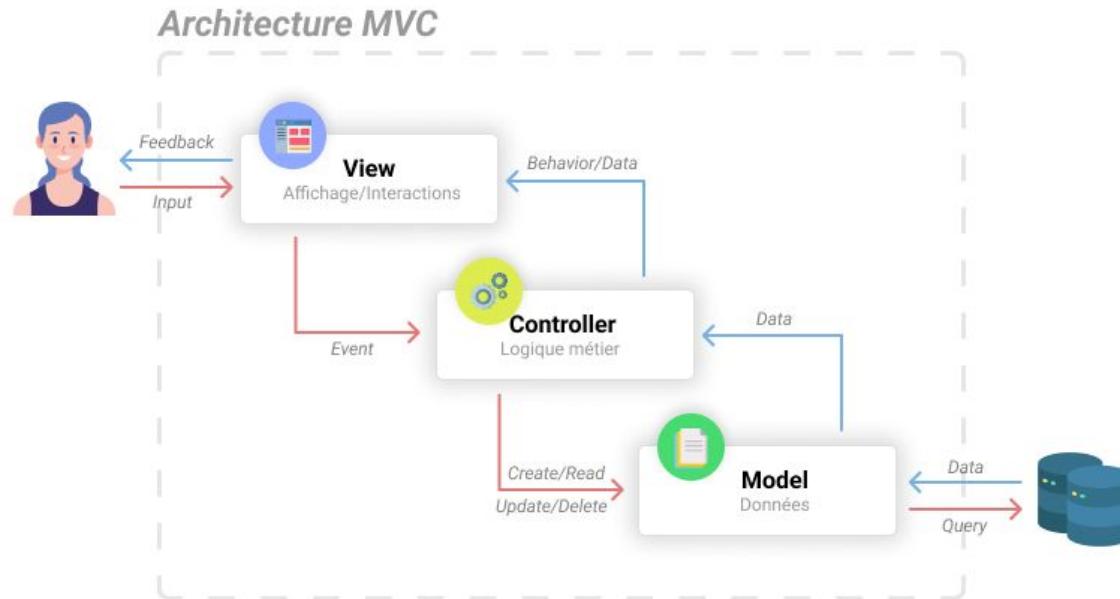
Introduction au MVC

Le contrôleur sera le chef d'orchestre de notre architecture, c'est lui qui **reçoit la requête du visiteur** et qui **contacte d'autres fichiers** (le modèle et la vue) pour leur demander des services.



Introduction au MVC

Si on ajoute une entrée client au sein de notre architecture, cela donnera un schéma comme ceci :



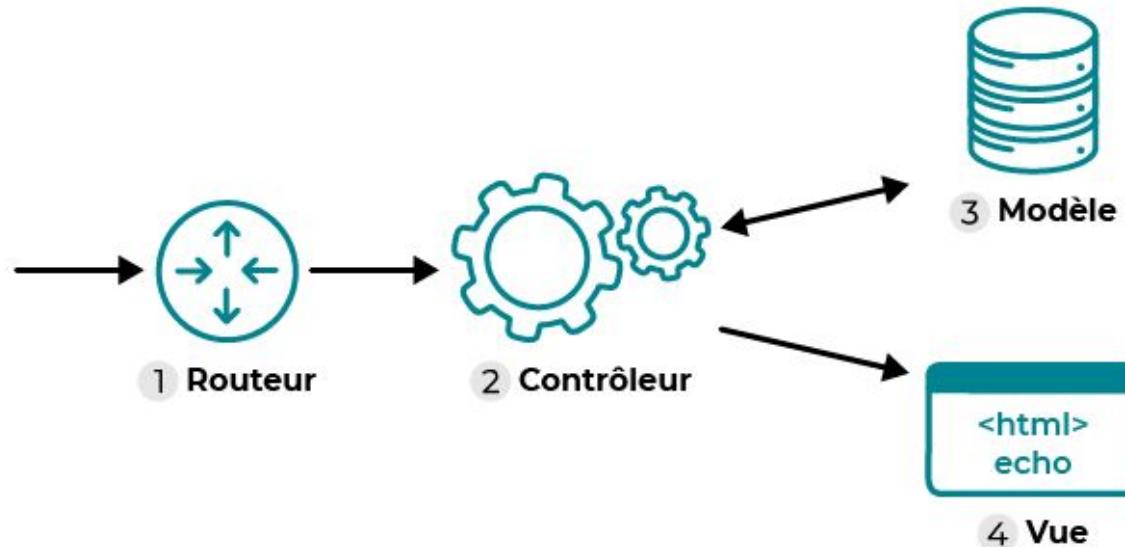
02

Le modèle Modèle Vue Contrôleur (MVC)

Routeur (ou
Router)

Routeur (ou Router)

Pour éviter d'avoir trop de fichiers pour les différentes pages de notre site, nous allons passer par **un contrôleur frontal** qui va jouer le rôle de **routeur** et qui va lui chercher le bon contrôleur ensuite.



Routeur (ou Router)

Notre routeur va avoir précisément **trois fonctions** :

Enregistrer la route et l'action à effectuer par le contrôleur (GET, récupérer des informations)

Enregistrer la route et l'action à effectuer au contrôleur (POST, envoyer des informations)

Exécuter la route en traitant la requête entrante (RUN)

Routeur (ou Router)

Pour créer notre routeur, nous allons créer une classe **Router**. Celle-ci contiendra toutes les routes qui concernent notre application.



```
namespace App\Core;

class Router {
    private $routes = [];
}
```

*Pour cette classe, on complétera le fichier **Router.php** dans le dossier **src/Core***

Routeur (ou Router)

Nous allons ensuite créer une méthode qui va nous permettre de récupérer une route (Avec la méthode GET ou POST).

\$path servira à stocker la route, **\$controllerAction** représente le contrôleur ciblé et l'action à effectuer.

```
namespace App\Core;

class Router {
    private $routes = [];

    public function get($path, $controllerAction) {
        $this→routes['GET'][$path] = $controllerAction;
    }

    public function post($path, $controllerAction) {
        $this→routes['POST'][$path] = $controllerAction;
    }
}
```

Routeur (ou Router)

Dans notre fichier **index.php**, nous allons définir nos différentes routes à l'aide des méthodes créées dans la classe **Router**.

```
<?php
require_once __DIR__ . '/vendor/autoload.php';

use App\Core\Router;

// Instanciation du routeur
$router = new Router();

// Définition des routes
$router→get('/', 'App\Controller\GameController::home');
$router→get('/combat', 'App\Controller\GameController::combat');
```

Cela limite les urls de notre site à / (index.php) et /combat (combat.php).

Nous créerons **GameController** un peu plus tard lorsque nous parlerons des **contrôleurs**.

Routeur (ou Router)

Maintenant que nos différentes routes sont créées, il faut que l'on puisse en lancer une et qu'**elle traite la requête**.

Dans la classe **Router**, nous allons créer la méthode **run**.



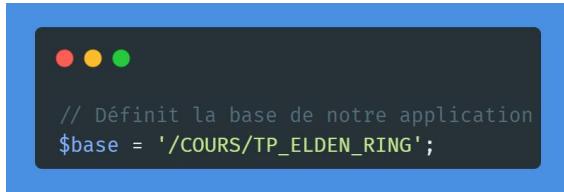
Voyons ce que nous allons mettre à l'intérieur de notre fonction **run**.

Routeur (ou Router)

```
$method = $_SERVER['REQUEST_METHOD'];
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
```

1. On demande à PHP quelle méthode a été utilisée pour accéder à notre application (par exemple, GET pour afficher une page ou POST pour envoyer des données).
 2. On récupère le chemin de l'URL. Par exemple, si l'utilisateur a visité http://localhost/COURS/TP_ELDEN_RING/index.php/combat, cette ligne extrait /COURS/TP_ELDEN_RING/index.php/combat.

Routeur (ou Router)



```
● ● ●  
// Définit la base de notre application  
$base = '/COURS/TP_ELDEN_RING';
```

Notre application est dans le dossier TP_ELDEN_RING qui se trouve dans COURS. Ainsi, la base de notre application est /COURS/TP_ELDEN_RING.

Routeur (ou Router)



```
// Retire la base si présente
if (strpos($uri, $base) === 0) {
    $uri = substr($uri, strlen($base));
}
```

Si le chemin commence par /COURS/TP_ELDEN_RING, on retire cette partie pour ne garder que ce qui suit. Par exemple, /COURS/TP_ELDEN_RING/index.php devient /index.php.

Cela permet de standardiser les routes : on ne veut pas que le chemin de base fasse partie de l'URL traitée par notre routeur.

Routeur (ou Router)



```
if ($uri === '' || $uri === '/index.php') {
    $uri = '/';
}
```

Si l'utilisateur arrive sur `http://localhost/COURS/TP_ELDEN_RING/` ou sur `http://localhost/COURS/TP_ELDEN_RING/index.php`, on considère cela comme la racine / de notre application.

Routeur (ou Router)

```
● ● ●  
if (isset($this→routes[$method][$uri])) {  
    $controllerAction = $this→routes[$method][$uri];
```

On vérifie si, pour la méthode utilisée (GET ou POST) et le chemin (par exemple, /combat), nous avons bien une action associée dans notre liste de routes.

Routeur (ou Router)

```
list($controller, $action) = explode('::', $controllerAction);
```

La chaîne de caractères associée à la route est décomposée en deux parties séparées par ::

- La première partie est le nom complet du contrôleur (par exemple, App\Controller\GameController).
- La deuxième partie est le nom de la méthode (action) à appeler (par exemple, combat).

Routeur (ou Router)

```
● ● ●

if (class_exists($controller) && method_exists($controller, $action)) {
    $instance = new $controller;
    call_user_func([$instance, $action]);
} else {
    http_response_code(404);
    echo "Controller or action not found!";
}
```

On vérifie que la classe du contrôleur existe et que la méthode demandée est bien présente dans cette classe.

- Si c'est le cas, on crée une instance du contrôleur et on appelle la méthode.
- Sinon, on renvoie un code d'erreur 404 et on affiche un message d'erreur.

Routeur (ou Router)

A blue terminal window icon with three colored dots (red, yellow, green) at the top. Inside the window, there is a snippet of PHP code:

```
    } else {
        http_response_code(404);
        echo "Route not defined!";
    }
```

Si aucune route n'est définie pour la combinaison de la méthode HTTP et de l'URI, on renvoie une erreur 404 avec un message approprié.

Router

```
$router->get('/combat', 'App\Controller\GameController::combat');
```

Prenons le cas de la route /combat pour comprendre la fonction **run** de manière plus générale.

```
public function run() {
    $method = $_SERVER['REQUEST_METHOD'];
    $uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

    // Définit la base de notre application
    $base = '/COURS/TP_ELDEN_RING';

    // Retire la base si présente
    if (strpos($uri, $base) === 0) {
        $uri = substr($uri, strlen($base));
    }

    // Si l'URI est vide ou égale à "/index.php", le définit comme "/"
    if ($uri === '' || $uri === '/index.php') {
        $uri = '/';
    }

    if (isset($this->routes[$method][$uri])) {
        $controllerAction = $this->routes[$method][$uri];
        list($controller, $action) = explode('::', $controllerAction);
        if (class_exists($controller) && method_exists($controller, $action)) {
            $instance = new $controller;
            call_user_func([$instance, $action]);
        } else {
            http_response_code(404);
            echo "Controller or action not found!";
        }
    } else {
        http_response_code(404);
        echo "Route not defined!";
    }
}
```

Étape 1 : On lit la méthode HTTP (GET) utilisée par l'utilisateur.

Étape 2 : On extrait le chemin de l'URL
(/COURS/TP_ELDEN_RING/combat.php devient /combat).

Étape 3 : On vérifie si le chemin (pour la méthode HTTP utilisée) existe dans notre tableau de routes (vu qu'on a appelé la méthode **get**, normalement la condition passe).

Étape 4 : Si oui, on décompose la chaîne de route en contrôleur et action (App\Controller\GameController et combat).

Étape 5 : On vérifie que le contrôleur et la méthode existent, puis on crée une instance du contrôleur (new GameController) et on appelle la méthode.

Étape 6 (Suite de l'**étape 3** et l'**étape 5**) : Si la route, le controller ou l'action du controller n'existe pas, on renvoie une erreur 404.

Routeur (ou Router)

```
<?php
require_once __DIR__ . '/vendor/autoload.php';

use App\Core\Router;

// Instanciation du routeur
$router = new Router();

// Définition des routes
$router->get('/', 'App\Controller\GameController::home');
$router->get('/combat', 'App\Controller\GameController::combat');

// Démarrage du routeur
$router->run();
```

Une fois la méthode **run** créé dans la classe **Router**, nous pouvons l'appeler dans l'**index.php** pour lancer le routeur.

03

Le modèle Modèle Vue Contrôleur (MVC)

Contrôleur (Ou
Controller)

Contrôleur (Ou Controller)

Contrôleur

Cette partie gère les échanges avec l'utilisateur. C'est en quelque sorte l'intermédiaire entre l'utilisateur, le modèle et la vue. Le contrôleur va recevoir des requêtes de l'utilisateur.

Dans notre cas, nous avons vu que notre routeur allait appeler les contrôleurs souhaités.

Nous allons maintenant créer notre **GameController** pour faire l'intermédiaire entre **les modèles** et **les vues**.

Contrôleur (Ou Controller)

Concrètement, que doit implémenter notre
GameController ?

- Une méthode home pour la page d'accueil
- Une méthode combat pour la page de combat



```
// Définition des routes
$router->get('/', 'App\Controller\GameController::home');
$router->get('/combat', 'App\Controller\GameController::combat');
```

Nous avons également besoin d'une méthode permettant de **définir la vue qui sera utilisé par notre route**, via notre controller.

Contrôleur (Ou Controller)

Étape 1: Crée notre **controller parent**

```
<?php
namespace App\Core;

class Controller {

    protected function render(string $view, array $params = []): void {
        // Extraction des variables pour les rendre directement accessibles dans la vue
        extract($params);
        // Inclut le fichier de vue situé dans le dossier /src/View/
        require __DIR__ . '/..../View/' . $view;
    }
}
```

Pour cette classe, on complétera le fichier **Controller.php** dans le dossier **src/Core**

Contrôleur (ou Controller)

Étape 2 : Créer notre **GameController**

Notre **GameController** héritera de **Controller**, pour pouvoir utiliser ces méthodes.



```
<?php
namespace App\Controller;

use App\Core\Controller;
use App\Service\CombatService;

class GameController extends Controller {
    // ...
}
```

A screenshot of a terminal window with a dark background and light-colored text. It shows the beginning of a PHP file for a 'GameController'. The code includes the opening tag, namespace declaration, use statements for 'Controller' and 'CombatService', and the start of the 'GameController' class definition which extends 'Controller'.

Pour cette classe, on créera le fichier **GameController.php** dans le dossier **src/Controller**

Contrôleur (Ou Controller)

Étape 3 : Créer notre méthode **home**

```
<?php  
namespace App\Controller;  
  
use App\Core\Controller;  
  
class GameController extends Controller {  
    public function home() {  
        $this→render('home.php', ['title' => 'Bienvenue dans Elden Ring Game']);  
    }  
}
```

Pour notre page **Home**, la méthode **home** permettra de “render” notre vue, en lui passant en paramètre les variables à rendre accessibles dans la vue.

Contrôleur (Ou Controller)

Étape 4 : Créez notre méthode **combat**



```
<?php
namespace App\Controller;

use App\Core\Controller;
use App\Service\CombatService;

class GameController extends Controller {
    public function home() {
        $this→render('home.php', ['title' => 'Bienvenue dans Elden Ring Game']);
    }

    public function combat() {
        $combatService = new CombatService();
        $result = $combatService→startCombat();
        $this→render('combat.php', ['result' => $result]);
    }
}
```

Pour notre page **Combat**, la méthode **combat** permettra de démarrer le combat grâce à un service (**CombatService**) et de "render" notre vue, en lui passant en paramètre les variables à rendre accessibles dans la vue.

04

Le modèle Modèle Vue Contrôleur (MVC)

Modèle (ou Repository)

Modèle (ou Repository)

Avant de créer notre service, nous allons créer les modèles qui vont aller chercher **nos héros** et **nos bosses** dans **notre base de données**.

Modèle (ou Repository)

Modèle

Cette partie gère ce qu'on appelle la logique métier de votre site. Elle comprend notamment la gestion des données qui sont stockées, mais aussi tout le code qui prend des décisions autour de ces données.

Une fois que nous avons défini l'action à réaliser avec notre contrôleur **GameController**, nous devons maintenant créer les modèles qui vont envoyer les informations que notre contrôleur souhaite.

Modèle (ou Repository)

Dans un premier temps, nous allons créer et gérer la base de données qui va nous permettre de **stocker nos données**.

Étape 1: Sur <http://localhost/phpmyadmin/>, se connecter au serveur MySQL.

Les identifiants de root :

Utilisateur : root / **Mot de passe** : Aucun mot de passe

Utilisateur : root / **Mot de passe** : root

The screenshot shows the phpMyAdmin login page. At the top, it says "Bienvenue dans phpMyAdmin". Below that is a "Langue (Language)" dropdown set to "Français - French". Underneath is a "Connexion" form with fields: "Utilisateur" (User) set to "root", "Mot de passe" (Password) set to "...", and "Choix du serveur" (Server choice) set to "MySQL". At the bottom of the form is a "Connexion" (Connection) button.

Modèle (ou Repository)

Dans un premier temps, nous allons créer et gérer la base de données qui va nous permettre de **stocker nos données**.

Étape 2 : Création de notre base de données.

The screenshot shows the phpMyAdmin interface for MySQL 3.306. On the left sidebar, under 'Bases de données', there is a list of existing databases: 'Nouvelle base de données', 'elden_ring_game', 'invoice_manager', and 'pyra_crm'. A black arrow points from the text 'Nouvelle base de données' to the 'Création d'une base de données' button in the bottom-left corner of the main panel. The main panel has three tabs: 'Paramètres généraux', 'Serveur de base de données', and 'Serveur Web'. In the 'Paramètres généraux' tab, the connection encoding is set to 'utf8mb4_unicode_ci'. In the 'Serveur de base de données' tab, the server information is listed. In the 'Serveur Web' tab, the Apache and PHP versions are shown. At the bottom of the main panel, there is a form with fields for 'Création d'une base de données' (containing 'elden_ring'), 'Collation' (containing 'utf8mb4_general_ci'), and a 'Créer' button.

utf8mb4_general_ci est l'encodage qui permet de prendre en charge la totalité des caractères Unicode. C'est l'évolution plus complète de l'**utf-8**.

Modèle (ou Repository)

Dans un premier temps, nous allons créer et gérer la base de données qui va nous permettre de **stocker nos données**.

Étape 3 : Création des tables et des données dans notre base de données.

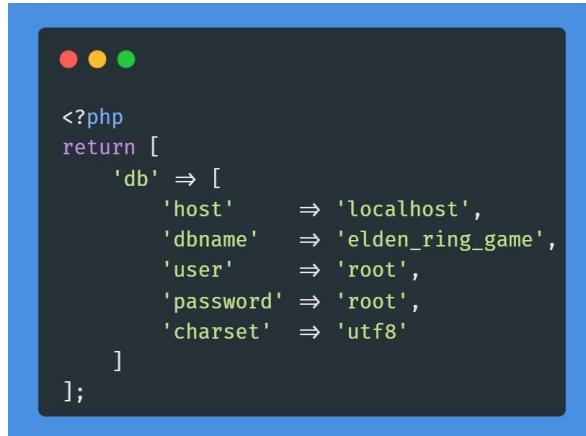
Table Heroes	Table Bosses
id	id
name	name
health	health
attackPower	attackPower
mana	
endurance	

Modèle (ou Repository)

Une fois notre base de données créée, nous allons créer une connexion à celle-ci dans notre architecture.

Étape 1: Créer la connexion à la base de données dans notre architecture MVC.

Sous-Étape 1: Créer un fichier de configuration regroupant les informations sur la connexion à la base de données.



```
<?php
return [
    'db' => [
        'host'      => 'localhost',
        'dbname'   => 'elden_ring_game',
        'user'     => 'root',
        'password' => 'root',
        'charset'  => 'utf8'
    ]
];
```

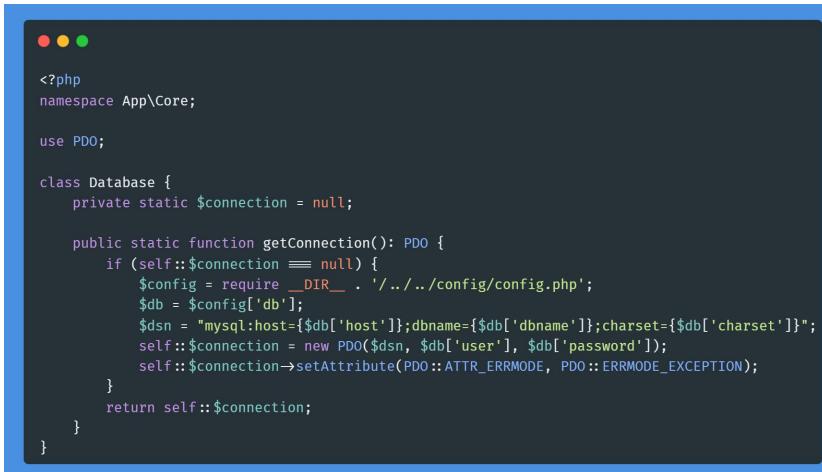
Pour la configuration, nous allons remplir le fichier **config.php** dans le dossier **config**

Modèle (ou Repository)

Une fois notre base de données créée, nous allons créer une connexion à celle-ci dans notre architecture.

Étape 1: Créer la connexion à la base de données dans notre architecture MVC.

Sous-Étape 2: Remplir notre classe **Database** pour créer une instance d'une connexion à la base de données.



```
<?php
namespace App\Core;

use PDO;

class Database {
    private static $connection = null;

    public static function getConnection(): PDO {
        if (self::$connection === null) {
            $config = require __DIR__ . '/../../config/config.php';
            $db = $config['db'];
            $dsn = "mysql:host={$db['host']};dbname={$db['dbname']};charset={$db['charset']}";
            self::$connection = new PDO($dsn, $db['user'], $db['password']);
            self::$connection->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        }
        return self::$connection;
    }
}
```

Pour la connexion à la base de données, nous allons remplir le fichier **Database.php** dans le dossier **src/Core**

Modèle (ou Repository)

Maintenant que notre base est créée et que l'on peut utiliser son instance dans notre code pour lui faire des requêtes, nous allons créer nos deux modèles pour récupérer les héros et les boss.

```
<?php
namespace App\Repository;

use App\Entity\Hero;
use App\Core\Database;
use PDO;

class HeroRepository {
    private $pdo;

    public function __construct() {
        $this->pdo = Database::getConnection();
    }
}
```

```
<?php
namespace App\Repository;

use App\Entity\Boss;
use App\Core\Database;
use PDO;

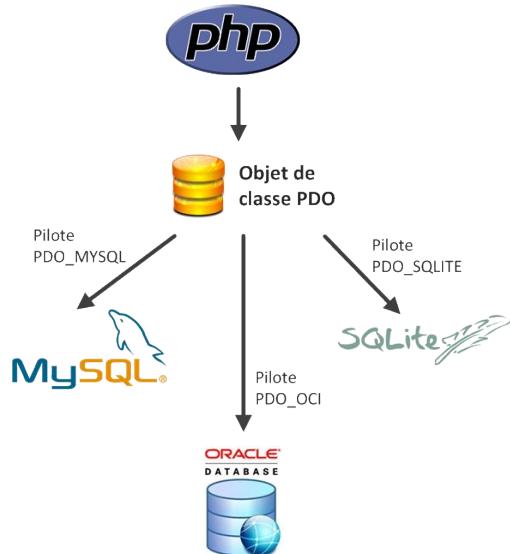
class BossRepository {
    private $pdo;

    public function __construct() {
        $this->pdo = Database::getConnection();
    }
}
```

Modèle (ou Repository)

PDO

PDO est un objet qui représente une connexion entre PHP et un serveur de base de données.



Les méthodes les plus connues de PDO sont :

PDO::connect – Connecte à une base de données

PDO::__construct – Crée une instance PDO qui représente une connexion à la base
PDO::errorCode – Retourne le SQLSTATE associé avec la dernière opération sur la base de données

PDO::errorInfo – Retourne les informations associées à l'erreur lors de la dernière opération sur la base de données

PDO::exec – Exécute une requête SQL et retourne le nombre de lignes affectées

PDO::getAttribute – Récupère un attribut d'une connexion à une base de données

PDO::lastInsertId – Retourne l'identifiant de la dernière ligne insérée ou la valeur d'une séquence

PDO::prepare – Prépare une requête à l'exécution et retourne un objet

PDO::query – Prépare et Exécute une requête SQL sans marque substitutive

PDO::setAttribute – Configure un attribut PDO

Modèle (ou Repository)

Dans chaque modèle, nous allons créer une méthode permettant de récupérer aléatoirement une entité.

```
<?php
namespace App\Repository;

use App\Entity\Hero;
use App\Core\Database;
use PDO;

class HeroRepository {
    private $pdo;

    public function __construct() {
        $this->pdo = Database::getConnection();
    }

    public function findRandom(): ?Hero {
        $stmt = $this->pdo->query('SELECT * FROM heroes ORDER BY RAND() LIMIT 1');
        $data = $stmt->fetch(PDO::FETCH_ASSOC);
        if ($data) {
            return new Hero($data['name'], $data['health'], $data['attackPower'], $data['mana'],
                $data['endurance']);
        }
        return null;
    }
}
```

Modèle (ou Repository)

Étape 1 : On fait une requête à la base de données avec son instance PDO.

Étape 2 : On fetch les données reçues pour les avoir sous forme de tableau.

Étape 3 : Si des données existent, on crée une nouvelle instance de **Boss** ou **Hero**.

Étape 4 : Sinon, on renvoie **null**.

```
<?php
namespace App\Repository;

use App\Entity\Boss;
use App\Core\Database;
use PDO;

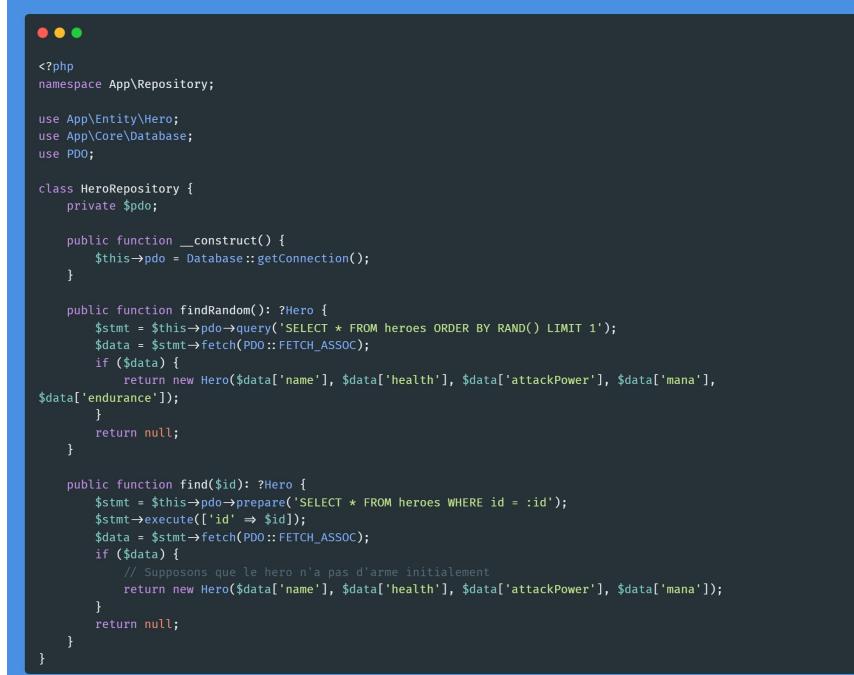
class BossRepository {
    private $pdo;

    public function __construct() {
        $this->pdo = Database::getConnection();
    }

    public function findRandom(): ?Boss {
        $stmt = $this->pdo->query('SELECT * FROM bosses ORDER BY RAND() LIMIT 1');
        $data = $stmt->fetch(PDO::FETCH_ASSOC);
        if ($data) {
            return new Boss($data['name'], $data['health'], $data['attackPower']);
        }
        return null;
    }
}
```

Modèle (ou Repository)

En plus de la méthode pour récupérer aléatoirement, nous allons aussi avoir une méthode pour récupérer une entité en fonction d'un ID.



```
<?php
namespace App\Repository;

use App\Entity\Hero;
use App\Core\Database;
use PDO;

class HeroRepository {
    private $pdo;

    public function __construct() {
        $this->pdo = Database::getConnection();
    }

    public function findRandom(): ?Hero {
        $stmt = $this->pdo->query('SELECT * FROM heroes ORDER BY RAND() LIMIT 1');
        $data = $stmt->fetch(PDO::FETCH_ASSOC);
        if ($data) {
            return new Hero($data['name'], $data['health'], $data['attackPower'], $data['mana'],
                $data['endurance']);
        }
        return null;
    }

    public function find($id): ?Hero {
        $stmt = $this->pdo->prepare('SELECT * FROM heroes WHERE id = :id');
        $stmt->execute(['id' => $id]);
        $data = $stmt->fetch(PDO::FETCH_ASSOC);
        if ($data) {
            // Supposons que le hero n'a pas d'arme initialement
            return new Hero($data['name'], $data['health'], $data['attackPower'], $data['mana']);
        }
        return null;
    }
}
```

Modèle (ou Repository)

Étape 1 : On fait une requête **préparée** à la base de données avec son instance PDO.

Étape 2 : On exécute la requête **préparée** avec **l'id du héros** en paramètre.

Étape 3 : On fetch les données reçues pour les avoir sous forme de tableau.

Étape 4 : Si des données existent, on crée une nouvelle instance de **Boss** ou **Hero**.

Étape 5 : Sinon, on renvoie **null**.

```
<?php
namespace App\Repository;

use App\Entity\Boss;
use App\Core\Database;
use PDO;

class BossRepository {
    private $pdo;

    public function __construct() {
        $this->pdo = Database::getConnection();
    }

    public function findRandom(): ?Boss {
        $stmt = $this->pdo->query('SELECT * FROM bosses ORDER BY RAND() LIMIT 1');
        $data = $stmt->fetch(PDO::FETCH_ASSOC);
        if ($data) {
            return new Boss($data['name'], $data['health'], $data['attackPower']);
        }
        return null;
    }

    public function find(int $id): ?Boss {
        $stmt = $this->pdo->prepare('SELECT * FROM bosses WHERE id = :id');
        $stmt->execute(['id' => $id]);
        $data = $stmt->fetch(PDO::FETCH_ASSOC);
        if ($data) {
            return new Boss($data['name'], $data['health'], $data['attackPower']);
        }
        return null;
    }
}
```

Modèle (ou Repository)

BONUS : On peut aussi créer une fonction dans nos modèles pour créer une nouvelle entité de Hero ou de Boss.

```
● ● ●

public function save(Hero $hero): bool {
    $stmt = $this→pdo→prepare('INSERT INTO heroes (name, health, attackPower, mana) VALUES (:name, :health,
:attackPower, :mana)');
    return $stmt→execute([
        'name' ⇒ $hero→getName(),
        'health' ⇒ $hero→getHealth(),
        'attackPower' ⇒ $hero→getAttackPower(),
        'mana' ⇒ $hero→getMana()
    ]);
}
```

Cette fois-ci, il suffit juste d'exécuter la requête **préparée**, aucun traitement supplémentaire n'est nécessaire.

05

Le modèle Modèle Vue Contrôleur (MVC)

Service

06

Le modèle Modèle Vue Contrôleur (MVC)

Vue (ou View)