

# CONCEPTION ET ARCHITECTURE WEB & IHM

<?= “2. PDO et POO” ?>

# Avant de commencer

---

Version de PHP

Ce cours se base sur PHP 7/8.



Rappel doc & stackoverflow-like

→ ne recopiez pas bêtement

→ la seule aide fiable est la doc officielle sur [php.net](https://php.net)

# Ce que l'on a vu

---

## 1. Concept client-serveur

- ✓ mécanisme de requête/réponse via le protocole HTTP

## 2. Intro PHP

- ✓ intégration intrinsèque dans du code HTML
- ✓ syntaxe : variables, tableaux, boucles et conditions
- ✓ inclusion de fichiers

## 3. Échange de données client ↔ serveur

- ✓ méthodes GET (`$_GET`) et POST (`$_POST`)
- ✓ vérification et protection des données transmises par le client : `isset`, `htmlentities`

## 4. Persistance des données

- ✓ cookies (`$_COOKIE`) → données stockées chez le client
- ✓ sessions (`$_SESSION`) → données stockées sur le serveur

# Au programme

---

1. Fonctions : syntaxe, typage
2. POO : classes et objets, mot-clé `this`
3. Exceptions : `try` / `catch` / `throw`
4. SQLite & PDO : requêtes simples et préparées
5. MVC : spécificités d'une application client-serveur
6. Héritage : interfaces et classes abstraites





# Les fonctions

# Syntaxe

```
function doIt( bool $arg, int $i ) : void
{
    // Traitements...
}
```

```
doIt( true, 1 );
```

*Sans retour*

```
function doItAgain() : string
{
    // Traitements...
    return $result;
}
```

```
$myString = doItAgain();
```

*Avec retour*

- ✓ Du typage statique dans PHP (seulement depuis la v.7) pour :
  - les arguments,
  - la valeur de retour d'une fonction.
- ✓ Le typage n'est pas obligatoire mais fortement recommandé.
- ✓ Les principaux types : `array`, `bool`, `float`, `int`, `string` et `object`.

# Transtypage (cast) coercitif

En PHP, typer signifie transtyper si le type utilisé n'est pas le bon.

```
function add( float $a, float $b ) : float {  
    return $a + $b;  
}  
echo add( 3, 4 );           // Affiche 7  
echo add( 3, 4.99 );       // Affiche 7.99  
echo add( 3, "4.3" );      // Affiche 7.3  
echo add( 3, "plop" );     // Error
```

*Transtypage des arguments*

```
function add( float $a, float $b ) : int {  
    return $a + $b;  
}  
echo add( 3, 4 );           // Affiche 7  
echo add( 3, 4.99 );       // Affiche 7  
echo add( 3, "4.3" );      // Affiche 7  
echo add( 3, "plop" );     // Error
```

*Transtypage de la valeur de retour*

Note : Il est possible d'imposer un typage strict en déclarant cette instruction en début de fichier :

```
declare(strict_types=1);
```



# Classes et objets

# Syntaxe

---

```
class Truc
{
    public int $membre;
    public function methode() { ... }
}
```

*Définition de la classe* Truc

```
$obj = new Truc();

echo var_dump($obj);
// Affiche :
//      object(Truc)#1 (1) { ["membre"]=> NULL }
```

*Instanciation d'un objet de type* Truc

## Définition classe vs. objet (rappels)

...

## Une classe

C'est la description d'un objet comme ensemble d'attributs et de méthodes. Une classe n'a pas d'existence propre.

## Un objet

C'est une instance d'une classe donnée. Il existe de sa création avec `new` jusqu'à sa destruction.

# Les attributs

```
class MaClasse {  
    public string $attribut1;  
    public int $attribut2 = 31;  
}
```

```
// Création d'un objet  
// de type MaClasse  
  
$obj = new MaClasse();
```

```
// Accès aux attributs publics  
echo $obj->attribut2;  
// Modification d'un attribut  
$obj->attribut1 = "CAWEBI";
```

## Définition

Aussi appelé “propriété”, c’est une variable interne d’une classe.

## Un attribut :

- ✓ ~~peut~~ doit être spécifié avec une **visibilité**
- ✓ ~~peut~~ doit être spécifié avec un **type**
- ✓ est accessible depuis un objet avec le symbole `->` suivi de son nom sans le symbole `$`

*Attention, le **typage des attributs** ne fonctionne qu’avec PHP >= 7.4 !*



# Les méthodes

```
class MaClasse
{
    private int $val;
    public function __construct( int $valeur )    { $this->val = $valeur; }
    public function __destruct()                {}
    public function getVal() : int                { return $this->val; }
    public function setVal( int $valeur ) : void { $this->val = $valeur; }
}
```

```
// Création d'un objet
$obj = new MaClasse(5);

echo $obj->getVal();
// Affiche "5"
$obj->setVal( 31 );
echo $obj->getVal();
// Affiche "31"
```

## Définition

Une méthode est une fonction interne à une classe. Elle est appelée sur un objet pour connaître son état ou le modifier.

Le mot-clé `$this` :

- ✓ représente n'importe quel objet instance de cette classe.
- ✓ est obligatoire en PHP (à l'inverse de Java par exemple)

→ L'implémentation d'une méthode définit un comportement pour tous les objets `$this` sur lesquels elle sera appelée.

# Exceptions

# Syntaxe

```
function diviser( int $num, int $denom ) : float
{
    if ( $denom == 0 ) {
        throw new Exception("Division par zéro");
    }
    return $num / $denom;
}
```

*Déclenchement d'une exception.*

```
try {
    echo diviser(4,5);
    echo diviser(4,0);
}
catch ( Exception $e ) {
    echo "Erreur : " . $e->getMessage();
}
```

*Capture d'une exception.*

## Définition

Les exceptions sont un mécanisme de gestion des comportements indésirables qui interrompt l'exécution d'une instruction.

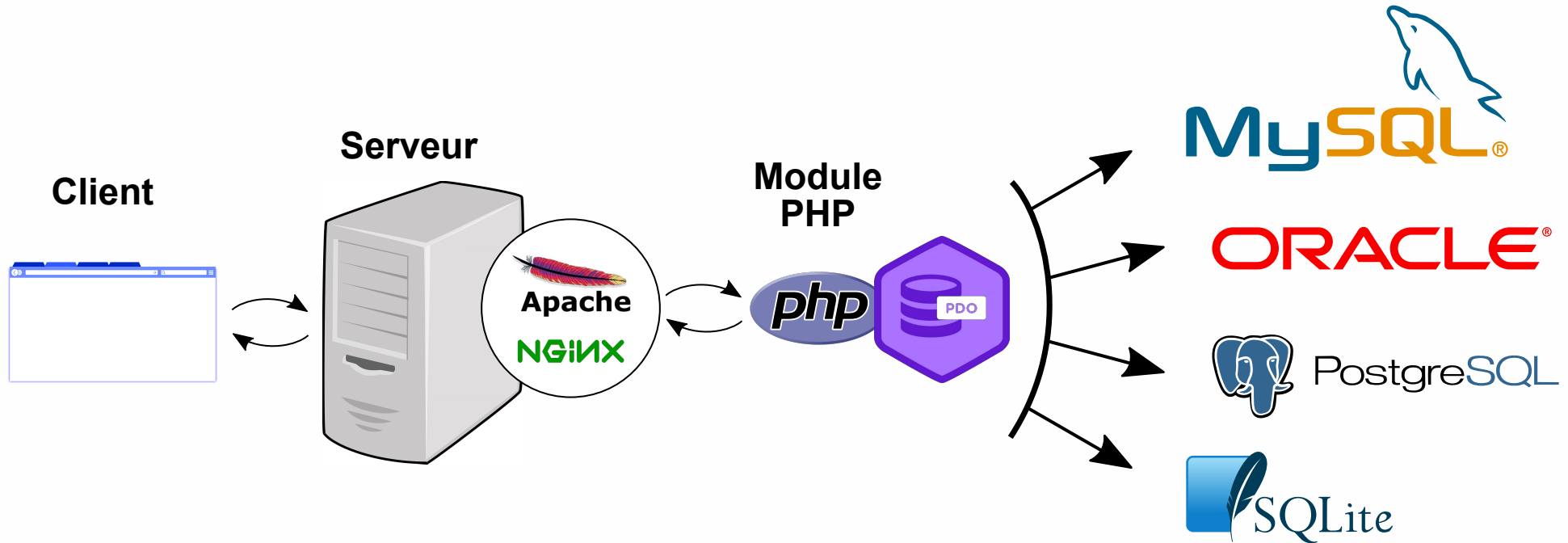
Une exception :

- ✓ se lance avec l'instruction `throw`
- ✓ se capture avec un bloc `try`
- ✓ se traite avec un bloc `catch`
- ✓ doit être de type `Exception` ou d'un sous-type

# SQLite & PDO



# Présentation



En PHP, il est nécessaire d'utiliser une extension pour manipuler une base de données. La référence en la matière est l'extension PDO.

PDO est une interface d'abstraction, c'est-à-dire qu'elle permet d'écrire du code indépendamment du SGBD utilisé.

# Création d'un objet PDO

```
$SQL_DSN = 'sqlite:/home/.../bdd.db';

try {
    $pdo = new PDO($SQL_DSN);
}
catch( PDOException $e ) {
    echo 'Erreur : ' . $e->getMessage();
    exit;
}
```

*SQLite*

```
$SQL_DSN = 'mysql:host=bdd.site.fr;dbname=cawebi';
$SQL_LOGIN = 'login';
$SQL_PASS = 'password';

try {
    $pdo = new PDO($SQL_DSN, $SQL_LOGIN, $SQL_PASS);
}
catch( PDOException $e ) {
    echo 'Erreur : ' . $e->getMessage();
    exit;
}
```

*MySQL/MariaDB*

La création d'un objet PDO pour SQLite nécessite un seul argument, le **DSN**, contenant le type de SGBD et le chemin absolu du fichier de BDD.

*Attention : Une `PDOException` est déclenchée si la connexion échoue, il faut toujours la “try-catcher” !*

# Requêtes simples

**PDO::query( string \$statement ) : PDOStatement**

- ✓ Pour interroger la BDD (SELECT).
- ✓ Retourne un objet PDOStatement contenant les données trouvées.
- ✓ Retourne false en cas d'erreur.

```
$request = $pdo->query( "SELECT * FROM Students" );
```

**PDO::exec( string \$statement ) : int**

- ✓ Pour modifier la BDD (INSERT, UPDATE, DELETE, etc.)
- ✓ Retourne le nombre de lignes modifiées.

```
$nbInsertions = $pdo->exec( "INSERT INTO Students(name) VALUES('Josh')" );  
$nbUpates = $pdo->exec( "UPDATE Students SET name='John' WHERE name='Josh'" );  
$nbDeletions = $pdo->exec( "DELETE FROM Students WHERE name='John'" );
```

# Parcours d'un "PDOStatement"

```
$request = $pdo->query( "SELECT name FROM Students" );
```

`$request` est un objet de la classe `PDOStatement`.

```
foreach( $request as $row ) {  
    echo 'Nom : ' . $row['name'];  
}
```

*Parcours avec* `foreach`.

```
while ( $row = $request->fetch() ) {  
    echo 'Nom : ' . $row['name'];  
}
```

*Parcours avec* `while` et `fetch`.

En complément de `foreach`, la méthode `fetch()` de la classe `PDOStatement` permet de parcourir les données obtenues :

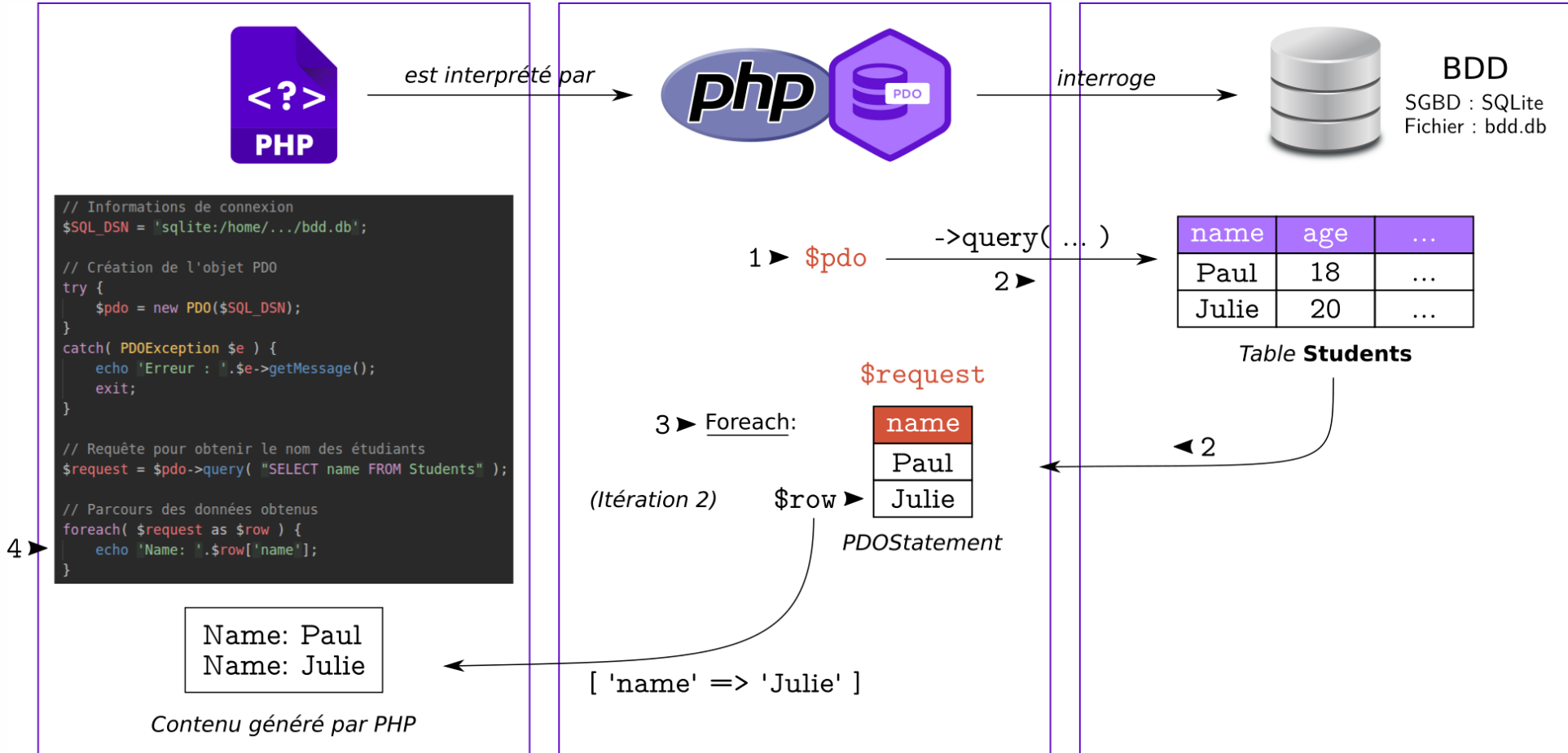
**`PDOStatement::fetch()` : mixed**

- ✓ Retourne la prochaine ligne de données d'un `PDOStatement`.
- ✓ Retourne `false` s'il n'y a pas de prochaine ligne.

*L'intérêt de la méthode `fetch` se trouve dans les paramètres qu'on peut lui spécifier → voir documentation.*



# Le parcours en action



# Requêtes préparées

**PDO::prepare( string \$statement ) : PDOStatement**

- ✓ Prépare une requête mais ne l'exécute pas.
- ✓ Se paramètre avec des marqueurs
- ✓ Retourne un objet de type `PDOStatement`.

```
$request = $pdo->prepare( "SELECT * FROM Students WHERE age > :value" );
```

**PDOStatement::bindValue( mixed \$param, mixed \$val [, int \$type ] ) : bool**

- ✓ Associe la valeur `$val` au marqueur `$param`

```
$request->bindValue( ':value', 20, PDO::PARAM_INT );
```

**PDOStatement::execute() : bool**

- ✓ Exécute la requête préparée

```
$request->execute();
```

# Marqueurs indexés vs. nommés

```
// 1.a Prépare la requête
$request = $pdo->prepare(
    "SELECT name FROM Students
    WHERE note > ? AND note < ?"
);
// 2.a Assigne 10 au 1er paramètre
$ok1 = $request->bindValue(
    1, 10, PDO::PARAM_INT
);
// 3.a Assigne 20 au 2ème paramètre
$ok1 &= $request->bindValue(
    2, 20, PDO::PARAM_INT
);
```

Marqueurs indexés : ?

```
// 1.b Prépare la requête
$request = $pdo->prepare(
    "SELECT name FROM Students
    WHERE note > :min AND note < :max"
);
// 2.b Assigne 10 au paramètre :min
$ok1 = $request->bindValue(
    ':min', 10, PDO::PARAM_INT
);
// 3.b Assigne 20 au paramètre :max
$ok1 &= $request->bindValue(
    ':max', 20, PDO::PARAM_INT
);
```

Marqueurs nommés : :name

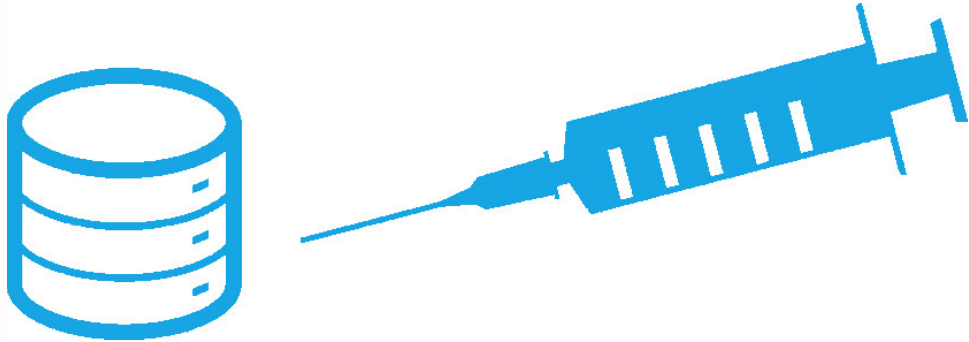
```
// 4. On exécute la requête $request
$ok1 &= $request->execute();

// 5. On parcourt et affiche les données trouvées dans la base
foreach ( $request as $row ) {
    echo "M. " . $row['name'] . ' a une note > 10';
}
```

Parcours des données identique.

# Requêtes simples vs. préparées

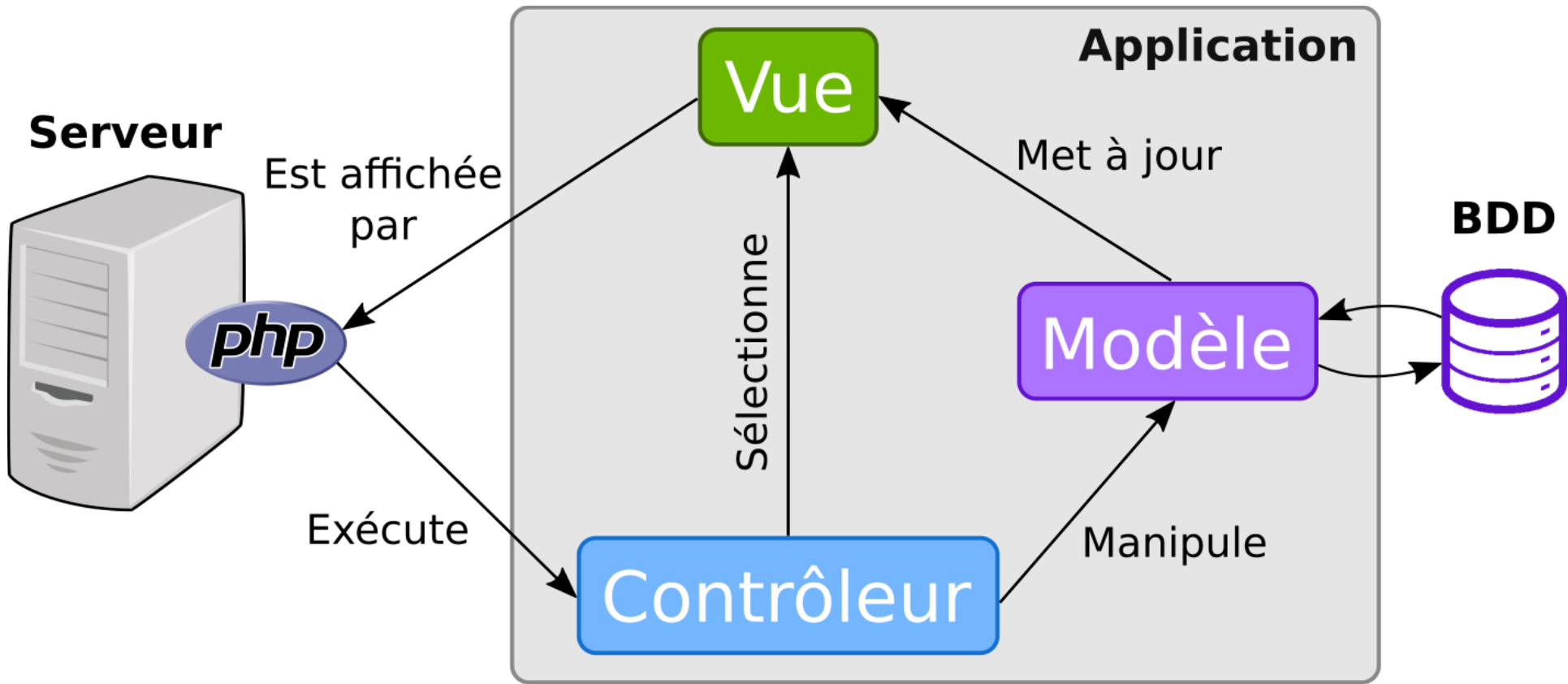
---



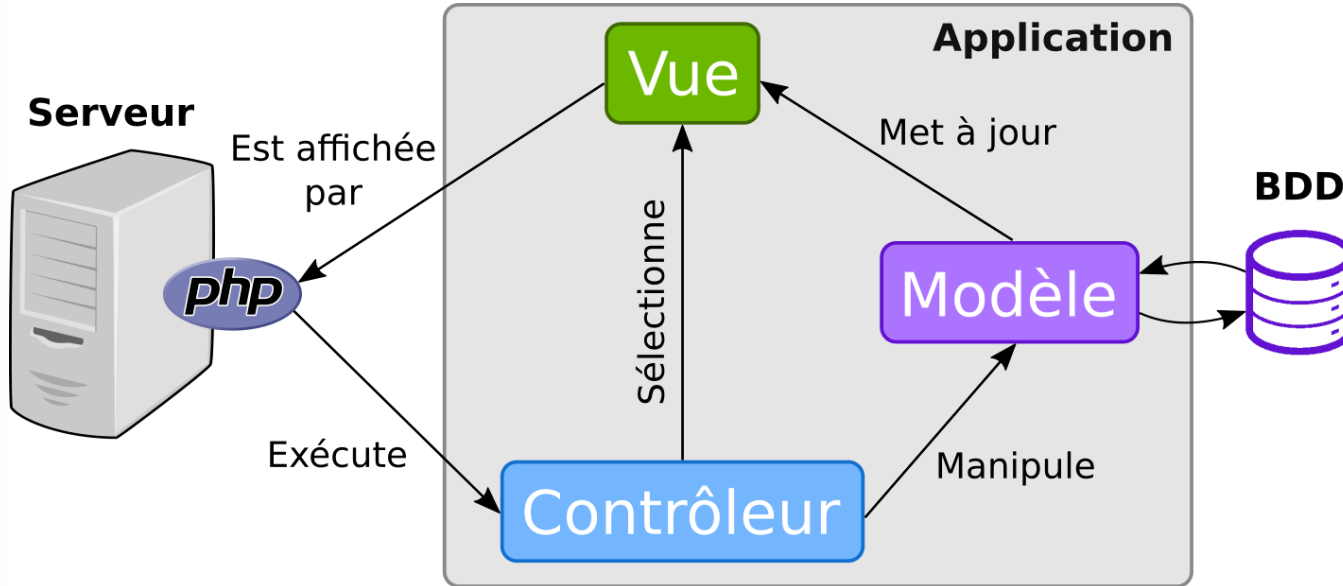
## Les requêtes préparées

- ✓ protègent des injections SQL grâce aux marqueurs
- ✓ sont réutilisables
- ✓ permettent d'utiliser moins de ressources
- ✓ s'exécutent plus rapidement
- ✓ affranchissent de la préoccupation des guillemets

# Conception MVC



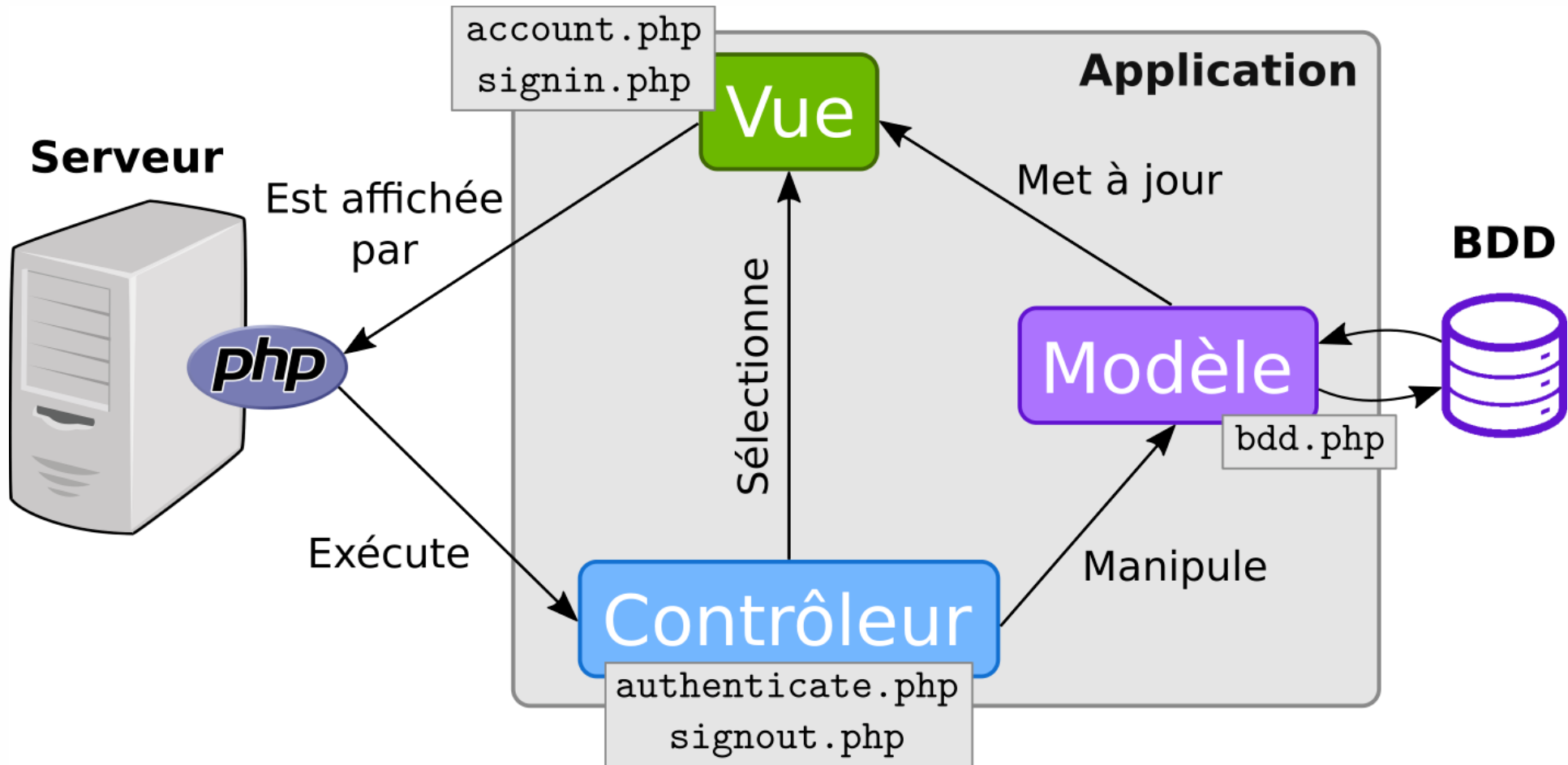
# Principe



MVC (*Modèle-Vue-Contrôleur*) est une architecture logicielle qui propose un découpage en trois entités :

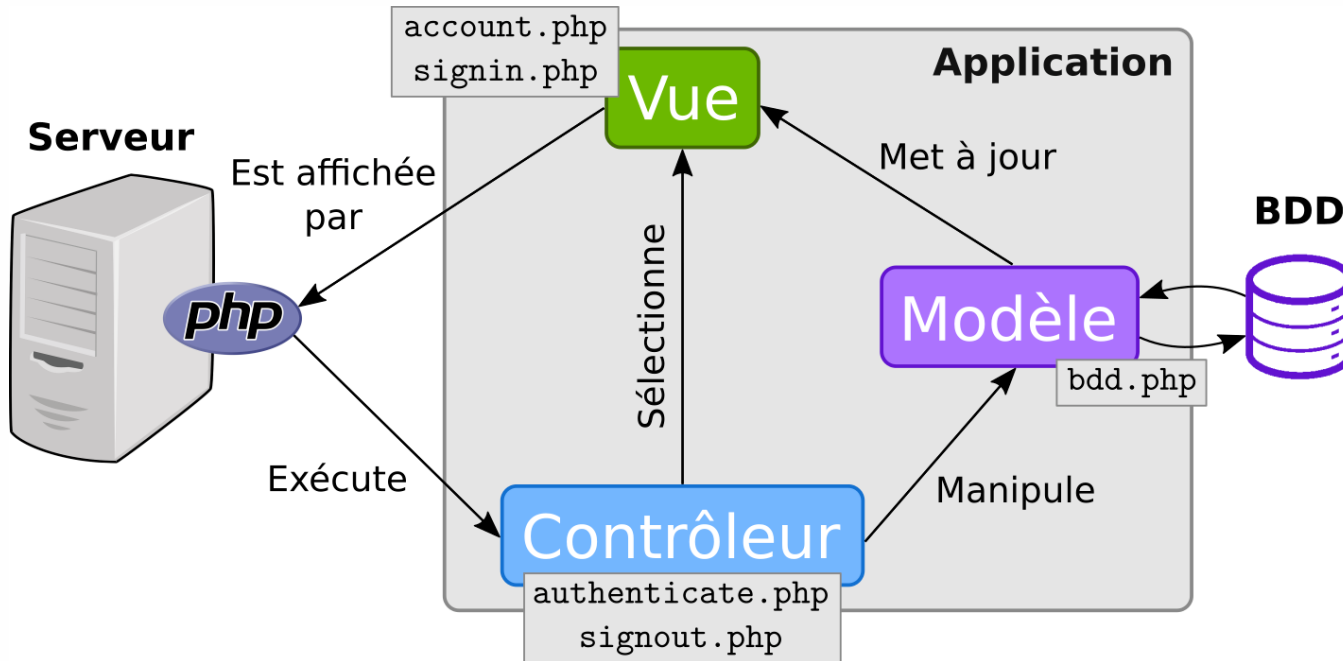
- ✓ les **Modèles** qui contiennent les données,
- ✓ les **Vues** qui définissent l’affichage du modèle,
- ✓ les **Contrôleurs** qui contiennent la logique de l’application et répondent aux requêtes en coordonnant les modèles et les vues.

# En pratique





# En pratique



## Vues

- ✓ `signin.php` affiche le formulaire d'authentification.
- ✓ `account.php` affiche la page d'accueil personnalisée.

# Héritage

# Les interfaces

```
interface Module
{
    function name() : string;
    function prof() : string;
}
```

```
class A1 implements Module
{
    public string $prof;
    function __construct( string $prof ) {
        $this->prof = $prof;
    }
    function name() : string { return "CAWEBI"; }
    function prof() : string { return $this->prof; }
}
```

## Définition

Une interface spécifie la signature des méthodes que doit définir une classe qui l'implémente.

Une interface :

- ✓ ne spécifie que des fonctions publiques
- ✓ ne peut pas contenir d'attribut
- ✓ ne peut pas définir l'implémentation d'une méthode

# Intérêt des interfaces

```
interface Module
{
    function name() : string;
    function prof() : string;
}
function printModule( Module $mod )
{
    echo $mod->name(). ' : ' . $mod->prof();
}
```

```
class A1 implements Module {
    function name() : string { return "CAWEBI"; }
    function prof() : string { return "FRE"; }
}

class A2 implements Module {
    function name() : string { return "ALGO"; }
    function prof() : string { return "SAR"; }
}
```

Définir une fonction qui manipule un objet implémentant une interface donnée garantit qu'il possède les méthodes de l'interface.

```
$mod1 = new A1();
printModule($mod1);
// Affiche "CAWEBI : FRE"
```

```
$mod2 = new A2();
printModule($mod2);
// Affiche "ALGO : SAR"
```

→ la fonction est donc... générique par polymorphisme !

# Les classes abstraites

```
abstract class Module
{
    private string $name;

    function __construct( string $name ) {
        $this->name = $name;
    }
    function print() : void {
        echo $this->name.' : '.$this->prof();
    }
    abstract function prof() : string;
}
```

```
class A1 extends Module
{
    function __construct() {
        parent::__construct("CAWEBI");
    }
    function prof() : string {
        return "FRE";
    }
}

$obj = new A1();
$obj->print(); // Affiche "CAWEBI : FRE"
```

## Définition

Une classe abstraite est à mi-chemin entre interface et classe concrète. Elle peut comporter des méthodes abstraites ET concrètes.

Une classe abstraite :

- ✓ ne peut pas être instanciée
- ✓ peut contenir des attributs

# Intérêt des classes abstraites

```
abstract class MaClasseAbstraite
{
    private $attribut1;
    protected $attribut2;

    abstract public function methode1();
    abstract public function methode2();
    public function methode3() { ... }
    public function methode4() { ... }
}
```

```
class MaClasse extends MaClasseAbstraite
{
    public function methode1() { ... }
    public function methode2() { ... }
    public function methode4() { ... }
}
```

```
$obj = new MaClasse();
$obj->methode2(); // OK
$obj->methode3(); // OK
$obj->methode4(); // OK
```

Comme les interfaces, elles permettent d'écrire du code générique. De plus, elles permettent de mutualiser du code.

Une classe qui hérite d'une classe abstraite :

- ✓ doit définir toutes les méthodes `abstract` de la classe mère
- ✓ peut surcharger une méthode concrète