



# **M205 Développer en back-end**

## **Support du cours**



## Table des matières

1. Programmer en PHP .....	2
2. Programmation orienté objet : .....	14
3. Développer une application avec PHP et MySQL .....	23
4. Découvrir les notions fondamentales des frameworks PHP .....	26
5. Préparer l'environnement de Laravel .....	35
6. Créer des routes .....	43
7. Créer des vues .....	50
8. Créer des contrôleurs .....	57
9. Créer et utiliser les modèles .....	61
10. Gérer les relations .....	77
11. Traiter les formulaires .....	88

## 1. Programmer en PHP

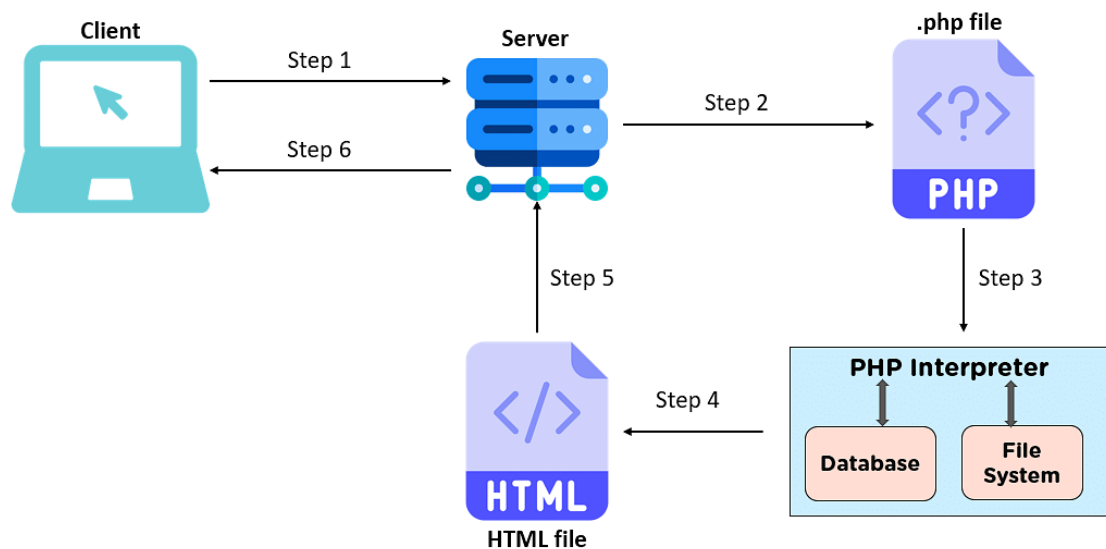
### 1.1. Introduction

PHP est un langage **interprété orienté Web**. Syntaxiquement, c'est un mélange de **C** et de **Perl**. Les scripts PHP sont lus et interprétés par le moteur PHP.

PHP comporte plus de 500 fonctions. Il est fourni avec des bibliothèques offrant des fonctionnalités diverses : accès aux bases de données, fonctions d'images, sockets, protocoles Internet divers...

### 1.2. Principe de fonctionnement :

Lorsqu'une requête HTTP est soumise au serveur Web pour une page dont l'extension est «**.php**», le serveur commence par rechercher dans son arborescence le fichier d'extension «**.php**». Il va ensuite passer la main à un sous-processus qui va **interpréter le script PHP** et produire dynamiquement du code HTML. Ce code HTML est alors envoyé au travers du réseau au navigateur client. De plus, aucune ligne de code PHP n'apparaît côté client dans la mesure où tout le code a été interprété.



### 1.3. Notions de base :

#### Mélange HTML/PHP

- PHP s'intègre dans l'HTML entre **<?php** ou **<?** et **?>**
- Les instructions se finissent par point-virgule ;
- Les commentaires sont soit entre **/\*** et **\*/**, soit après **//** ou **#**

#### Les variables

- En PHP, les variables sont représentées par un signe dollar "\$" suivi du nom de la variable.
- Le nom est sensible à la casse : **\$var** != **\$Var**
- Pas de déclaration, typage implicite

- Les variables PHP sont à typage faible. C'est PHP qui décide de son type lors de l'affectation.
- Il existe plusieurs types de données :
  - Entier (int, integer)
  - Décimal (real, float, double)
  - Chaîne de caractères (string)
  - Tableau (array)
  - Objet (object)
  - Booléen (boolean)
- La fonction **gettype** permet de connaître le type de la variable. Elle renvoie une chaîne : "string" ou "integer" ou "double" ou "array" ou "object".

```
<?php
    $number = 10;
    $name = 'ALLAOUI';
    echo gettype($number)." ".gettype($name);
```

**Affichage :**

integer string

- On peut également tester un type particulier à l'aide des fonctions `is_array`, `is_string`, `is_int`, `is_float`, `is_object`.

```
<?php
    $number = 10;
    $name = 'ALLAOUI';
    echo is_integer($number)." ".is_string($name);
```

**Affichage :**

1 1

- Il est parfois utile de forcer le type d'une variable. On utilise la fonction **settype** ou les **opérateurs de casting** ou bien utiliser **strval**, **intval**, **doubleval**, **boolval** qui renvoient la variable convertie en chaîne / entier / réel / booléen.

```
<?php
    $number = 10;
    echo gettype($number)." ";
    settype($number, "double");
    echo gettype($number)." ";
    $numstr = strval($number);
    echo gettype($numstr);
```

**Affichage :**

integer double string

- La fonction **isset** permet de tester si une variable est définie.
- La fonction **unset** permet de supprimer la variable, et de dés-allouer la mémoire utilisée.

```
<?php
    $number = 10;
    if (isset($number)) echo "Number est défini".PHP_EOL;
    unset($number);
    if (!isset($number)) echo "Number est non défini";
```

**Affichage :**

```
Number est défini
Number est non défini
```

### Les constantes

- PHP permet de définir des constantes à l'aide de la fonction **define**.
- On les utilise directement (sans \$)

```
<?php
    define("PI", 3.14);
    echo PI;
```

**Affichage :**

```
3.14
```

### – Constantes magiques

- `__LINE__` La ligne courante dans le fichier.
- `__FILE__` Le chemin complet et le nom du fichier courant avec les liens symboliques résolus.
- `__DIR__` Le dossier du fichier.
- `__CLASS__` Le nom de la classe courante.
- `__METHOD__` Le nom de la méthode courante.

### Les entrées/sorties

- Les entrées à l'aide de formulaires
- Les sorties On peut afficher avec la commande **echo** (avec ou sans parenthèses)
- **print** est équivalente à **echo**

### Opérateurs

- Arithmétiques : + - \* / % ++ --
- Affectation : = .= += -= \*= /= %=
- Comparaison : == < != > === <= !== >=
- Logiques : and && or || not !
- Conditionnel : ... ? ... : ...

## Instructions conditionnelles

- L’instruction conditionnelle **if**

Structure	Exemple
<pre>if ( cond ) {     ... } elseif ( cond ) {     ... } else {     ... }</pre>	<pre>&lt;?php     \$note = 15;     if (\$note&lt;10)         echo "redoublant";     elseif (\$note&gt;10 and \$note&lt;15)         echo "moyen";     else         echo "excellent";</pre>
<b>Affichage :</b>	
excellent	

- L’instruction conditionnelle **switch**

Structure	Exemple
<pre>switch ( expr ) {     case VALEUR1 :         ...         break ;     default :         ...         break ; }</pre>	<pre>&lt;?php     \$couleur = 'Vert';     switch (\$couleur) {         case 'Vert':             echo "Démarrer"; break;         case 'Rouge':             echo "Stop"; break;         case 'Orange':             echo "Attention"; break;     }</pre>
<b>Affichage :</b>	
Démarrer	

## Instructions itératives

- La boucle **for**

Structure	Exemple
<pre>for ( init ; cond ; modif ) {     ... }</pre>	<pre>&lt;?php     for (\$i=1; \$i&lt;5; \$i++) {         echo \$i.' ';     }</pre>
<b>Affichage :</b>	
1 2 3 4	

– La boucle **while**

Structure	Exemple
<pre>while ( cond ) {     ... }</pre>	<pre>&lt;?php     \$i=1;     while (\$i&lt;5) {         echo \$i.' ';         \$i++;     }</pre>
<b>Affichage :</b>	
1 2 3 4	

– La boucle **do while**

Structure	Exemple
<pre>do {     ... } while ( cond ) ;</pre>	<pre>&lt;?php     \$i=1;     do {         echo \$i.' ';         \$i++;     } while (\$i&lt;5);</pre>
<b>Affichage :</b>	
1 2 3 4	

## Les tableaux

– **Introduction**

- Chaque élément du tableau a une clé et une valeur
- Les valeurs des éléments ne sont pas forcément toutes du même type

```
<?php
    $fruits= array();
    $fruits[0]= "pomme";
    $fruits[1]= "banane";
    $fruits[] .= "orange"; // équivaut a $fruits[2]= "orange"
    $fruits= array( "pomme", "banane", "orange" );
```

– **Parcours de tableaux**

– Parcours classique avec **for**

Structure	Exemple
<pre>for (init; cond; modif) {     ... }</pre>	<pre>&lt;?php     \$numbers = array(1, 2, 3, 4);     for(\$i=0;\$i&lt;4;\$i++)         echo \$numbers[\$i].' ';</pre>
<b>Affichage :</b>	
1 2 3 4	

– Parcours spécifique avec **foreach**

Structure	Exemple
<pre>foreach (\$tab as \$value) {     ... } foreach (\$tab as \$key =&gt; \$value) {     ... }</pre>	<pre>&lt;?php \$numbers= array(1, 2, 3, 4); foreach (\$numbers as \$number)     echo \$number.' ';</pre>
<b>Affichage :</b>	
1 2 3 4	

– **Fonctions prédéfinies**

- **sizeof (\$tab)** : Renvoie le nombre d'éléments d'un tableau.
- **is\_array (\$tab)** : renvoie true si la variable est de type tableau (ou tableau associatif), false sinon.
- **count (\$tab)** : compte le nombre d'éléments initialisés
- **current (\$tab)** : retourne la valeur de l'élément en cours
- **key (\$tab)** : retourne l'indice de l'élément en cours
- **reset (\$tab)** : déplace le pointeur vers le premier élément
- **list (\$indice, \$valeur)** avec **each (\$tab)** : permettent de parcourir les couples (indice, valeur) même si les indices ne sont pas consécutifs
- **next (\$tab)** : déplace le pointeur vers l'élément suivant
- **prev (\$tab)** : déplace le pointeur vers l'élément précédent
- **sort (\$tab)** : trie les valeurs et réaffecte les indices
- **asort (\$tab)** : trie les valeurs et ne réaffecte pas les indices
- **rsort (\$tab)** : id sort mais dans l'ordre inverse
- **arsort (\$tab)** : id asort mais dans l'ordre inverse
- **ksort (\$tab)** : trie les indices
- **krsort (\$tab)** : id ksort mais dans l'ordre inverse
- **usort (\$tab, \$critere), uasort (\$tab, \$critere), uksort (\$tab, \$critere)** : trie selon un critère

## Les chaînes de caractères

– **Introduction**

- Une chaîne de caractères est une série de caractères, où un caractère est la même chose qu'un octet.
- La façon la plus simple de spécifier une chaîne de caractères est de l'entourer de guillemets simples (le caractère ').
- Si la chaîne de caractères est entourée de guillemets doubles ("), PHP interprétera les séquences d'échappement pour les caractères spéciaux.



```
<?php
    $number = 18;
    echo 'Numéro $number'.PHP_EOL;
    echo "Numéro $number";
```

**Affichage :**

```
Numéro $number
Numéro 18
```

- Une troisième façon de délimiter une chaîne de caractères est la syntaxe Heredoc : <<<. Après cet opérateur, un identifiant est fourni, suivi d'une nouvelle ligne. La chaîne de caractères en elle-même vient ensuite, suivie du même identifiant pour fermer la notation.

```
<?php
    $age = 24;
    $text = <<<END
    Je suis Mohamed ALLAOUI
    et j'ai $age ans
    END;
    echo $text;
```

**Affichage :**

```
Je suis Mohamed ALLAOUI
et j'ai 24 ans
```

### – Fonctions prédéfinies

- **strlen( \$str )** : longueur de \$str
- **strcmp(ch1, ch2)** : Comparaison (==)
- **str\_repeat(ch, nb)** : répétition
- **strtolower(ch)** : minuscules
- **strtoupper(ch)** : majuscules
- **ucwords(ch)** : initiales en majuscules
- **ucfirst(ch)** : 1er lettre en majuscule
- **ltrim(ch, liste)** : suppression de caractères au début
- **rtrim(ch, liste)** : suppression de caractères à la fin
- **trim(ch, liste)** : suppression de caractères au début et à la fin
- **strstr(ch, ch2)** : recherche sensible à la casse
- **stristr(ch, ch2)** : recherche insensible à la casse
- **substr(ch, indice, N)** : extraction de chaîne de caractères
- **substr\_count(ch, ssch)** : décompte du nombre d'occurrence d'une sous-chaîne
- **str\_replace(oldssch, newssch, ch)** : remplacement
- **strpos(ch, ssch)** : position



## Les variables superglobales PHP

### – Les variables superglobales PHP

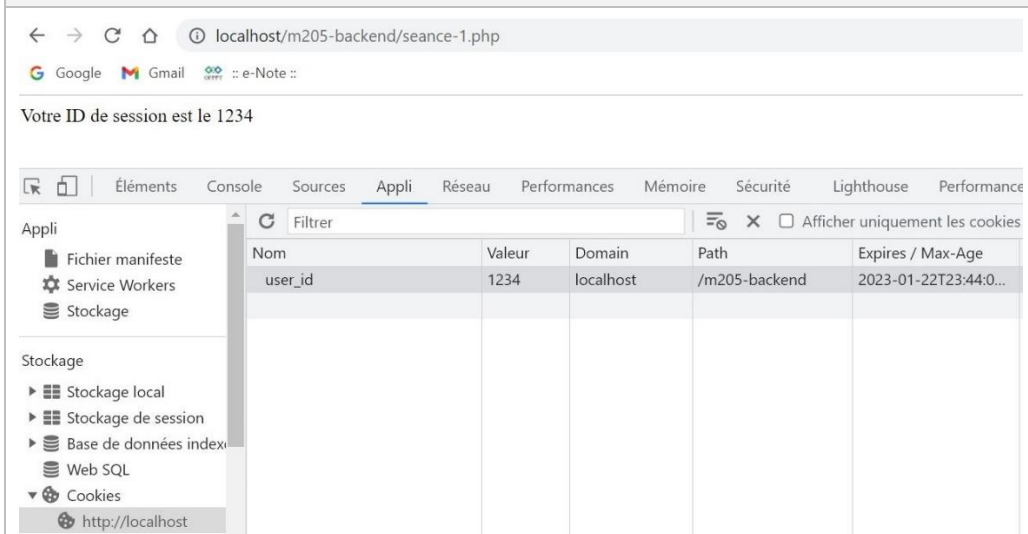
- Les variables superglobales sont des variables internes au PHP, ce qui signifie que ce sont des variables créées automatiquement par le PHP.
- Ces variables vont être accessibles n'importe où dans le script et quel que soit le contexte, qu'il soit local ou global. C'est d'ailleurs la raison pour laquelle on appelle ces variables « superglobales ».
- Les variables superglobales PHP sont les suivantes :
  - **\$GLOBALS** permet d'accéder à des variables définies dans l'espace global depuis n'importe où dans le script et notamment depuis un espace local
  - **\$\_SERVER** contient des variables définies par le serveur utilisé ainsi que des informations relatives au script.
  - **\$\_REQUEST** contient toutes les variables envoyées via HTTP GET, HTTP POST et par les cookies HTTP.
  - **\$\_GET** et **\$\_POST** vont être utilisées pour manipuler les informations envoyées via un formulaire HTML.
  - **\$\_FILES** contient des informations sur un fichier téléchargé, comme le type du fichier, sa taille, son nom, etc.
  - **\$\_ENV** contient des informations liées à l'environnement dans lequel s'exécute le script.
  - **\$\_COOKIE** est un tableau associatif qui contient toutes les variables passées via des cookies HTTP.
  - **\$\_SESSION** est un tableau associatif qui contient toutes les variables de session.

### – Création et gestion des cookies en PHP

- Un cookie est un petit fichier texte qui ne peut contenir qu'une quantité limitée de données.
- Les cookies vont être stockés sur les ordinateurs de vos visiteurs. Ainsi, à tout moment, un utilisateur peut lui-même supprimer les cookies de son ordinateur.
- De plus, les cookies vont toujours avoir une durée de vie limitée. On pourra définir la date d'expiration d'un cookie.
- Pour créer un cookie en PHP, nous allons utiliser la fonction **setcookie()**.
- Syntaxe :  
**setcookie( string \$name, string \$value = "", int \$expires\_or\_options = 0, ....)**

```
<?php
    setcookie('user_id', '1234', time()+3600*24);
?>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
    </head>
    <body>
        <?php
            if(isset($_COOKIE['user_id'])){
                echo 'Votre ID de session est le '
                .$_COOKIE['user_id'];
            }
        ?>
    </body>
</html>
```

#### Affichage :



The screenshot shows a web browser at the URL `localhost/m205-backend/seance-1.php`. The page displays the text "Votre ID de session est le 1234". Below the browser window, the Chrome DevTools "Cookies" panel is open, showing a table of cookies for the domain `http://localhost`.

Nom	Valeur	Domain	Path	Expires / Max-Age
user_id	1234	localhost	/m205-backend	2023-01-22T23:44:0...

#### – Définir et utiliser les sessions en PHP

- Une session en PHP correspond à une façon de stocker des données différentes pour chaque utilisateur en utilisant un identifiant de session unique.
- Les identifiants de session vont généralement être envoyés au navigateur via des cookies de session et vont être utilisés pour récupérer les données existantes de la session.
- Pour démarrer une session en PHP, on va utiliser la fonction **`session_start()`**.
- Pour définir et récupérer les valeurs des variables de session, nous allons pouvoir utiliser la variable superglobale **`$_SESSION`**.

```
<?php
    session_start();
    $id_session = session_id();
    $_SESSION['nom'] = 'ALLAOUI';
?>
<!DOCTYPE html>
<html>
<head><meta charset="utf-8"></head>
<body>
    <?php
        echo 'Bonjour ' . $_SESSION['nom'] . '<br/>';
        if(isset($id_session)){
            echo 'Votre ID de session est '
.$id_session . '<br/>';
        }
        if(isset($_COOKIE['PHPSESSID'])){
            echo 'Votre ID de session est '
.$_COOKIE['PHPSESSID'] . PHP_EOL;
        }
    ?>
</body>
</html>
```

#### Affichage :

```
Bonjour ALLAOUI
Votre ID de session est 710me24hn9l4tpiro6o0bb47g1
Votre ID de session est 710me24hn9l4tpiro6o0bb47g1
```

### Les fonctions

- Une fonction est une suite d'instructions qui peut remplir n'importe quelle tâche.
- Il n'y a pas de distinction fonctions / procédures en PHP.

```
<?php
    function ma_fonc ( $param1 , $param2 , ... ) {
        ...
        return ...;
    }
```

```
<?php
    function bonjour() {
        echo " Bonjour ";
    }
    bonjour();
```

#### Affichage :

```
Bonjour BENANI HIND
```

- **Pas de type** pour les paramètres ou la valeur de retour
- Nombre fixe de paramètres
- Le nom est **insensible à la casse** et **ne commence pas par \$**
- Le résultat est renvoyé avec la commande **return** (une seule valeur de retour)
- Passage des paramètres par valeur (par défaut)

```
<?php
function bonjour($prenom, $nom) {
    $chaine = "Bonjour $prenom $nom";
    return $chaine ;
}
echo bonjour("BENANI" , "HIND") ;
```

**Affichage :**

Bonjour BENANI HIND

- Passage par référence : **&\$param**

```
<?php
function bonjour(&$phrase, $prenom, $nom) {
    $phrase = "Bonjour $prenom $nom" ;
}
$chaine = " ";
bonjour($chaine, "BENANI" , "HIND") ;
echo $chaine ;
```

**Affichage :**

Bonjour BENANI HIND

- Les paramètres optionnels sont autorisés : il suffit de leur affecter une valeur par défaut.

```
<?php
function mafonction( $param1 = "inconnu", $param2="" ) {
    echo "param1=$param1 param2=$param2\n";
}
mafonction( "toto", "titi" );
mafonction( "toto" );
mafonction();
```

**Affichage :**

param1=toto param2=titi  
param1=toto param2=  
param1=inconnu param2=

- Par défaut, les variables globales ne sont pas connues à l'intérieur du corps d'une fonction. On peut cependant y accéder à l'aide du mot-clé **global**.

- Les variables utilisées à l'intérieur d'une fonction sont détruites à la fin, sauf si on les définit avec **static** ou **global**

```
<?php
function ma_fonc( ) {
    global $var;
    $var = 54;
}
$var = 0;
echo "Avant ma_fonc $var".PHP_EOL;
ma_fonc( ) ;
echo "Après ma_fonc $var";
```

**Affichage :**

```
Avant ma_fonc 0
Après ma_fonc 54
```

## Les fichiers

- PHP fournit plusieurs fonctions qui permettent de prendre en charge l'accès au système de fichiers du système d'exploitation du serveur.
- Pour lire un fichier entier en PHP, on utilise l'une des fonctions suivantes : `file_get_contents()`, `file()`, `readfile()`
- **Opérations élémentaires sur les fichiers en PHP :**
  - **copy(\$source, \$destination)** Copie d'un fichier,
  - **\$fp=fopen("filemane", \$mode)** Ouvre un fichier et retourne un "id" de fichier,
  - **fclose(\$fp)** Ferme un fichier ouvert,
  - **rename("ancien", "nouveau")** Renomme un fichier,
  - **fwrite(\$fp, \$str)** Ecrit la chaîne de caractères \$str,
  - **fputs(\$fp, \$str)** Correspond à fwrite(),
  - **readfile( "filename")** Lit un fichier et retourne son contenu,
  - **fgets(\$fp, \$maxlength)** Lit une ligne d'un fichier,
  - **fread(\$fp, \$length)** Lit un nombre donné d'octets à partir d'un fichier.

## Inclusion de fichiers

- On inclus un fichier en utilisant les deux instructions **include** ou **require** ou **require\_once** ou **include\_once**
- Les variantes include provoquent des warnings au lieu d'erreurs en cas de problème
- Les variantes **\_once** n'incluent le fichier que si celui n'a pas déjà été inclut



## 2. Programmation orienté objet :

### 2.1. Introduction

- PHP dispose des concepts de POO (Programmation Orientée Objet) au travers des classes.
- Rappelons d'abord qu'un objet possède des attributs et des méthodes, et doit utiliser les mécanismes d'héritage et de polymorphisme.
- **Attribut** : caractéristique d'un objet.
- **Méthode** : action qui s'applique à un objet
- **Héritage** : définition d'un objet comme appartenant à la même famille qu'un autre objet plus général, dont il hérite des attributs et des méthodes.
- **Polymorphisme** : capacité d'un ensemble d'objet à exécuter des méthodes de même nom, mais dont le comportement est propre à chacune des différentes versions.

### 2.2. Classe, Objet, Propriété et Méthode :

- En PHP, on crée une nouvelle classe avec le mot clef **class**. On peut donner n'importe quel nom à une nouvelle classe du moment qu'on n'utilise pas un mot réservé du PHP et que le premier caractère du nom de notre classe soit une lettre ou un underscore.
- Notez qu'on appellera généralement nos fichiers de classe « **maClasse.class.php** » afin de bien les différencier des autres et par convention une nouvelle fois.
- Une propriété est tout simplement une variable définie dans une classe (ou éventuellement ajoutée à un objet après sa création).
- Une méthode est tout simplement une fonction déclarée dans une classe.

#### Fichier classes/utilisateur.class.php :

```
<?php
class Utilisateur {
    public $login;
    public $password;
    public function getLogin(){
        return $this->login;
    }
    public function setLogin($new_login){
        $this->login = $new_login;
    }
    public function setPasseword($new_password){
        $this->password = $new_password;
    }
}
```

?>

#### Fichier index.php

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="UTF-8"></head>
<body>
    <?php
        require_once 'classes/Utilisateur.class.php';
        $user1 = new Utilisateur();
        $user1->login = "ALLAOUI";
        $user1->password = "all123";

        $user2 = new Utilisateur();
        $user2->setLogin("BENANI");
        $user2->setPasseword("ben123");

        echo "Utilisateur 1 : ".$user1->getLogin()."<br/>";
        echo "Utilisateur 2 : ".$user2->getLogin()."<br/>";
    ?>
</body>
</html>
```

#### Affichage :

Utilisateur 1 : ALLAOUI  
Utilisateur 2 : BENANI

- On déclare un constructeur de classe en utilisant la syntaxe **function \_\_construct()**
- Il faut bien comprendre ici que le PHP va rechercher cette méthode lors de la création d'un nouvel objet et va automatiquement l'exécuter si elle est trouvée.

#### Fichier classes/utilisateur.class.php :

```
<?php
    class Utilisateur {
        private $login;
        private $password;

        public function __construct($new_login,
        $new_password){
            $this->login = $new_login;
            $this->password = $new_password;
        }

        public function getLogin(){
            return $this->login;
        }
    }
?>
```



#### Fichier index.php

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="UTF-8"></head>
<body>
    <?php
        require_once 'classes/Utilisateur.class.php';
        $user1 = new Utilisateur("ALLAOUI", "all123");
        $user2 = new Utilisateur("BENANI", "ben123");

        echo "Utilisateur 1 : ".$user1->getLogin()."<br/>";
        echo "Utilisateur 2 : ".$user2->getLogin()."<br/>";
    ?>
</body>
</html>
```

#### Affichage :

Utilisateur 1 : ALLAOUI  
Utilisateur 2 : BENANI

## 2.3. Les méthodes magiques en PHP

- Les méthodes magiques sont des méthodes prédéfinies et toutes préfixées par double sous-tirets (\_\_) dans une classe PHP. Elles sont appelées automatiquement suite à un événement spécial qui peut survenir lors de l'exécution.
- Le constructeur **\_\_construct()** et le destructeur **\_\_destruct()** sont tous les deux des méthodes magiques qui s'exécutent automatiquement suite à un événement.
- **\_\_get(\$att)** connue par le nom de getter est une méthode qui s'exécute automatiquement quand on appelle un attribut inexistant ou inaccessible.

```
<?php
class Utilisateur {
    private $login;
    public function __get($att){
        return 'L\'attribut login est inaccessible.';
    }
}
$objet=new Utilisateur();
echo $objet->login;
```

#### Affichage :

L'attribut login est inaccessible.



- **\_\_set(\$att,\$val)** connue sous le nom de setter est appelée automatiquement quand on essaie de modifier un attribut inexistant ou inaccessible.

```
<?php
class Utilisateur {
    private $login;
    public function __get($att){
        return $this->$att;
    }
    public function __set($att,$val){
        $this->$att=$val;
    }
}
$objet=new Utilisateur();
$objet->login="BENANI";
echo $objet->login;
?>
```

**Affichage :**

BENANI

- **\_\_isset(\$att)** est appelé automatiquement quand on exécute la fonction `isset()` en lui passant en paramètre un attribut inexistant ou inaccessible.
- **\_\_unset(\$att)** est appelé automatiquement quand on essaie de supprimer un attribut inexistant ou inaccessible à l'aide de la fonction `unset()`.
- **\_\_toString()** est appelé automatiquement quand on tente de traiter un objet en tant que chaîne de caractères (à l'aide de la structure `echo` par exemple).

```
<?php
class Utilisateur {
    private $login;
    public function __construct($param){
        $this->login = $param;
    }
    public function __toString(){
        return "Bonjour ".$this->login;
    }
}
$objet=new Utilisateur("BENANI");
echo $objet;
?>
```

**Affichage :**

Bonjour BENANI



## 2.4. L'héritage

- L'héritage est un concept fondamental de la POO. C'est d'ailleurs l'un des plus importants puisqu'il permet de réutiliser le code d'une classe autant de fois que l'on souhaite tout en ayant la liberté d'en modifier certaines parties.
- Pour pouvoir étendre une classe grâce au mot clef **extends**. En utilisant ce mot clef, on va créer une classe « fille » qui va hériter de toutes les propriétés et méthodes de son parent par défaut et qui va pouvoir les manipuler de la même façon (à condition de pouvoir y accéder).

```
<?php
class Mere{
}
class Fille extends Mere{
}
?>
```

Exemple :

```
<?php
class Utilisateur {
    public $login;
    public function __construct($param){
        $this->login = $param;
    }
    public function methode1(){
        return "Bonjour l'utilisateur ".$this->login;
    }
}
class Admin extends Utilisateur{
    public function methode2(){
        return "Bonjour l'admin ".$this->login;
    }
}
$objet=new Admin("BENANI");
echo $objet->methode1().PHP_EOL;
echo $objet->methode2();
?>
```

Affichage :

```
Bonjour l'utilisateur BENANI
Bonjour l'admin BENANI
```

- Le mot clé **parent** désigne la classe dont on a hérité.



```
<?php
class Utilisateur {
    public $login;
    public function __construct($param){
        $this->login = $param;
    }
    public function methode1(){
        return $this->login;
    }
}

class Admin extends Utilisateur{
    public function methode2(){
        $str = parent::methode1();
        return "Bonjour l'admin ".$str;
    }
}
$objet=new Admin("BENANI");
echo $objet->methode2();
?>
```

**Affichage :**

Bonjour l'admin BENANI

- Une classe abstraite et une classe dont on doit impérativement hériter. Autrement dit, on ne peut pas l'instancier directement. Pour ce faire, on définit la classe abstraite à l'aide du mot clé **abstract**.

```
<?php
abstract class A{
}
?>
```

- Une classe finale est une classe dont on ne peut pas hériter. Il faut donc l'instancier directement. Pour définir une classe finale il suffit de la précéder par le mot clé **final**.

```
<?php
final class A{
}
?>
```



## 2.5. Les interfaces et les traits en PHP

- Une interface se comporte comme une classe abstraite dont toutes les méthodes sont abstraites.
- Pour définir une interface, on utilise le mot clé **interface** suivi du nom de celle-ci (avec une première lettre majuscule de préférence, tout comme pour une classe) suivi des accolades qui renfermeront les membres de l'interface comme ceci:

```
<?php
    interface Entretien{
        public function peindre($c);
        public function nettoyer();
    }
?>
```

- **Les traits** ont été intégrés au PHP à partir de sa version 5.4. Ils permettent de minimiser le code en réutilisant des méthodes déjà déclarées sans être obligé d'hériter d'une classe entière.
- Un trait ne peut être ni instancié comme une classe ni implémenté comme une interface. Il s'agit d'un bloc de code qui sera réutilisé par une classe.
- Pour créer un trait on déclare le mot clé **trait** suivi de son nom (capitalisé de préférence) puis des accolades qui contiennent le corps comme ceci :

```
<?php
    trait Notification{
        public function afficher(){
            echo "Alerte signalée !";
        }
    }
?>
```

- Pour se servir du trait Notification dans une classe on procède comme ceci:

```
<?php
    class Operation{
        use Notification;
    }
    $op=new Operation();
    $op->afficher();
?>
```

**Affichage :**

Alerte signalée !



## 2.6. Gestion des exceptions

- Les erreurs en PHP sont gérées à travers le système **Error Reporting** que l'on peut configurer grâce à la directive correspondante dans le fichier **php.ini**. Ce système affiche les messages d'erreur sur l'écran avec leurs différents niveaux de gravité comme les notices, les alertes ou les erreurs fatales:
  - Les **notices** sont des erreurs non critiques. Ils n'arrêtent pas l'exécution du programme (comme la tentative d'affichage d'une variable non initialisée).
  - Les **alertes** sont des erreurs dues à une mauvaise exécution d'une instruction (comme l'inclusion d'un fichier inexistant). Pourtant, le programme poursuit son exécution.
  - Les **erreurs fatales** sont des erreurs critiques qui interrompent l'exécution du programme (suite à une erreur de syntaxe par exemple).
- Cependant, il existe une autre manière de gérer les erreurs. C'est via le système des exceptions.
- La **gestion des exceptions** est apparue en PHP dans sa version 5. C'est un concept très utilisé dans d'autres langages de programmation (orientés objet) comme Java, Python ou encore Javascript...

### Bloc try catch

- Le principe est simple: on place le code qui peut générer une erreur éventuelle dans le bloc **try** et on prévoit un traitement alternatif dans le bloc **catch**. Si le bloc try ne génère aucune erreur, alors le bloc catch sera ignoré, sinon il sera exécuté pour rattraper l'erreur du bloc try.

```
<?php
    $a=10;
    $b=0;
    try{
        $c=$a/$b;
        echo $c;
    } catch(Exception $e) {
        echo $e->getMessage();
    }
?>
```

**Affichage :**

PHP Warning: Division by zero

- Une exception peut être lancée grâce au mot clé **throw** de n'importe où dans le programme. Une exception lancée représente une instance de la classe **Exception** prédéfinie en PHP5 et plus. Il ne faut pas oublier de renseigner, au moins, le message d'erreur ou le code de celui-ci au constructeur.



```
<?php
    $a=10;
    $b=0;
    try{
        if($b==0)
            throw new Exception("Le dénominateur ne doit pas
            être null.");
        $c=$a/$b;
        echo $c;
    } catch(Exception $e) {
        echo $e->getMessage();
    }
?>
```

**Affichage :**

Le dénominateur ne doit pas être null.

- La classe Exception contient 6 méthodes dont les plus importants sont:
  - **getMessage():** retourne le message d'erreur passé au constructeur lors du lancement de l'exception.
  - **getCode():** retourne le code d'erreur passé aussi au constructeur.
  - **getLine():** retourne le numéro de la ligne où l'exception a été lancée.
  - **getFile():** retourne le nom du document où les choses se passent.



### 3. Développer une application avec PHP et MySQL

#### 3.1. Manipuler des données dans des bases MySQL avec PDO

##### Se connecter à une base de données MySQL en PHP

- Pour pouvoir manipuler nos bases de données MySQL en PHP (sans passer par phpMyAdmin), nous allons déjà devoir nous connecter à MySQL.
- Pour cela, le PHP met à notre disposition deux API (Application Programming Interface) :
  - L'extension **MySQLi** ;
  - L'extension **PDO** (PHP Data Objects).
- Chacune de ces deux API possède des forces différentes et comme vous vous en doutez elles ne sont pas forcément interchangeables.
- Il existe notamment une différence notable entre ces deux API : l'extension MySQLi ne va fonctionner qu'avec les bases de données MySQL tandis que PDO va fonctionner avec 12 systèmes de bases de données différents.

##### Connexion au serveur avec MySQLi

- Pour se connecter au serveur et accéder à nos bases de données MySQL en MySQLi orienté objet, nous allons avoir besoin de trois choses : le nom du serveur ainsi qu'un nom d'utilisateur (avec des privilèges de type administrateur) et son mot de passe.

```
<?php
    $servername = 'localhost';
    $username = 'root';
    $password = '';
    //On établit la connexion
    $conn = new mysqli($servername, $username, $password);
    //On vérifie la connexion
    if(!$conn){
        die('Erreur : ' . mysqli_connect_error());
    }
    echo 'Connexion réussie';
    $conn->close();
?>
```

##### Connexion au serveur avec PDO

- Pour se connecter en utilisant PDO, nous allons devoir instancier la classe PDO en passant au constructeur la source de la base de données (serveur + nom de la base de données) ainsi qu'un nom d'utilisateur et un mot de passe.





```
<?php
    $servername = 'localhost';
    $username = 'root';
    $password = '';
    try{
        $conn = new PDO("mysql:host=$servername;dbname=bddtest",
    $username, $password);
        echo 'Connexion réussie';
    } catch(PDOException $e) {
        echo "Erreur : " . $e->getMessage();
    }
    $conn = null;
?>
```

### Exécuter une requête

- **PDO::prepare** : Prépare une requête à l'exécution et retourne un objet
- **PDO::exec** : Exécute une requête SQL et retourne le nombre de lignes affectées

```
$stmt = $conn->prepare("SELECT * FROM utilisateurs WHERE id=?");
$stmt->bindParam(1,$id);
$stmt->execute();
```

- Ensuite, on traite les données avec un foreach (exemple avec affichage des colonnes nom :

```
$res = $stmt->fetchAll();
foreach ( $res as $row ) {
    echo $row['nom'];
}
```

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

```
$sql = 'SELECT nom, prenom FROM utilisateurs ORDER BY name';
foreach ($conn->query($sql) as $row) {
    print $row['nom'] . "\t";
    print $row['prenom'] . "\n";
}
```



### 3.2. Gestion des formulaires avec PHP

#### Rappels sur les formulaires HTML

- Pour créer un formulaire, nous allons utiliser l'élément HTML form. Cet élément **form** va avoir besoin de deux attributs pour fonctionner normalement : les attributs **method** et **action**.
- L'attribut **method** va indiquer comment doivent être envoyées les données saisies par l'utilisateur. Cet attribut peut prendre deux valeurs : **get** et **post**.

#### Récupérer et manipuler les données des formulaires HTML en PHP

- La première chose à comprendre ici est que toutes les données du formulaire vont être envoyées et être accessibles dans le script PHP mentionné en valeur de l'attribut **action**, et cela quelle que soit la méthode d'envoi choisie (**post** ou **get**).
- En effet, le PHP possède dans son langage deux variables superglobales **\$\_GET** et **\$\_POST** qui sont des variables tableaux et dont le rôle va justement être de stocker les données envoyées via des formulaires.
- Plus précisément, la superglobale **\$\_GET** va stocker les données envoyées via la méthode **get** et la variable **\$\_POST** va stocker les données envoyées via la méthode **post**.
- Les valeurs vont être stockées sous forme d'un tableau associatif c'est-à-dire sous la forme **clef => valeur** où la **clef** va correspondre à la valeur de l'attribut **name** d'un champ de formulaire et la **valeur** va correspondre à ce qui a été rempli (ou coché, ou sélectionné) par l'utilisateur pour le champ en question.
- A noter : On va également pouvoir utiliser la variable superglobale **\$\_REQUEST** pour accéder aux données d'un formulaire sans se soucier de la méthode d'envoi.



## 4. Découvrir les notions fondamentales des frameworks PHP

### 4.1. Patrons de conception (Design Patterns) :

Les patrons de conception (design patterns) sont des solutions classiques à des problèmes récurrents de la conception de logiciels. Chaque patron est une sorte de plan ou de schéma que vous pouvez personnaliser afin de résoudre un problème récurrent dans votre code.

Il est habituel de regrouper ces modèles communs dans trois grandes catégories :

- les modèles de création
- les modèles de structuration
- les modèles de comportement

Exemples :

#### Fabrique (Factory)

La fabrique permet de créer un objet dont le type dépend du contexte : cet objet fait partie d'un ensemble de sous-classes. L'objet retourné par la fabrique est donc toujours du type de la classe mère mais grâce au polymorphisme les traitements exécutés sont ceux de l'instance créée.

Ce motif de conception est utilisé lorsqu'à l'exécution il est nécessaire de déterminer dynamiquement quel objet d'un ensemble de sous-classes doit être instancié.

L'utilisation d'une fabrique permet de rendre l'instanciation d'objets plus flexible que l'utilisation de l'opérateur d'instanciation new.

Ce design pattern peut être implémenté sous plusieurs formes dont les deux principales sont :

- Déclarer la fabrique abstraite et laisser une de ses sous-classes créer l'objet
- Déclarer une fabrique dont la méthode de création de l'objet attend les données nécessaires pour déterminer le type de l'objet à instancier

#### Singleton (Singleton)

Ce patron de conception de création propose de n'avoir qu'une seule et unique instance d'une classe dans une application.

Le Singleton est fréquemment utilisé dans les applications car il n'est pas rare de ne vouloir qu'une seule instance pour certaines fonctionnalités (pool, cache, ...). Ce modèle est aussi particulièrement utile pour le développement d'objets de type gestionnaire. En effet ce type d'objet doit être unique car il gère d'autres objets par exemple un gestionnaire de logs.

La mise en œuvre du design pattern Singleton doit :

- assurer qu'il n'existe qu'une seule instance de la classe
- fournir un moyen d'obtenir cette instance unique



## Adaptateur

Ce patron de conception structurel permet de faire collaborer des objets ayant des interfaces normalement incompatibles. Autrement dit, Il permet de convertir l'interface d'une classe en une autre interface que le client attend. Adaptateur fait fonctionner un ensemble de classes qui n'auraient pas pu fonctionner sans lui, à cause d'une incompatibilité d'interfaces.

## Composite

Ce patron de conception structurel permet d'agencer les objets dans des arborescences afin de pouvoir traiter celles-ci comme des objets individuels.

Il permet en particulier de créer des objets complexes en reliant différents objets selon une structure en arbre. Ce patron impose que les différents objets aient une même interface, ce qui rend uniformes les manipulations de la structure. Par exemple dans un traitement de texte, les mots sont placés dans des paragraphes disposés dans des colonnes dans des pages ; pour manipuler l'ensemble, une classe composite implémente une interface. Cette interface est héritée par les objets qui représentent les textes, les paragraphes, les colonnes et les pages

## Observateur

Ce patron de conception comportemental qui permet de mettre en place un mécanisme de souscription pour envoyer des notifications à plusieurs objets, au sujet d'événements concernant les objets qu'ils observent.

Dans ce patron, un objet le sujet tient une liste des objets dépendants des observateurs qui seront avertis des modifications apportées au sujet. Quand une modification est apportée, le sujet émet un message aux différents observateurs. Le message peut contenir une description détaillée du changement.

## 4.2. L'architecture MVC :

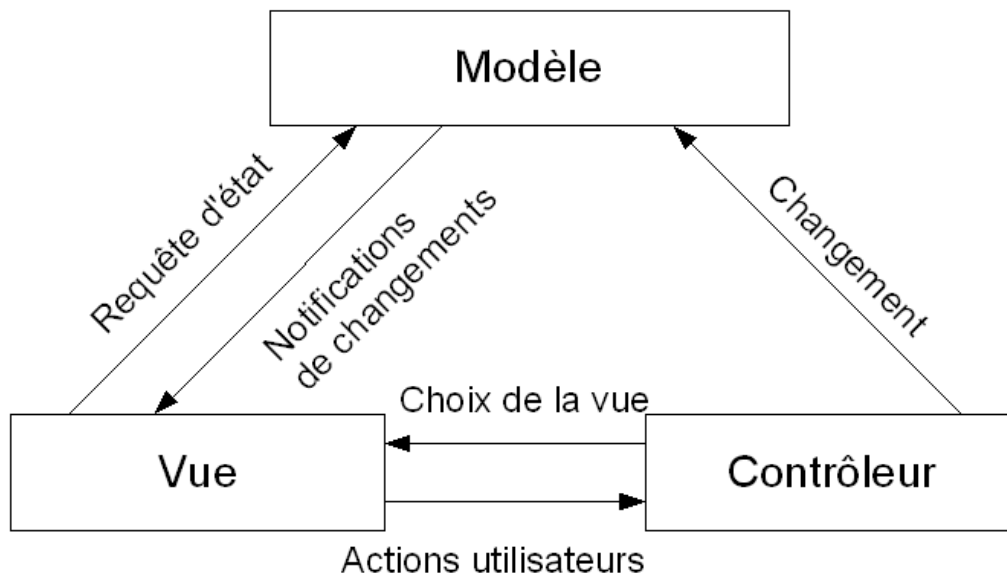
Le design pattern Modèle-Vue-Contrôleur (MVC) est un pattern architectural qui sépare les données (le modèle), l'interface homme-machine (la vue) et la logique de contrôle (le contrôleur).

Ce modèle de conception impose donc une séparation en trois couches :

- **le modèle** : il représente les données de l'application. Il définit aussi l'interaction avec la base de données et le traitement de ces données ;
- **la vue** : elle représente l'interface utilisateur, ce avec quoi il interagit. Elle n'effectue aucun traitement, elle se contente d'afficher les données que lui fournit le modèle. Il peut tout à fait y avoir plusieurs vues qui présentent les données d'un même modèle ;
- **le contrôleur** : il gère l'interface entre le modèle et le client. Il va interpréter la requête de ce dernier pour lui envoyer la vue correspondante. Il effectue la synchronisation entre le modèle et les vues.

La synchronisation entre la vue et le modèle se passe avec le pattern **Observer**. Il permet de générer des événements lors d'une modification du modèle et d'indiquer à la vue qu'il faut se mettre à jour.

Voici un schéma des interactions entre les différentes couches :





### 4.3. Frameworks PHP :

Un framework PHP est une plate-forme permettant de créer des applications web en PHP. Les frameworks PHP fournissent des bibliothèques de code pour les fonctions les plus courantes, ce qui réduit la quantité de code original à écrire.

Il existe de nombreuses bonnes raisons d'utiliser des frameworks PHP plutôt que de coder à partir de zéro.

- **Un développement plus rapide**
- **Moins de code à écrire**
- **Bibliothèques pour les tâches communes**
- **Suivre les bonnes pratiques de codage**
- **Plus sécurisé que d'écrire vos propres applications**

Pour utiliser les frameworks PHP

- **Maitriser le langage PHP et la programmation orienté objet.**
- **Maitriser les bases de données et le langage SQL.**
- **Comprendre le fonctionnement des serveurs web comme Apache et Nginx**
- **Comprendre l'architecture MVC**

Un framework PHP doit répondre à vos exigences techniques pour un projet. La plupart des frameworks disposent d'une version minimale de PHP et de certaines extensions PHP avec lesquelles ils fonctionnent. Assurez-vous que votre framework supporte la ou les bases de données de votre choix et que vous pouvez utiliser le framework avec le serveur web sur lequel vous souhaitez le déployer.

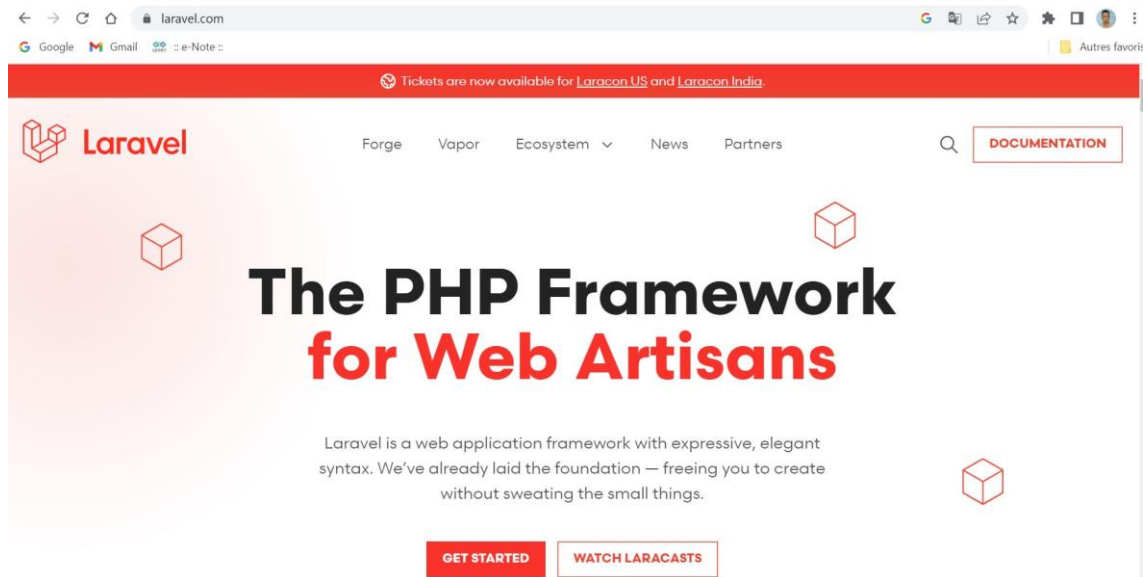
Choisissez un framework avec un bon équilibre de fonctionnalités. Un framework riche en fonctionnalités peut être une aubaine pour certains projets. Par contre, si vous n'avez pas besoin de beaucoup de fonctionnalités, choisissez un framework dépouillé et minimal.

Les frameworks PHP fournissent un code bien organisé et réutilisable. Il y a de dizaine de frameworks PHP de nos jours, ce cours est un bref aperçu les deux meilleurs frameworks PHP à utiliser pour le développement web :

- **Laravel**
- **Symfony**



## Laravel



Le framework Laravel est un framework PHP open-source qui utilise son propre langage de templating, Blade. Il suit le modèle MVC (Model-View-Controller) permettant aux développeurs de gérer les tâches de codage de manière simple, facile et bien documentée. Il utilise l'ORM Eloquent, un mappeur objet-relationnel, permettant aux développeurs de définir des modèles et des relations en PHP qui sont traduits et exécutés en SQL.

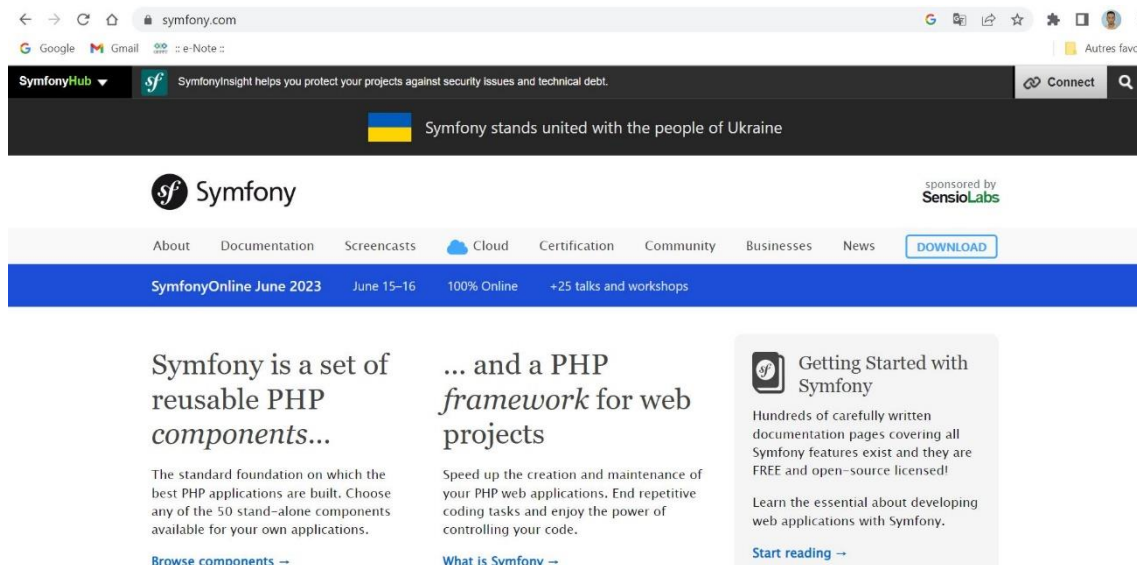
Avec une excellente documentation et une grande communauté, Laravel est l'une des meilleures options pour les débutants. Sa syntaxe facile à apprendre en fait le framework PHP le plus populaire parmi les développeurs. Il permet également le déploiement sur les fournisseurs de cloud les plus populaires. Laravel comprend la plupart des fonctionnalités qui peuvent être nécessaires pour les applications avancées. Les utilisateurs peuvent également installer une variété de paquets et d'applications s'ils ont besoin de plus de fonctionnalités.

Quelques caractéristiques de Laravel :

- Moteur de routage simple et rapide.
- Puissant conteneur d'injection de dépendances.
- Traitement robuste des tâches en arrière-plan.
- ORM de base de données expressif et intuitif.
- Migrations de schémas agnostiques de bases de données.
- Diffusion d'événements en temps réel.



## Symfony



The screenshot shows the Symfony website homepage. At the top, there's a navigation bar with links like 'About', 'Documentation', 'Screencasts', 'Cloud', 'Certification', 'Community', 'Businesses', and 'News'. A 'DOWNLOAD' button is visible. Below the navigation bar, there's a blue banner for 'SymfonyOnline June 2023' with dates 'June 15-16', '100% Online', and '+25 talks and workshops'. The main content area features three columns: 'Symfony is a set of reusable PHP components...', '... and a PHP framework for web projects', and 'Getting Started with Symfony'. Each column has a brief description and a link to explore further.

Le framework Symfony possède également une architecture MVC. Il s'agit d'une plateforme très fiable qui permet aux développeurs d'ajouter des modules supplémentaires. Le système de composants modulaires de Symfony offre une flexibilité supplémentaire, permettant aux développeurs de choisir les composants dont ils ont besoin pour le projet. Il prend également en charge la plupart des bases de données, notamment Drizzle, MySQL, Oracle, PostgreSQL, SAP Sybase SQL Anywhere, SQLite et SQLServer.

Sa base de code bien conçue permet aux développeurs d'écrire des codes en moins de lignes, ce qui rend les applications plus rapides et plus fluides. Symfony est idéal pour les projets à grande échelle et il est également facile à installer et à configurer sur la plupart des plateformes.





#### 4.4. CMS PHP :

Un système de gestion de contenu (content management system ou CMS) est un programme permettant de créer un site internet, un blogue ou encore un site de vente en ligne.

Au lieu de construire votre propre système pour créer des pages Web, stocker des images et d'autres fonctions, le système de gestion de contenu s'occupe de tout ce qui concerne l'infrastructure de base pour vous afin que vous puissiez vous concentrer sur des parties plus orientées vers l'avenir de votre site.

Les fonctionnalités varient d'un système de gestion de contenu à un autre. Idéalement, un SGC devrait offrir ces différentes fonctionnalités :

- **Contrôle des versions**
- **Gestion des utilisateurs et des droits**
- **Chaîne de validation (workflow)**
- **Support des métadonnées**
- **Indexation et recherche**
- **Intégration de sources de données externes**

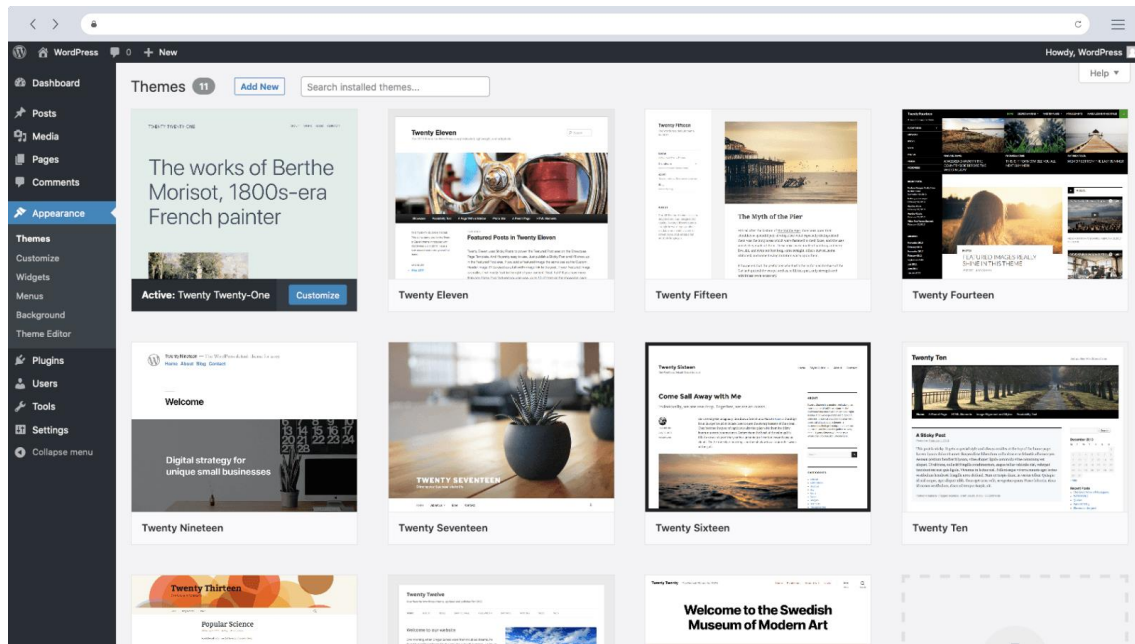
Les systèmes open source WordPress, Joomla et Drupal sont devenus des solutions standards pour le fonctionnement d'un site Internet professionnel.

CMS	% Tous les sites web	% Marché des CMS
WordPress	43	65
Joomla	3	5
Drupal	2	3

## WordPress

WordPress est l'un des systèmes de gestion de contenu / blog les plus populaires sur le marché à l'heure actuelle. WordPress est un système simple et facile à utiliser, supportant à la fois les blogs personnels ou même les sites web entièrement fonctionnels. Il est par ailleurs extrêmement simple de créer des sauvegardes de votre site web sous WP !

Les plugins et thèmes pour WordPress sont nombreux, ce qui permet de construire des sites web à la fois lourds et compliqués.



### Informations techniques

- Plus de 58 000 plugins et de nombreux thèmes gratuits.
- Installation en 5 minutes avec l'aide d'un assistant
- URL adaptées aux moteurs de recherche
- Édition et outils de gestion pour les solutions mobiles

### Configuration système requise pour WordPress 6

- **Serveur Web** : tout serveur avec support PHP et MySQL / MariaDB (recommandation : Apache ou NGINX)
- **Middleware** : PHP 7.4 +
- **Système de gestion de bases de données** : MySQL 5.7 + / MariaDB 10.3 +
- **Autres recommandations** : support HTTPS



## Joomla

Joomla est un CMS / système de gestion de contenu très populaire. C'est un système open source, ce qui signifie que vous n'avez rien à payer pour l'utiliser. Si vous êtes prêt à payer un peu d'argent, alors il y a beaucoup de thèmes professionnels et de plugins disponibles à l'achat. Joomla peut être utilisé pour construire un large éventail de différents types de sites web.

### Informations techniques

- Actuellement plus de 6 000 extensions disponibles
- Gestion du contenu orienté objet
- Communauté francophone très forte.

### Configuration système requise pour le CMS Joomla 4

- **Serveur Web** : serveur HTTP Apache 2.4+ ou NGINX 1.10+
- **Middleware** : PHP 7.3+
- **Système de gestion de bases de données** : MySQL 5.6+ ou PostgreSQL 11.0+

## Drupal

Drupal est un système incroyablement puissant et complet. Il est construit à l'aide de modules, qui peuvent être facilement adaptés à tous vos besoins personnels. Il existe également une large sélection de plugins et d'addons disponibles pour Drupal.

Drupal prendra un peu plus de temps à maîtriser, comparé à d'autres systèmes de gestion de contenu. Pour cette raison, ce n'est pas le meilleur choix pour les développeurs de sites web et les blogueurs à la recherche d'une solution rapide et facile.

### Informations techniques

- Programme très flexible avec construction modulable
- Installation de base légère mais dotée de plus de 46 000 modules d'extension
- Accent mis sur l'édition sociale et les projets communautaires
- Plus de 1300 distributions comme solution complète pour des applications spécifiques

### Configuration système requise pour Drupal 10

- **Serveur Web** : Apache 2.4.7+, NGINX 0.7.x+, MS IIS ou tout autre serveur Web avec support PHP
- **Middleware** : PHP 7.3+
- **Système de gestion de bases de données** : MySQL 5.7.8 +, PostgreSQL 10.0+, SQLite 3.26+

## 5. Préparer l'environnement de Laravel

### 5.1. Installation sur Windows :

Installer composer : <https://getcomposer.org/>

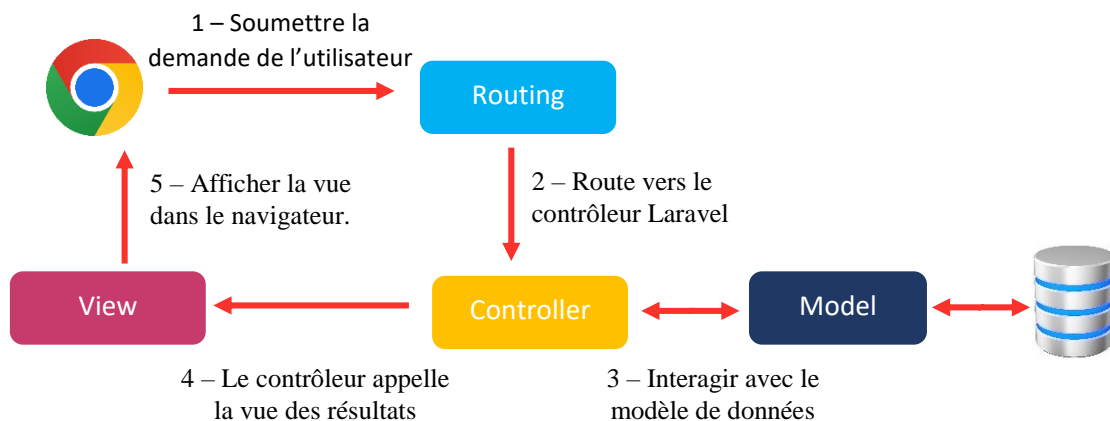
Vérifier la configuration du serveur (WampServer) :

- PHP >= 7.3
- Extension PDO,
- Extension Mbstring,
- Extension OpenSSL,
- Extension Tokenizer,
- Extension XML.,
- Extension BCMath,
- Extension CType,
- Extension JSON
- Extension Fileinfo

### 5.2. Architecture du Framework Laravel :

Laravel se base effectivement sur le patron de conception **MVC**, c'est-à-dire **modèle-vue-contrôleur**.

La séparation des composants d'une application en ces trois catégories permet une clarté de l'architecture des dossiers et simplifie grandement la tâche aux développeurs.



#### Les Modèles (Models)

Un Modèle est en réalité un fichier PHP qui ne fait que gérer les échanges avec la base de données. Lorsque nous avons besoin de lire ou écrire dans la base de données, nous faisons appel au Modèle.

- Le modèle est la couche représentant les données. On l'appellera parfois logique métier.



- Le modèle consiste en une série de classes. Si les données sont tirées de la BD, chacune des classes représentera une table.
- Parmi les fonctionnalités codées dans le modèle, on retrouve les relations entre les tables, les accesseurs et modificateurs, les champs calculés, etc.

### **Les Vues (Views)**

- La vue est constituée de balises HTML qui représentent ce qui sera affiché à l'écran, c'est une interface utilisateur.
- En plus des balises HTML, la vue peut utiliser des directives et instructions prévues par le moteur d'affichage afin d'effectuer différentes opérations, comme par exemple tester une condition ou encore boucler dans les données fournies par le modèle.
- La vue pourra faire appel à des ressources externes, comme des feuilles de style, des fichiers JavaScript, des images, etc.
- Sous Laravel, le moteur d'affichage s'appelle Blade.

### **Les Contrôleurs (Controllers)**

- Véritable tour de contrôle de notre application, le contrôleur a pour fonction de faire l'interface entre les modèles et les vues. Il est chargé de demander les données par l'intermédiaire des modèles, de traiter ces données et de les transmettre aux vues, prêtes à être utilisées.

### **Le routing**

- Bien qu'indépendant de l'architecture MVC, le routing fait partie intégrante de tous les Frameworks PHP.
- Dans une architecture classique, nous pointons vers des fichiers :
  - <http://monsite.fr/inscription.php>
  - <http://monsite.fr/login.php>
- Dans une architecture MVC, nous allons pointer vers des dossiers virtuels appelés routes
  - <http://monsite.fr/user/inscription>
  - <http://monsite.fr/user/login>
- Cette architecture offre de nombreux avantages :
  - Protection des fichiers, ceux-ci n'étant plus affichés par le navigateur
  - Des URLs plus simples à mémoriser pour les utilisateurs
  - Amélioration du référencement si les routes contiennent des mots-clés contenus dans la page correspondante

### 5.3. Création d'un premier projet :

#### Créer le projet via Composer

Pour créer un projet Laravel. Ouvrez votre terminal (invite de commande) et tapez la ligne suivante :

```
composer create-project laravel/laravel mon-premier-projet
```

#### Créer le projet via Laravel Installer

Une autre alternative est de créer un projet en utilisant l'installateur de Laravel.

Premièrement. Installer l'installateur de Laravel à partir de la commande suivante :

```
composer global require laravel/installer
```

Maintenant, vous pouvez créer votre projet en tapant la commande suivante en plaçant le terminal dans votre répertoire de travail

```
laravel new mon-premier-projet
```

#### Lancer l'application

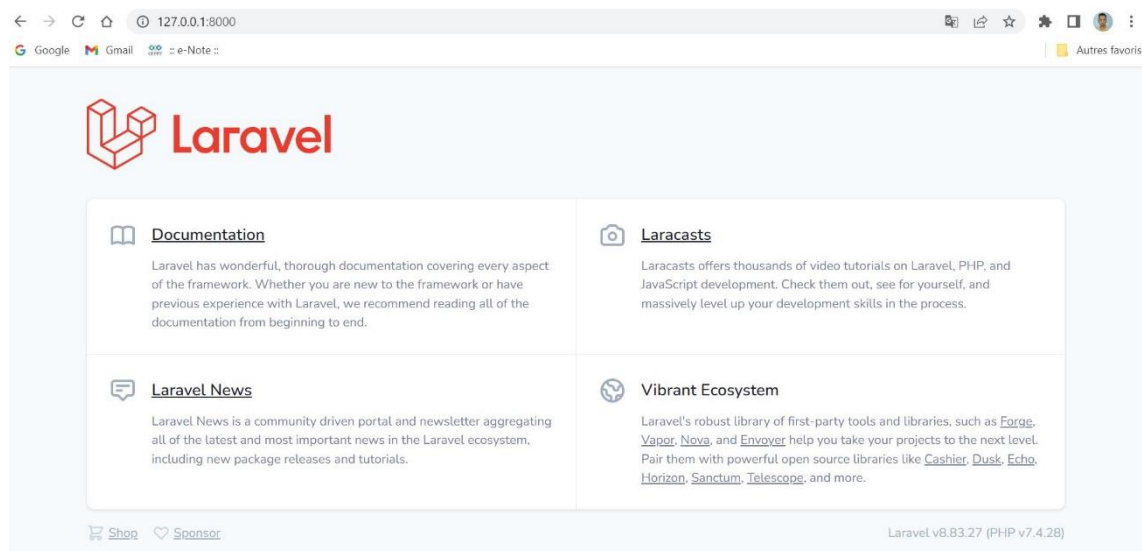
Placez le terminal dans le dossier racine de votre projet :

```
cd mon-premier-projet
```

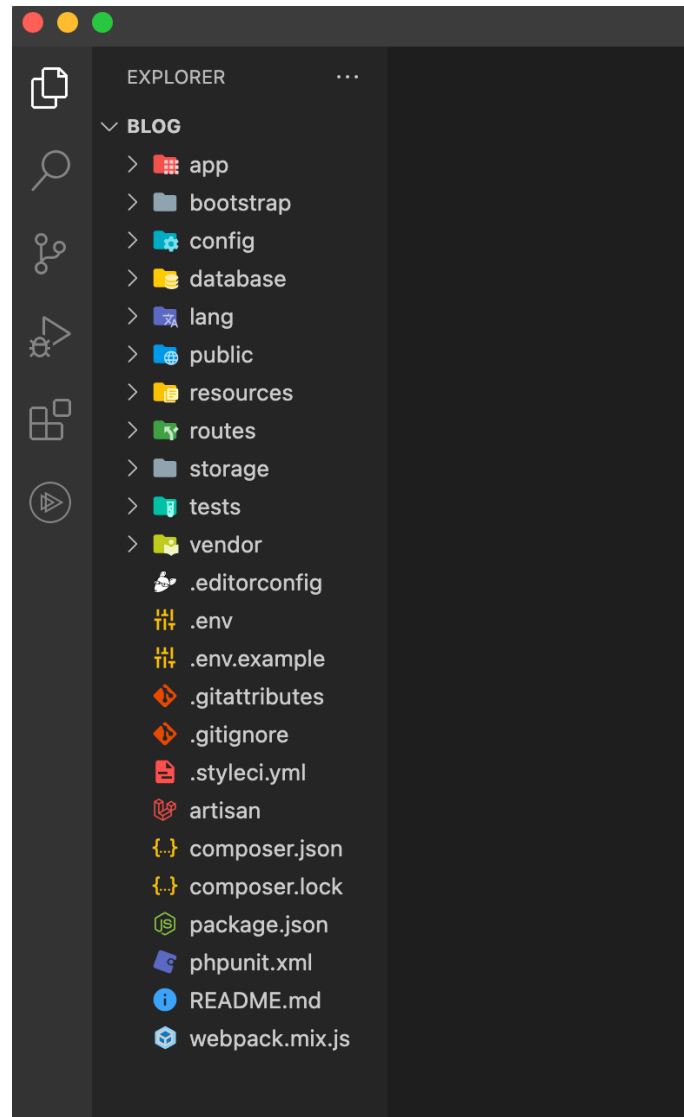
Puis taper la commande suivante pour démarrer le serveur de l'application :

```
php artisan serve
```

Ouvrez le navigateur et allez sur <http://localhost:8000/>, et vous obtiendrez le résultat suivant :



## 5.4. Structure de projet :



### Répertoires racines

#### app

Le répertoire app contient le code principal de votre application. Nous explorerons ce répertoire plus en détail bientôt ; cependant, presque toutes les classes de votre application se trouveront dans ce répertoire.

#### config

Le répertoire config, comme son nom l'indique, contient tous les fichiers de configuration de votre application. C'est une bonne idée de lire tous ces fichiers et de vous familiariser avec toutes les options qui s'offrent à vous.

#### database

Le répertoire database contient vos migrations de base de données, les données factices de vos modèles. Si vous le souhaitez, vous pouvez également utiliser ce répertoire pour contenir une base de données SQLite. Il est donc le répertoire qui gère la structure de votre base de données.



## public

Le répertoire public contient le fichier index.php, qui est le point d'entrée pour toutes les demandes entrant dans votre application et configure le chargement automatique. Ce répertoire héberge également vos fichiers statiques tels que les images, JavaScript et CSS.

## resources

Le répertoire resources contient vos **vues** ainsi que vos fichiers bruts non compilés tels que CSS, SASS ou JavaScript.

## routes

Le répertoire routes contient toutes les définitions des URLs pour votre application. Par défaut, 4 fichiers de route sont inclus avec Laravel : web.php, api.php, console.php et channels.php. Dans ce tutoriel, vous avez besoin seulement de web.php.

## storage

Le répertoire storage contient vos logs, modèles de vues (blades) compilés, sessions basées sur des fichiers, fichiers caches et autres fichiers générés par le framework. Ce répertoire est divisé en répertoires app, framework et logs. Le répertoire app peut être utilisé pour stocker tous les fichiers générés par votre application. Le répertoire framework est utilisé pour stocker les fichiers et les caches générés par le framework. Enfin, le répertoire logs contient les fichiers journaux de votre application.

Le répertoire **storage/app/public** peut être utilisé pour stocker des fichiers générés par l'utilisateur, tels que des avatars de profil, les images des vos publications, qui doivent être accessibles au public.

## tests

Le répertoire tests contient vos tests automatisés. Des exemples de tests unitaires PHPUnit et de tests de fonctionnalités sont fournis prêts à l'emploi. Chaque classe de test doit être suffixée par le mot Test. Vous pouvez exécuter vos tests à l'aide des commandes **php unit** ou **php vendor/bin/phpunit**. Ou, si vous souhaitez une représentation plus détaillée et plus belle de vos résultats de test, vous pouvez exécuter vos tests à l'aide de la commande Artisan **php artisan test**.

## Répertoires de app

### http

Le répertoire Http contient vos contrôleurs, middleware et requests. La quasi-totalité de la logique de gestion des requêtes entrant dans votre application sera placée dans ce répertoire. Les contrôleurs (app/http/controllers) sont les intermédiaires entre nos vues et les modèles.





## Models

Le répertoire Models contient toutes vos classes de modèle Eloquent. **L'ORM Eloquent** inclus avec Laravel fournit une belle et simple implémentation *ActiveRecord* pour travailler avec votre base de données. Chaque table de base de données a un "modèle" correspondant qui est utilisé pour interagir avec cette table. Les modèles vous permettent d'interroger des données dans vos tables, ainsi que d'insérer de nouveaux enregistrements dans la table.

## Les fichiers de l'application

À la racine de notre projet, Composer a également ajouté de nombreux fichiers.

La plupart sont des fichiers de configuration pour des outils externes (Git, Composer, NPM, PHPUnit...) et nous ne les utiliserons pas avant d'avoir découvert ces outils.

### Fichier .env :

- Les fichiers .env contient les mots de passe de vos services ainsi que toutes les données sensibles de votre application (mot de passe de base de données, adresse de la base de données...).
- Ce fichier ne doit jamais être partagé. Afin de connaître les informations à renseigner, il existe un fichier .env.example qui contient uniquement des valeurs d'exemple.

### Fichier artisan :

- Le fichier artisan permet de lancer des commandes comme **php artisan serve**. Ces commandes vont nous permettre de faire beaucoup de choses avec Laravel et nous verrons au fur et à mesure des tutoriels les différentes commandes existantes.

## 5.5. Artisan :

Artisan est l'interface en ligne de commande de Laravel. Cet outil permet de lancer des commandes à destination de Laravel pour créer un contrôleur, une migration, vider le cache, traiter les files d'attente, etc. Plus généralement, Artisan permet de gagner du temps dans la création de fichiers ou dans l'exécution de scripts.

Pour utiliser Artisan, il faut ouvrir un terminal de ligne de commande et taper une commande commençant par **php artisan** depuis la racine de l'application.

Si vous affichez la liste des commandes artisan, vous verrez que votre commande est déjà connue :

```
php artisan list
```

```
$ php artisan list
Laravel Framework 8.83.27

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display help for the given command. When no command is g
  -q, --quiet            Do not output any message
  -V, --version          Display this application version
  --ansi|--no-ansi      Force (or disable --no-ansi) ANSI output
  -n, --no-interaction  Do not ask any interactive question
  --env[=ENV]           The environment the command should run under
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output,
                        2 for more verbose output and 3 for debug

Available commands:
  clear-compiled      Remove the compiled class file
  completion          Dump the shell completion script
  db                  Start a new database CLI session
  down                Put the application into maintenance / demo mode
  env                 Display the current framework environment
  help                Display help for a command
  inspire             Display an inspiring quote
  list                List commands
  migrate             Run the database migrations
  optimize            Cache the framework bootstrap files
  serve               Serve the application on the PHP development server
  test                Run the application tests
  tink                Interact with your application
  up                  Bring the application out of maintenance mode

auth
  auth:clear-resets   Flush expired password reset tokens

cache
  cache:clear          Flush the application cache
  cache:forget         Remove an item from the cache
  cache:table          Create a migration for the cache database table

config
  config:cache         Create a cache file for faster configuration loading
  config:clear         Remove the configuration cache file

db
  db:seed              Seed the database with records
  db:wipe               Drop all tables, views, and types

event
  event:cache          Discover and cache the application's events and listeners
  event:clear          Clear all cached events and listeners
  event:generate        Generate the missing events and listeners based on regi
  event:list           List the application's events and listeners

key
  key:generate         Set the application key

make
  make:cast            Create a new custom Eloquent cast class
  make:channel          Create a new channel class
  make:command          Create a new Artisan command
  make:component        Create a new view component class
  make:controller       Create a new controller class
  make:event            Create a new event class
  make:exception        Create a new custom exception class
  make:factory          Create a new model factory
  make:job              Create a new job class
  make:listener         Create a new event listener class
  make:mail             Create a new email class
  make:middleware       Create a new middleware class
  make:migration        Create a new migration file
  make:model            Create a new Eloquent model class
  make:notification     Create a new notification class
  make:observer         Create a new observer class
  make:policy           Create a new policy class
  make:provider         Create a new service provider class
  make:request          Create a new form request class
  make:resource         Create a new resource
  make:rule             Create a new validation rule
  make:seeder           Create a new seeder class
  make:test            Create a new test class

migrate
  migrate:fresh         Drop all tables and re-run all migrations
  migrate:install       Create the migration repository
  migrate:refresh        Reset and re-run all migrations
  migrate:reset         Rollback all database migrations
  migrate:rollback      Rollback the last database migration
  migrate:status        Show the status of each migration

model
  model:prune          Prune models that are no longer needed

notifications
  notifications:table   Create a migration for the notifications table

optimize
  optimize:clear        Remove the cached bootstrap files

package
  package:discover      Rebuild the cached package manifest
```



Démarrer le serveur de l'application :

**php artisan serve**

Lister toutes les routes définies :

**php artisan route:list**

Créer le squelette pour créer un modèle

**php artisan make:model Actualite**

Créer la migration

**php artisan migrate**

Créer un contrôleur

**php artisan make:controller ActualiteController**

Afficher l'environnement actuel

**php artisan env**

Vider tous les caches de l'application

**php artisan cache:clear**

Vider les caches pour les fichiers de configuration

**php artisan config:clear**

Créer une nouvelle commande

**php artisan make:command nomDeLaCommande**

Mettre le site en mode maintenance

**php artisan down**

Mettre le site en ligne

**php artisan up**

Pour générer une clef d'application

**php artisan key:generate**

Pour supprimer toutes les migrations

**php artisan migrate:fresh**



## 6. Créer des routes

### 6.1. Prise en main :

#### Définir une route

Laravel fournit un système de routes simple. Déclarer une route permet de lier une URI (identifiant de ressource uniforme, autrement dit la partie de l'adresse qui suit le nom de domaine) à un code à exécuter.

La liste des routes se trouve dans le fichier **routes/web.php** d'un nouveau projet Laravel. Il faudra alimenter ce fichier au fur et à mesure de l'ajout de nouvelles pages sur un site. C'est donc en quelque sorte la carte du site, le fichier qui regroupe l'ensemble des routes.

Par défaut, Laravel s'installe avec un dossier de routes, pour gérer divers besoins des itinéraires, dans son répertoire racine, ce dossier contient quatre fichiers, `api.php` (utilisé pour gérer les routes de l'API), `channels.php`, `console.php` et `web.php` (gère les routes normales).

Pour créer une route, il faut ainsi appeler la classe Route avec la méthode HTTP souhaitée (get par exemple). Indiquez à cette méthode l'URI concernée et le retour à afficher pour le visiteur comme dans l'exemple ci-dessous.

#### Fichier routes/web.php

```
use Illuminate\Support\Facades\Route;
Route::get('/', function () {
    return 'Bienvenue sur le site !';
});
```

#### Les méthodes HTTP

Le HTTP (Hypertext Transfer Protocol) est un protocole de communication entre un client et un serveur. Le client demande une ressource au serveur en envoyant une requête et le serveur réagit en envoyant une réponse, en général une page Html.

La liste des méthodes disponibles pour le routeur (accessibles via la façade Route) correspond simplement à la liste des méthodes HTTP :

- **GET** : c'est la plus courante, on demande une ressource qui ne change jamais, on peut mémoriser la requête, on est sûr d'obtenir toujours la même ressource,

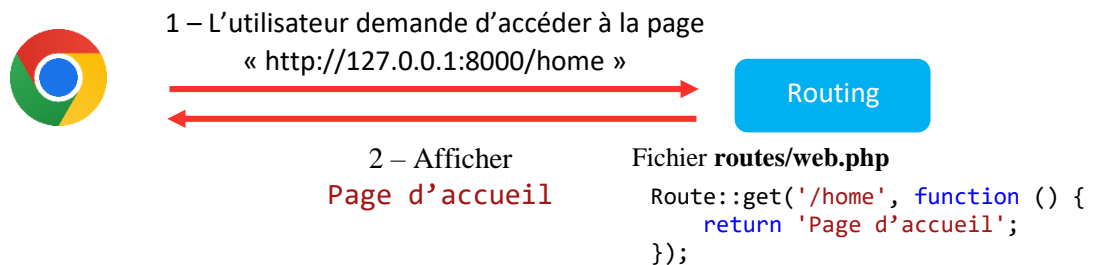
```
Route::get($uri, function () { /* ... */ });
```

- **POST** : elle est aussi très courante, là la requête modifie ou ajoute une ressource, le cas le plus classique est la soumission d'un formulaire (souvent utilisé à tort à la place de PUT),

```
Route::post($uri, function () { /* ... */ });
```

- **PUT** : on ajoute ou remplace complètement une ressource,  
`Route::put($uri, function () { /* ... */ });`
- **DELETE** : on supprime une ressource.  
`Route::delete($uri, function () { /* ... */ });`

Il faut retenir que lorsqu'un utilisateur demande d'accéder à une page dans le navigateur, le framework y répond grâce à la méthode `get` du routeur : `Route::get()`



Parfois, vous devrez peut-être enregistrer une route qui répond à plusieurs verbes HTTP. Vous pouvez le faire en utilisant la méthode **match**. Ou, vous pouvez même enregistrer une route qui répond à tous les verbes HTTP en utilisant la méthode **any**:

```
Route::match(['get', 'post'], '/', function () { /* ... */ });
Route::any('/', '/', function () { /* ... */ });
```

## 6.2. Paramètres des routes :

### Déclarer des paramètres

A l'installation, Laravel a une seule route qui correspond à l'url de base. Voyons maintenant comment créer d'autres routes. Imaginons que nous ayons 3 pages qui doivent être affichées avec ces urls :

- `http://127.0.0.1:8000/1`
- `http://127.0.0.1:8000/2`
- `http://127.0.0.1:8000/3`

J'ai fait apparaître en gras la partie spécifique de l'url pour chaque page. Il est facile de réaliser cela avec ce code :

```
Route::get('1', function() { return 'Je suis la page 1 !'; });
Route::get('2', function() { return 'Je suis la page 2 !'; });
Route::get('3', function() { return 'Je suis la page 3 !'; });
```



### Utiliser les paramètres capturés

On peut utiliser un paramètre pour une route qui accepte des éléments variables en utilisant des accolades. Regardez ce code :

```
Route::get('{n}', function($n) {  
    return 'Je suis la page ' . $n . ' !';  
});
```

**Les paramètres des routes sont toujours entre crochets {} et doivent être composés de caractères alphabétiques et des tirets bas « \_ ».**

### Utiliser plusieurs paramètres

Il est possible de créer des routes avec termes séparés par des barres obliques « / » et avec plusieurs paramètres.

```
Route::get('posts/{post}/comments/{comment}', function ($postId,  
$commentId) {  
    /* ... */  
});
```

### Paramètres optionnels

Le point d'interrogation « ? » permet de rendre le paramètre optionnel.

Chaque variable qui est passé à la fonction de rappel de la route doit avoir une valeur par défaut. (Cas de paramètre optionnel)

```
Route::get('page/{n?}', function($n = null) {  
    if ($n === null)  
        return 'Je suis une page sans numéro !';  
    else  
        return 'Je suis la page ' . $n . ' !';  
});
```

### Routes nommées

Comme son nom l'indique, nous pouvons nommer les routes, ce qui permet de générer des URL ou des redirections pour des routes spécifiques.

Une façon simple de créer une route nommée est fournie par la méthode **name** enchaînée sur la façade **Route**. Le nom de chaque route doit être unique :

```
Route::get('/', function () { /*...*/ })->name("homepage");
```

Pour générer une URL en utilisant le nom de la route

```
$url = route('home'); ou <a href="{ route('home') }">
```



### 6.3. Réponses :

Toutes les routes et tous les contrôleurs doivent renvoyer une réponse à renvoyer au navigateur de l'utilisateur. Laravel propose plusieurs façons différentes de renvoyer des réponses.

La réponse la plus basique consiste à renvoyer une chaîne à partir d'une route ou d'un contrôleur. Le framework convertira automatiquement la chaîne en une réponse HTTP complète :

```
Route::get('/', function () {  
    return 'Bienvenue sur le site !';  
});
```

En plus de renvoyer des chaînes à partir de vos routes et de vos contrôleurs, vous pouvez également renvoyer des tableaux. Le framework convertira automatiquement le tableau en une réponse JSON :

```
Route::get('/', function () {  
    return [1, 2, 3];  
});
```

#### Les réponses textuelles

En règle générale, vous ne renverrez pas simplement de simples chaînes ou des tableaux à partir de vos actions de routage. Au lieu de cela, vous renverrez des instances **Illuminate\Http\Response** ou des vues complètes.

Le renvoi d'une instance **Response** complète vous permet de personnaliser le code d'état HTTP et les en-têtes de la réponse. Une instance **Response** hérite de la classe **Symfony\Component\HttpFoundation\Response**, qui fournit diverses méthodes pour créer des réponses HTTP :

```
Route::get('/', function () {  
    return response('Bienvenue sur le site !', 200)  
        ->header('Content-Type', 'text/plain');  
});
```

#### Les réponses JSON

La méthode **json** définira automatiquement l'en-tête Content-Type sur **application/json**, ainsi que convertira le tableau donné en JSON à l'aide de la fonction **json\_encode** PHP :

```
Route::get('/', function () {  
    return response()->json([1, 2, 3]);  
});
```





## Les redirections

La fonction **redirect()** peut recevoir un paramètre pour lui indiquer à quel endroit elle doit effectuer la redirection.

Pour effectuer une simple redirection :

```
Route::get('ancienne', function () {  
    return redirect('nouvelle');  
});
```

ou

```
Route::redirect('ancienne', 'nouvelle');
```

Par défaut, **Route::redirect** renvoie un **302** code d'état. Vous pouvez personnaliser le code d'état à l'aide du troisième paramètre facultatif :

```
Route::redirect('ancienne', 'nouvelle', 301);
```

Il est possible de rediriger l'utilisateur vers la précédente page visitée, grâce à l'instruction `return redirect()->back();`

### Redirection vers des routes nommées

```
Route::get('/login', function () {  
    return 'Page de connexion';  
})->name("login");  
Route::get('/', function () {  
    return redirect()->route('login');
```

### Redirection vers des actions du contrôleur

Vous pouvez également générer des redirections vers les actions du contrôleur.

Pour ce faire, passez le contrôleur et le nom de l'action à la méthode `action` :

```
use Illuminate\Support\Facades\Route;  
use App\Http\Controllers\IndexController;  
Route::get('/', [IndexController::class, 'index']);
```

## 6.4. Groupes de routes :

### Préfixes de routes

La méthode **prefix** peut être utilisée pour préfixer chaque route du groupe avec un URI donné. Par exemple, vous pouvez préfixer tous les URI suivants :

```
Route::get('/admin/stats', function () { /* ... */ });  
Route::get('/admin/users', function () { /* ... */ });  
Route::get('/admin/logs', function () { /* ... */ });
```





Au sein du groupe avec **admin** :

```
Route::prefix('admin')->group(function () {  
    Route::get('/stats', function () {  
        // Matches The "/admin/stats" URL  
    });  
    Route::get('/users', function () {  
        // Matches The "/admin/users" URL  
    });  
    Route::get('/logs', function () {  
        // Matches The "/admin/logs" URL  
    });  
});
```

## 6.5. Routes et vues

Les vues sont les fichiers **.blade.php** que nous utilisons pour rendre le frontend de notre application Laravel. Elles utilisent le moteur de templating blade, et c'est la manière par défaut de construire une application full-stack en utilisant uniquement Laravel.

Si nous voulons que notre route renvoie une vue (**views/home.blade.php** par exemple), nous pouvons simplement utiliser la méthode **view** sur la façade Route. Elle accepte un paramètre de route, un nom de vue et un tableau facultatif de valeurs à transmettre à la vue.

```
Route::view('/', 'home');
```

ou

```
Route::get('/', function () {  
    return view('home');  
});
```

Supposons que notre vue veuille dire «**Bonjour, {!! \$name !!} !** » en passant un tableau optionnel avec ce paramètre. Nous pouvons le faire avec le code suivant (si le paramètre manquant est requis dans la vue, la requête échouera et lancera une erreur) :

```
Route::view('/', 'home', ['name' => "ALLAOUI"]);
```

ou

```
Route::get('/', function () {  
    return view('home', ['name' => "ALLAOUI"]);  
});
```



## 6.6. Commandes utiles

Pour fournir une vue d'ensemble de toutes les routes définies par votre application, utilisez la commande :

```
php artisan route:list
```

Par défaut, le middleware de route affecté à chaque route ne sera pas affiché dans la sortie **route:list** ; cependant, vous pouvez demander à Laravel d'afficher le middleware de route en ajoutant l'option **-v** à la commande :

```
php artisan route:list -v
```

Vous pouvez également demander à Laravel de n'afficher que les routes commençant par un URI donné :

```
php artisan route:list --path=api
```

De plus, vous pouvez demander à Laravel de masquer toutes les routes définies par des packages tiers en fournissant l'option **--except-vendor** lors de l'exécution de la commande **route:list** :

```
php artisan route:list --except-vendor
```

Lors du déploiement de votre application en production, l'utilisation du cache de route réduira considérablement le temps nécessaire pour enregistrer toutes les routes de votre application.

Pour générer un cache de route, exécutez la commande Artisan **route:cache** :

```
php artisan route:cache
```

Après avoir exécuté cette commande, votre fichier de routes en cache sera chargé à chaque requête. N'oubliez pas que si vous ajoutez de nouvelles routes, vous devrez générer un nouveau cache de routes. Pour cette raison, vous ne devez exécuter la commande **route:cache** que pendant le déploiement de votre projet.

Vous pouvez utiliser la commande **route:clear** pour vider le cache de route :

```
php artisan route:clear
```



## 7. Créer des vues

Les vues contiennent le code HTML du site. Créer une vue signifie donc créer un fichier contenant du HTML qui sera retourné par une route.

Les vues de Laravel doivent se trouver dans le dossier ressources/views et avoir pour extension .php ou .blade.php

Pour retourner une vue, il faut utiliser la fonction view qui prend en paramètre un nom de vue.

```
Route::get('/', function () {  
    return view('home');  
});
```

Dans ce code, lorsqu'un utilisateur vient sur l'URI « / », le fichier home.blade.php présent dans ressources/views est exécuté et retourner à l'utilisateur.

### 7.1. Le moteur de gabarit Blade :

**Blade** est le moteur de template utilisé par Laravel. Son but est de **permettre d'utiliser du php sur notre vue mais d'une manière assez particulière**. Pour créer un fichier qui utilise le moteur de template Blade il vous faut ajouter l'extension **".blade.php"**. Comme nous l'avons vu dans la présentation de l'architecture de Laravel, les fichiers de vos vues se situent dans le dossier ressources/views.

### 7.2. Les variables :

#### Afficher des données

La première fonctionnalité la plus basique de Blade est l'affichage d'une simple variable. Exemple : Si je transmets à ma vue la variable \$maVariable = 'Hello World' ! Dans ma vue {{ \$maVariable }} affichera 'Hello World'.

Il existe une variante de ces accolades pour vous permettre de ne pas échapper les caractères html.

```
$mavariabale = '<h1>Mon Titre</h1>';  
{{ $maVariable }} <!-- donnera : '<h1>Mon Titre</h1>' -->  
{!! $maVariable !!} <!-- donnera : 'Mon Titre' dans une balise  
HTML h1 -->
```

#### Passer des variables aux vues

Vous pouvez afficher les données transmises à vos vues Blade en entourant la variable d'accolades. Par exemple, avec la route suivante :

```
Route::get('/', function () {  
    return view('welcome', ['name' => 'ALLAOUI']);  
});
```

Vous pouvez afficher le contenu de la variable name comme ceci :

Hello, {{ \$name }}.

Par défaut, les instructions Blade `{{ }}` sont automatiquement envoyées via la fonction **htmlspecialchars** de PHP pour empêcher les attaques XSS. Si vous ne souhaitez pas que vos données soient échappées, vous pouvez utiliser la syntaxe suivante :

Hello, `{{ $name }}`.

### 7.3. Les directives :

#### Les conditions

Blade vous permet de manipuler des données comme le ferait le php, il vous est donc possible d'utiliser les différentes structures de contrôle PHP existantes. À la différence que leur écriture diffèrent un tout petit peu.

Vous pouvez construire des instructions if en utilisant les **@if** , **@elseif** , **@else** et **@endif**. Ces directives fonctionnent de manière identique à leurs homologues PHP :

```
<?php $animal = 'cheval'; ?>
@if ( $animal === 'chien' )
    <p>L'animal est un chien.</p>
@elseif ( $animal === 'chat' )
    <p>L'animal n'est pas un chien.</p>
@else
    <p>L'animal n'est ni un chat ni un chien.</p>
@endif
```

**Donnera :**

```
<p>L'animal n'est ni un chat ni un chien.</p>
```

En plus des directives conditionnelles déjà évoquées, les directives **@isset** et **@empty** peuvent être utilisées comme raccourcis pratiques pour leurs fonctions PHP respectives :

#### @isset

```
<?php $produit = 'costume'; ?>
@isset($produit)
    <p>Le produit existe</p>
@endisset
```

**Donnera :**

```
<p>Le produit existe</p>
```

#### @empty

```
<?php $produit = ''; ?>
@empty($produit)
    <p>Le produit n'existe pas.</p>
@endempty
```

**Donnera :**

```
<p>Le produit n'existe pas.</p>
```

Les instructions Switch peuvent être construites à l'aide des **@switch** , **@case** , **@break** , **@default** et **@endswitch** :

```
<?php $age = 19; ?>
@switch($age)
    @case( $age < 18 )
        <p>La personne est mineure.</p>
        @break
    @case( $age > 18 )
        <p>La personne est majeure.</p>
        @break
    @default
        <p>valeur par défaut.</p>
@endswitch
```

**Donnera :**

```
<p>La personne est majeure.</p>
```

## Les boucles

En plus des déclarations conditionnelles, Blade fournit des directives simples pour travailler avec les structures de boucles de PHP.

### @while

```
<?php $i = 1; ?>
@while ($i < 3)
    <p>$i est égal à {{ $i ++ }}</p>
@endwhile
```

**Donnera :**

```
<p>$i est égal à 1</p>
<p>$i est égal à 2</p>
```

### @foreach

```
<?php $letters = ['a', 'b', 'c']; ?>
@foreach ( $letters as $letter )
    <p>Lettre : {{ $letter }}</p>
@endforeach
```

**Donnera :**

```
<p>Lettre : a</p>
<p>Lettre : b</p>
<p>Lettre : c</p>
```

### @for

```
<?php $numbers = [1, 2, 3]; ?>
@for ($i = 0; $i < count($numbers); $i++)
    <p>Nombre : {{ $numbers[$i] }}</p>
@endfor
```

**Donnera :**

```
<p>Nombre : 1</p>
<p>Nombre : 2</p>
<p>Nombre : 3</p>
```

**@forelse**

Le **@forelse** est un foreach qui vous permet de retourner ce que vous souhaitez si le tableau est vide. Cela vous économisera l'ajout d'une condition if

```
<?php $animals = ['chien', 'chat', 'cheval']; ?>
@forelse ($animals as $animal)
    <li>{{ $animal }}</li>
@empty
    <p>Aucun animal existant.</p>
@endforelse
```

**Donnera :**

```
<li>chien</li>
<li>chat</li>
<li>cheval</li>
```

Lorsque vous utilisez des boucles, vous pouvez également terminer la boucle ou sauter l'itération en cours :

```
<?php $days = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi',
'samedi', 'dimanche']; ?>
@foreach ($days as $day)
    @if ($day == 'mardi')
        @continue
    @endif
    <p>{{ $day }}</p>
    @if ($day == 'jeudi')
        @break
    @endif
@endforeach
```

**Donnera :**

```
<p>lundi</p>
<p>mercredi</p>
<p>jeudi</p>
```

Vous pouvez également inclure la condition avec la déclaration de la directive sur une seule ligne :

```
@foreach ($days as $day)
    @continue($day == 'mardi')
    <p>{{ $day }}</p>
    @break($day == 'jeudi')
@endforeach
```

Lors de la boucle, une variable **\$loop** sera disponible à l'intérieur de votre boucle. Cette variable donne accès à des informations utiles telles que l'index de la boucle actuelle et s'il s'agit de la première ou de la dernière itération de la boucle :

```
<?php $days = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']; ?>
@foreach ($days as $day)
    @if ($loop->first)
        <p>C'est le premier jour de la semaine : {{ $day }}</p>
    @endif
    @if ($loop->last)
        <p>C'est le dernier jour de la semaine : {{ $day }}</p>
    @endif
@endforeach
```

**Donnera :**

```
<p>C'est le premier jour de la semaine : lundi</p>
<p>C'est le dernier jour de la semaine : dimanche</p>
```

La variable **\$loop** contient également une variété d'autres propriétés utiles :

Property	Description
<b>\$loop-&gt;index</b>	L'index de l'itération de la boucle actuelle (commence à 0).
<b>\$loop-&gt;iteration</b>	L'itération actuelle de la boucle (commence à 1).
<b>\$loop-&gt;remaining</b>	Les itérations restantes dans la boucle.
<b>\$loop-&gt;count</b>	Le nombre total d'éléments dans le tableau en cours d'itération.
<b>\$loop-&gt;first</b>	Si c'est la première itération de la boucle.
<b>\$loop-&gt;last</b>	Si c'est la dernière itération de la boucle.
<b>\$loop-&gt;even</b>	Si c'est une itération paire dans la boucle.
<b>\$loop-&gt;odd</b>	Si c'est une itération impaire dans la boucle.
<b>\$loop-&gt;depth</b>	Le niveau d'imbrication de la boucle actuelle.
<b>\$loop-&gt;parent</b>	Dans une boucle imbriquée, la variable de boucle du parent.

### Autres

#### Commentaires

Blade vous permet également de définir des commentaires dans vos vues. Cependant, contrairement aux commentaires HTML, les commentaires Blade ne sont pas inclus dans le HTML renvoyé par votre application :

```
{{ -- Ce commentaire ne sera pas présent dans le HTML rendu -- }}
```



### PHP

Dans certaines situations, il est utile d'intégrer du code PHP dans vos vues. Vous pouvez utiliser la directive Blade **@php** pour exécuter un bloc de PHP simple dans votre modèle :

```
@php
    //Ceci est de PHP
@endphp
```

## 7.4. Organiser les vues :

### Définir le template

Deux des principaux avantages de l'utilisation de Blade sont l'héritage des modèles et les sections. Pour commencer, examinons un exemple simple. Tout d'abord, nous examinerons une mise en page « maître ». Étant donné que la plupart des applications Web conservent la même disposition générale sur différentes pages, il est pratique de définir cette disposition comme une seule vue Blade :

```
<!-- Voici le template situé dans resources/views/layout.blade.php
-->
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Application - @yield('title')</title>
</head>
<body>
    @section('sidebar')
        Contenu de la section 'sidebar' du parent
    @show
    <div class="container">
        @yield('content')
    </div>
</body>
</html>
```

Comme vous pouvez le voir, ce fichier contient un balisage HTML typique. Cependant, prenez note des directives **@section** et **@yield**. La directive **@section**, comme son nom l'indique, définit une section de contenu, tandis que la directive **@yield** est utilisée pour afficher le contenu d'une section donnée.





### Extension d'une mise en page

Lors de la définition d'une vue enfant, utilisez la directive Blade **@extends** pour spécifier la disposition dont la vue enfant doit « hériter ». Les vues qui étendent une mise en page Blade peuvent injecter du contenu dans les sections de la mise en page à l'aide des directives **@section**. N'oubliez pas, comme le montre l'exemple ci-dessus, le contenu de ces sections sera affiché dans la mise en page en utilisant **@yield** :

```
<!-- Stored in resources/views/child.blade.php -->
@extends('layout')

@section('title', 'Titre de la page enfant')

@section('sidebar')
    <!-- Ajout de contenu avant le contenu du parent -->
    @parent <!-- Contenu de la section 'sidebar' du parent -->
    <!-- Ajout de contenu après le contenu du parent -->
@endsection

@section('content')
    <!-- Contenu de la section 'content' de l'enfant -->
@endsection
```

Les vues de lames peuvent être renvoyées à partir des routes à l'aide de l'assistant de **view** globale :

```
Route::get('/', function () {
    return view('child');
});
```



## 8. Créer des contrôleurs

### 8.1. Définition et usage :

Un contrôleur est une classe qui va contenir différentes méthodes. Chaque méthode correspondant généralement à une opération (URL) de votre application.

Les routes créées dans les chapitres précédents sont surchargées de code pour traiter les actions dans leurs fonctions de rappel.

Exemple de route avec fonction de rappel

```
Route::get('/', function() { return 'Bienvenue sue la page  
d'accueil !'; });
```

L'objectif est de les libérer de ce code pour obtenir un fichier de routes propre qui fait simplement le lien entre les URI de l'application et les actions à effectuer pour chacune de ces URI.

Exemple de route sans fonction de rappel

```
Route::get('/', [IndexController::class, 'index']);
```

### 8.2. Créer et utiliser un contrôleur :

#### Créer un contrôleur avec Artisan

Créer un contrôleur sous sa forme la plus simple se fait grâce à la commande suivante :

```
php artisan make:controller IndexController
```

Les contrôleurs sont créés dans le dossier **app/Http/Controllers**

```
<?php  
namespace App\Http\Controllers;  
class IndexController extends Controller  
{  
    //  
}
```

#### Structure d'un contrôleur

Un contrôleur est une classe qui hérite la classe de base Controller et dont chaque méthode publique représente généralement une action qui correspond à une route.

```
<?php  
namespace App\Http\Controllers;  
class IndexController extends Controller  
{  
    public function index() {  
        return 'Bienvenue sue la page d'accueil !';  
    }  
}
```



Une méthode d'un contrôleur retourne une réponse. Dans cet exemple, la méthode `index` retourne une réponse sous la forme d'une chaîne de caractère, mais il peut retourner une vue ou un objet JSON.

```
<?php
namespace App\Http\Controllers;
class IndexController extends Controller
{
    public function index() {
        return view('welcome');
    }
    public function page($id) {
        return view('pages.show', ['id'=>$id]);
    }
}
```

### Lier une route à un contrôleur

Dans le fichier `routes/web.php`, à la place d'utiliser des fonctions de rappel, vous pouvez signaler au routeur le contrôleur à chercher et quelle action effectuer.

```
Route::get('/', [IndexController::class, 'index']);
Route::get('/pages/{id}', [IndexController::class, 'page']);
```

## 8.3. Contrôleur de ressources :

Créer un contrôleur de type CRUD se fait grâce à la commande suivante :

```
php artisan make:controller PostsController --resource
```

En ajoutant le paramètre `--resource`, la commande génère une structure de contrôleur avec toutes les méthodes classiques de manipulation de ressource.

Cette commande générera un contrôleur

à **`app/Http/Controllers/PostsController.php`**. Le contrôleur contiendra une méthode pour chacune des opérations de ressources disponibles.

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index() { /*...*/ }
```



```
/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */
public function create(){ /*...*/ }

/**
 * Store a newly created resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request){ /*...*/ }

/**
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function show($id){ /*...*/ }

/**
 * Show the form for editing the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function edit($id){ /*...*/ }

/**
 * Update the specified resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id){ /*...*/ }

/**
 * Remove the specified resource from storage.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
```

```
    public function destroy($id){ /*...*/ }
}
```

### Déclaration des routes :

Vous pouvez enregistrer une route de ressources qui pointe vers le contrôleur :

```
<?php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\PostController;
Route::resource('posts', PostController::class);
```

Cette déclaration de route unique crée plusieurs routes pour gérer diverses actions sur la ressource.

### Actions gérées par le contrôleur de ressources

Verbe	URI	Action	Route
GET	/posts	index	posts.index
GET	/posts/create	create	posts.create
POST	/posts	store	posts.store
GET	/posts/{slug}	show	posts.show
GET	/posts/{slug}/edit	edit	posts.edit
PUT/PATCH	/posts/{slug}	update	posts.update
DELETE	/posts/{slug}	destroy	posts.destroy

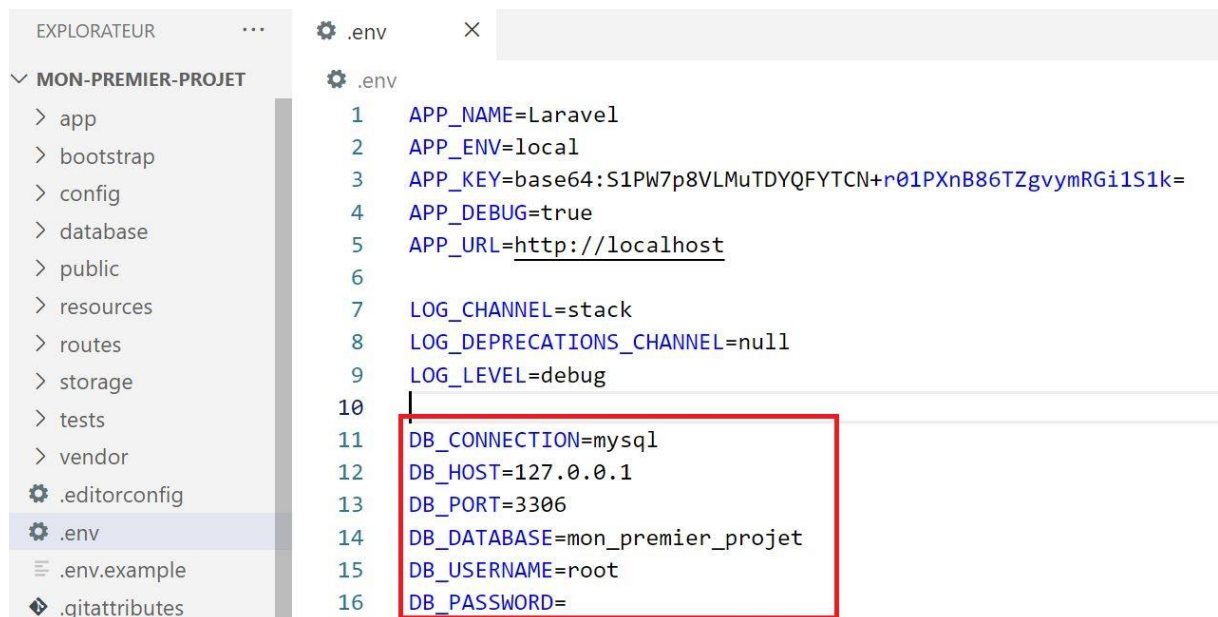
Pour tester, taper la commande suivante :

**php artisan route:list**

```
GET|HEAD | posts | posts.index | App\Http\Controllers\PostController@index
POST     | posts | posts.store | App\Http\Controllers\PostController@store
GET|HEAD | posts/create | posts.create | App\Http\Controllers\PostController@create
GET|HEAD | posts/{post} | posts.show | App\Http\Controllers\PostController@show
PUT|PATCH | posts/{post} | posts.update | App\Http\Controllers\PostController@update
DELETE   | posts/{post} | posts.destroy | App\Http\Controllers\PostController@destroy
GET|HEAD | posts/{post}/edit | posts.edit | App\Http\Controllers\PostController@edit
```



Le stockage de données confidentielles comme les informations d'identification dans `.env` est particulièrement utile pour les référentiels git et autres plates-formes de partage.



```

1 APP_NAME=Laravel
2 APP_ENV=local
3 APP_KEY=base64:S1PW7p8VLMuTDYQFYTCN+r01PXnB86TZgvymRGi1S1k=
4 APP_DEBUG=true
5 APP_URL=http://localhost
6
7 LOG_CHANNEL=stack
8 LOG_DEPRECATIONS_CHANNEL=null
9 LOG_LEVEL=debug
10
11 DB_CONNECTION=mysql
12 DB_HOST=127.0.0.1
13 DB_PORT=3306
14 DB_DATABASE=mon_premier_projet
15 DB_USERNAME=root
16 DB_PASSWORD=

```

Vous devrez spécifier le nom de la base de données **mon\_premier\_projet**, que nous avons créée, après **DB\_DATABASE =** et également spécifier le nom d'utilisateur et le mot de passe en fonction de vos besoins. Enregistrez ensuite le fichier.

Vous pouvez utiliser écrire le code suivant pour vérifier la connexion à la base de données dans laravel:

```

<?php
    try {
        DB::connection()->getPdo();
        if (DB::connection()->getDatabaseName()) {
            echo "Yes! Successfully connected to the DB: " .
                DB::connection()->getDatabaseName();
        } else {
            die("Could not find the database");
        }
    } catch (\Exception $e) {
        die("Could not connect to the database. error:" . $e );
    }
?>

```

**Affichage:**



Yes! Successfully connected to the DB: mon\_premier\_projet





## 9.2. Object Relational Mapping

Laravel eloquent ORM est livré avec le framework Laravel pour fournir un moyen facile, sans tracas et simple de travailler avec une base de données. Certaines des fonctionnalités qui ont fait sa renommée sont la suppression douce, le timestamp, l'implémentation d'ActiveRecord, la gestion de bases de données multiples, le chargement rapide, les observateurs de modèle, les événements de modèle, et bien plus encore. Les relations Eloquent sont identiques aux méthodes des classes de modèles Eloquent. Les relations sont de puissants constructeurs de requêtes lorsqu'elles sont définies comme des méthodes, ce qui permet de chaîner les méthodes et d'obtenir des capacités de requêtes puissantes.

## 9.3. Les migrations

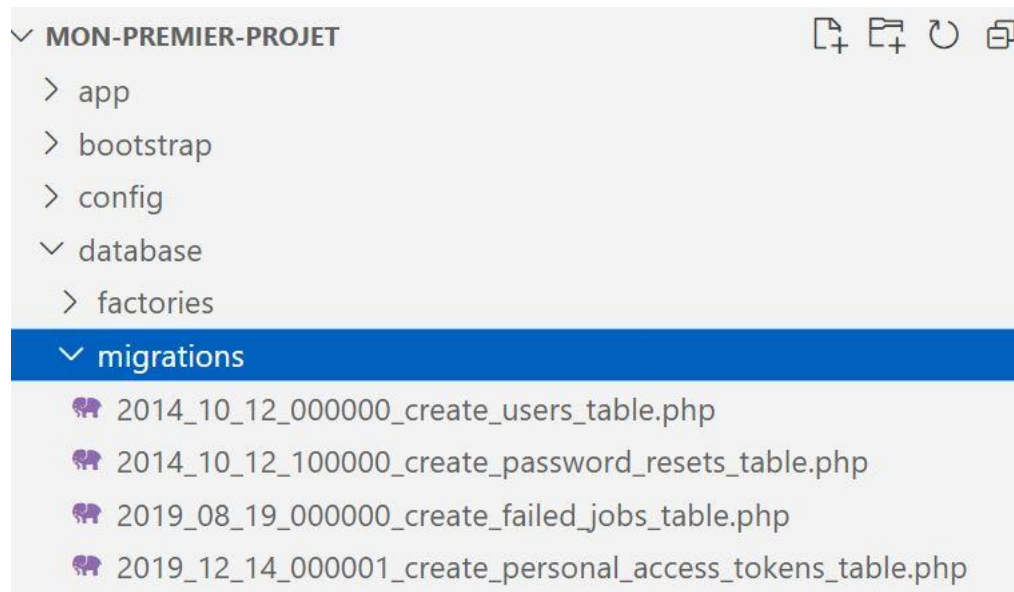
### Introduction

Les migrations sont une sorte de carnet de bords de toutes les modifications qui peuvent être effectuées dans l'organisation de votre base de données (création d'une table, ajout, suppression ou renommage de colonne dans une table etc...).

Vos migrations sont des fichiers php que vous pouvez retrouver directement dans vos dossiers, elles se situent dans le dossier : **database/migrations**

Si vous regardez dans le dossier database/migrations il y a déjà 4 migrations présentes :

- table **users** : c'est une migration de base pour créer une table des utilisateurs,
- table **password\_resets** : c'est une migration liée à la précédente qui permet de gérer le renouvellement des mots de passe en toute sécurité,
- table **failed\_jobs** : une migration qui concerne les queues,
- table **personal\_access\_tokens** concerne les api.



Chaque migration dispose de deux fonctions :

- **up()** qui contient le code de création de la table et de ses colonnes
- **down()** qui contient le code de suppression de la table





Si vous ouvrez le fichier  
database/migrations/2014\_10\_12\_000000\_create\_users\_table.php vous trouvez ce  
code :

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

### Lancer les migrations

Pour lancer les migrations on utilise la commande migrate :

#### **php artisan migrate**

Depuis Laravel 5.4, un changement au jeu de caractères par défaut de la base de données (utf8mb4 qui inclut le support pour le stockage des emojis) risque de vous poser quelques problèmes lors de l'installation de votre BDD MySQL ou MariaDB.



Vous allez rencontrer cette erreur lorsque vous essayez d'exécuter des migrations :

```
PS E:\M205-back-end\mon-premier-projet> php artisan migrate
Migrating: 2014_10_12_000000_create_users_table
```

**Illuminate\Database\QueryException**

```
SQLSTATE[42000]: Syntax error or access violation: 1071 La clé est trop longue. Longueur maximale: 1000 (SQL: alter table `users` add unique `users_email_unique` (`email`))
```

Pour corriger ce problème, vous avez deux solutions : Mettre MySQL à jour (recommandé) ou modifier votre fichier AppServiceProvider.php et définir une longueur de chaîne par défaut dans la méthode boot :

#### app/Providers/AppServiceProvider.php

```
<?php
namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Schema;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Schema::defaultStringLength(191);
    }
}
```

On voit que les 4 migrations présentes ont été exécutées

```
PS E:\M205-back-end\mon-premier-projet> php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (722.16ms)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (936.65ms)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (463.94ms)
Migrating: 2019_12_14_000001_create_personal_access_tokens_table
Migrated: 2019_12_14_000001_create_personal_access_tokens_table (1,254.30ms)
```

Et on trouve les 4 tables dans la base (en plus de celle de gestion des migrations) :

Table	Action
<input type="checkbox"/> failed_jobs	★ Parcourir Structure Rechercher Insérer Vider Supprimer
<input type="checkbox"/> migrations	★ Parcourir Structure Rechercher Insérer Vider Supprimer
<input type="checkbox"/> password_resets	★ Parcourir Structure Rechercher Insérer Vider Supprimer
<input type="checkbox"/> personal_access_tokens	★ Parcourir Structure Rechercher Insérer Vider Supprimer
<input type="checkbox"/> users	★ Parcourir Structure Rechercher Insérer Vider Supprimer

Pour comprendre le lien entre migration et création de la table associée voici une illustration pour la table **users** :

#	Nom	Type
1	id	bigint(20)
2	name	varchar(255)
3	email	varchar(255)
4	email_verified_at	timestamp
5	password	varchar(255)
6	remember_token	varchar(100)
7	created_at	timestamp
8	updated_at	timestamp

```

Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email')->unique();
    $table->timestamp('email_verified_at')->nullable();
    $table->string('password');
    $table->rememberToken();
    $table->timestamps();
});

```

La méthode **timestamps** permet la création des deux colonnes **created\_at** et **updated\_at**.

### Annuler ou rafraichir une migration

Pour annuler une migration on utilise la commande **rollback** :

**php artisan migrate:rollback**

Les méthodes **down** des migrations sont exécutées et les tables sont supprimées.

Pour annuler et relancer en une seule opération on utilise la commande **refresh** :

**php artisan migrate:refresh**

Pour éviter d'avoir à coder la méthode **down** on a la commande **fresh** qui supprime automatiquement les tables concernées :

**php artisan migrate:fresh**

## Créer une migration

Pour créer une nouvelle migration il vous suffit de rentrer la commande :

**php artisan make:migration CreateArticlesTable**

La migration est créée dans le dossier :



Après avoir créé votre migration vous pouvez commencer à mettre en place votre table.

On nous précise que la migration a été créée avec succès. On peut donc se rendre dans notre fichier. Dans cette table articles je veux qu'il y ait un titre, une description, un contenu et un auteur.

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateArticlesTable extends Migration
{
    /**
     * Run the migrations.
     * @return void
     */
    public function up() {
        Schema::create('articles', function (Blueprint $table) {
            $table->increments('id');
            $table->string('title', 255);
            $table->text('description', 1048);
            $table->text('contenu');
            $table->unsignedInteger('user_id');
            $table->foreign('user_id')
                ->references('id')
                ->on('users');
            $table->timestamps();
        });
    }
}
```



```
/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
    Schema::dropIfExists('articles');
}
}
```

Lancer la migration on utilise la commande migrate :

**php artisan migrate**

```
PS E:\M205-back-end\mon-premier-projet> php artisan migrate
Migrating: 2023_02_14_143117_create_articles_table
Migrated: 2023_02_14_143117_create_articles_table (546.64ms)
PS E:\M205-back-end\mon-premier-projet> []
```

## Les tables

### Créer une table

Pour créer une nouvelle table de base de données, utilisez la méthode **create** sur la façade **Schema**. La méthode **create** accepte deux arguments. Le premier est le nom de la table, tandis que le second est un objet **Blueprint** utilisé pour définir la nouvelle table :

```
Schema::create('articles', function (Blueprint $table) {
    $table->increments('id');
});
```

### Vérification de l'existence d'une table/colonne

Vous pouvez facilement vérifier l'existence d'une table ou d'une colonne en utilisant les méthodes **hasTable** et **hasColumn**:

```
if (Schema::hasTable('articles')) { /* ... */ }
if (Schema::hasColumn('articles', 'id')) { /* ... */ }
```

### Renommer / supprimer des tables

Pour renommer une table de base de données existante, utilisez la méthode **rename**:

```
Schema::rename($from, $to);
```

Pour supprimer une table existante, vous pouvez utiliser les méthodes **drop** ou **dropIfExists**:

```
Schema::drop('users');
Schema::dropIfExists('users');
```

## Les colonnes

### Créer une colonne

La méthode **table** sur la façade **Schema** peut être utilisée pour mettre à jour des tables existantes. Comme la méthode **create**, la méthode **table** accepte deux arguments : le nom de la table et une instance **Blueprint** que vous pouvez utiliser pour ajouter des colonnes à la table :

```
Schema::create('articles', function (Blueprint $table) {
    $table->string('title');
});
```

### Types de colonnes

La façade **Schema** contient une variété de types de colonnes que vous pouvez spécifier lors de la création de vos tables :

Code	Description
<code>\$table-&gt;increments('id');</code>	ID (clé primaire) auto incrément
<code>\$table-&gt;integer('votes');</code>	Un entier signé
<code>\$table-&gt;float('amount', 8, 2);</code>	Un nombre à virgule flottante
<code>\$table-&gt;decimal('amount', 5, 2);</code>	Un nombre décimal avec une précision de 5 et une échelle de 2.
<code>\$table-&gt;double('column', 15, 8);</code>	Un nombre double avec précision, 15 chiffres au total et 8 après la virgule.
<code>\$table-&gt;string('email', 100);</code>	Une chaîne de caractère avec une longueur maximale
<code>\$table-&gt;binary('data');</code>	Une colonne de type BLOB (biniaire)
<code>\$table-&gt;boolean('confirmed');</code>	Un booléen
<code>\$table-&gt;date('created_at');</code>	Une date
<code>\$table-&gt;dateTime('created_at');</code>	Une colonne de datetime
<code>\$table-&gt;enum('choices', ['foo', 'bar']);</code>	Un type ENUM pour la base de données.
<code>\$table-&gt;json('options');</code>	Un objet de type Json
<code>\$table-&gt;text('description');</code>	Un text
<code>\$table-&gt;timestamps();</code>	Ajoute les colonnes created_at et updated_at.

### Modificateurs de colonne

En plus des types de colonne répertoriés ci-dessus, il existe plusieurs "modificateurs" de colonne que vous pouvez utiliser lors de l'ajout d'une colonne à une table de base de



données. Par exemple, pour rendre la colonne "nullable", vous pouvez utiliser la méthode **nullable** :

```
Schema::table('articles', function (Blueprint $table) {
    $table->string('description')->nullable();
});
```

Vous trouverez ci-dessous une liste de tous les modificateurs de colonne disponibles.

Modificateur	Description
->after('column')	Placer la colonne "après" une autre colonne
->comment('my comment')	Ajouter un commentaire à une colonne
->default(\$value)	Spécifier une valeur "par défaut" pour la colonne
->first()	Placer la colonne "première" dans la table
->nullable()	Autoriser l'insertion de valeurs NULL dans la colonne
->storedAs(\$expression)	Créer une colonne générée stockée
->unsigned()	Définir UNSIGNED sur les colonnes integer
->virtualAs(\$expression)	Créer une colonne générée virtuelle

### Supprimer des colonnes

Pour supprimer une colonne, utilisez la méthode **dropColumn** sur le générateur de schéma.

```
Schema::table('articles', function (Blueprint $table) {
    $table->dropColumn('contenu');
});
```

### Index

La façade **Schema** prend en charge plusieurs types d'index.

#### Types d'index disponibles

Code	Description
\$table->primary('id');	Ajoutez une clé primaire.
\$table->primary(['first', 'last']);	Ajoutez des clés composites.
\$table->unique('email');	Ajouter un index unique.
\$table->unique('state', 'my_index_name');	Ajoutez un nom d'index personnalisé.
\$table->unique(['first', 'last']);	Ajoutez un index unique composite.
\$table->index('state');	Ajouter un index de base.

## Supprimer d'index

Pour supprimer un index, vous devez spécifier le nom de l'index. Par défaut, Laravel attribue automatiquement un nom raisonnable aux index. Concaténer simplement le nom de la table, le nom de la colonne indexée et le type d'index.

Commande	Description
<code>\$table-&gt;dropPrimary('users_id_primary');</code>	Supprimer une clé primaire.
<code>\$table-&gt;dropUnique('users_email_unique');</code>	Supprimer un index unique.
<code>\$table-&gt;dropIndex('geo_state_index');</code>	Supprimez un index de base.

## Contraintes de clé étrangère

Laravel prend également en charge la création de contraintes de clé étrangère, qui sont utilisées pour forcer l'intégrité référentielle au niveau de la base de données. Par exemple, définissons une colonne **user\_id** sur la table **articles** qui fait référence à la colonne **id** sur une table **users** :

```
Schema::create('articles', function (Blueprint $table) {
    $table->unsignedInteger('user_id');
    $table->foreign('user_id')->references('id')->on('users');
});
```

Vous pouvez également spécifier l'action souhaitée pour les propriétés "on delete" et "on update" de la contrainte :

```
$table->foreign('user_id')->references('id')->on('users')-
    >onDelete('cascade');
```

Pour supprimer une clé étrangère, vous pouvez utiliser la méthode `dropForeign`. Les contraintes de clé étrangère utilisent la même convention de dénomination que les index.

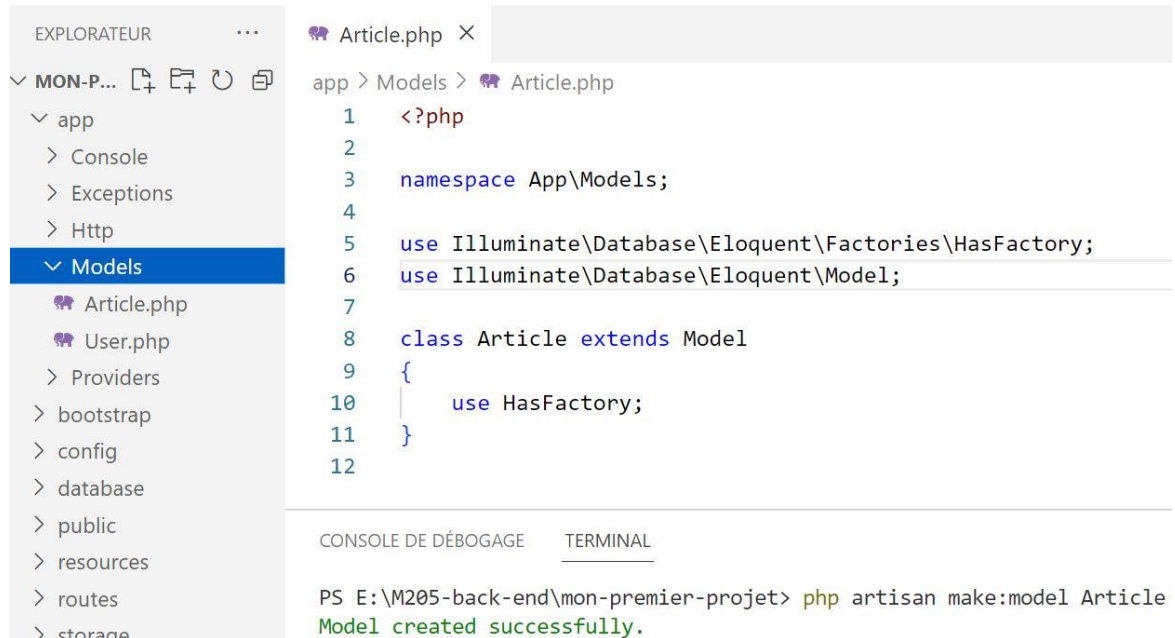
```
$table->dropForeign('articles_user_id_foreign');
```



## 9.4. Construire des modèles

Avec Eloquent une table est représentée par une classe qui étend la classe **Model**. On peut créer un modèle avec Artisan :

**php artisan make:model Article**



```

EXPLORATEUR
MON-P...
  app
    > Console
    > Exceptions
    > Http
    > Models
      Article.php
      User.php
    > Providers
  bootstrap
  config
  database
  public
  resources
  routes
  storage

Article.php
1  <?php
2
3  namespace App\Models;
4
5  use Illuminate\Database\Eloquent\Factories\HasFactory;
6  use Illuminate\Database\Eloquent\Model;
7
8  class Article extends Model
9  {
10     use HasFactory;
11 }
12

CONSOLE DE DÉBOGAGE
TERMINAL
PS E:\M205-back-end\mon-premier-projet> php artisan make:model Article
Model created successfully.
  
```

Par sécurité ce type d'assignement de masse est limité par la propriété **\$fillable** au niveau du modèle qui désigne précisément les noms des colonnes susceptibles d'être modifiées. Dans le modèle **Article** on va ajouter ce code :

```

<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    protected $fillable = [
        'title',
        'description',
        'contenu',
        'user_id',
    ];
}
  
```

Ce sont les seules colonnes qui seront impactées par la méthode **create**.

### Créer un modèle et une migration

Vous pouvez créer le modèle et sa migration en utilisant la commande suivante :

**php artisan make:model Article --migration**



## 9.5. Utiliser les modèles (Créer, récupérer, modifier et supprimer une instance de modèle)

### Créer et enregistrer un modèle

Une fois la classe de modèle générée et la table construite, créer une nouvelle instance d'un modèle se fait simplement à l'aide du mot-clé **new** sur la classe, comme pour créer n'importe quel autre objet en PHP.

**Les instances d'un modèle peuvent être créées n'importe où dans le code : dans une méthode d'un contrôleur, ou directement dans la fonction de rappel d'une route.**

Vous pouvez enregistrer un modèle grâce à la fonction **save** disponible sur toutes les instances de modèles.

```
$user1 = new User;  
$user1->name = 'ALLAOUI';  
$user1->email = 'allaoui@gmail.com';  
$user1->password = '123456';  
$user1->save();
```

Lorsqu'un modèle est enregistré, cela signifie qu'il est persisté en base de données.

Il est possible de créer et enregistrer un modèle en base de données grâce à la méthode statique **create** disponible sur toutes les instances de modèles.

```
User::create([  
    'name' => 'BENANI',  
    'email' => 'benani@gmail.com',  
    'password' => '123456',  
]);
```

*Les champs `create_at` et `update_at` seront automatiquement mis à jour selon la date de système.*

2	BENANI	benani@gmail.com	NULL	123456	NULL	2023-02-15 08:23:51	2023-02-15 08:23:51
---	--------	------------------	------	--------	------	------------------------	------------------------

### Récupérer un modèle

#### Récupérer un modèle par la méthode find

Pour récupérer un modèle, on utilise la méthode **find** héritant de **Model** qui permet de charger un modèle par valeur de son identifiant.

```
$user1 = User::find(1);  
echo $user1->name;
```

Il existe une autre méthode **findOrFail** qui permet de renvoyer l'instance s'il existe ou de renvoyer une réponse http 404.



## Récupérer un modèle par une requête

On souhaite parfois chercher un modèle par une autre propriété. Pour cela, il est possible d'utiliser la méthode **where** suivie de la méthode **first** si vous voulez récupérer le premier élément qui correspond aux critères passés dans la fonction **where**.

Il existe une autre méthode **firstOrFail** qui retourne une réponse http 404 dans le cas où le modèle n'est pas trouvé.

```
$user1 = User::where('email', 'allaoui@gmail.com')->first();  
echo $user1->name;
```

## Modifier un modèle

### Modifier un modèle

Appeler la fonction **save** disponible sur toutes les instances de modèles sur un modèle récupéré permet de mettre à jour ce modèle.

```
$user1 = User::find(1);  
$user1->email = 'allaoui@ofppt.ma';  
$user1->save();
```

*Le champ `update_at` sera automatiquement mis à jour si le modèle existe.*

### Modifier plusieurs modèles

Il est possible de mettre à jour plusieurs modèles grâce à la combinaison des méthodes **where** et **update**.

Par exemple, le code suivant change le statut des articles publiés y a 5 ans

```
$old = now()->subYears(5);  
Article::where('created_at', '<', $old)  
->update('status'=>'archived');
```

## Supprimer un modèle

### Supprimer un modèle

Il existe plusieurs méthodes pour supprimer un modèle.

Supprimer un modèle avec la méthode **delete**

```
$user1 = User::find(2);  
$user1->delete();
```

Supprimer un modèle avec la méthode **destroy**

```
User::destroy(2);
```

### Supprimer plusieurs modèles

La méthode **destroy** permet aussi de supprimer un ensemble des modèles en passant en paramètre un tableau des identifiants.

```
User::destroy([2, 3, 4]);
```



## 9.6. Utiliser une collection de modèles

### Récupérer une liste

La méthode **all** de la classe **Model** permet de récupérer l'ensemble des enregistrements d'une table.

Par exemple : Pour afficher les noms des utilisateurs, on utilise le code suivant :

```
$users = User::all();  
foreach($users as $user) {  
    echo $user->name;  
}
```

```
#SELECT * FROM `users`;
```

Il est aussi possible de récupérer l'ensemble des enregistrements grâce à la combinaison des méthodes **where** et **get**.

Par exemple : Pour afficher les titres des articles archivés, on utilise le code suivant :

```
$articles = Article::where('status', 'archived')->get();  
foreach($articles as $article) {  
    echo $article->titre;  
}
```

```
#SELECT * FROM `articles` where 'status' LIKE 'archived';
```

### Requête complexe (Query Builder)

Il est possible d'élaborer des requêtes complexes simplement en chainant des méthodes comme **where**, **whereIn**, **orderBy**, **groupBy**, **count**, etc

#### Utilisation de la clause WHERE

```
$users = User::where('name', 'BENANI')->get();
```

```
#SELECT * FROM `users` where 'name' LIKE 'BENANI';
```

#### Utilisation de la clause OR WHERE

```
$users = User::where('name', 'BENANI')->orWhere('name', 'ALLAOUI')->get();
```

```
#SELECT * FROM `users` where 'name' LIKE 'BENANI' OR 'name' LIKE 'ALLAOUI';
```

#### Utilisation de la clause WHEREIN

```
$users = User::whereIn('name', ['BENANI', 'ALLAOUI'])->get();
```

```
#SELECT * FROM `users` where 'name' IN ('BENANI', 'ALLAOUI');
```

```
$users = User::whereNotIn('name', ['BENANI', 'ALLAOUI'])->get();
```

```
#SELECT * FROM `users` where 'name' NOT IN ('BENANI', 'ALLAOUI');
```



### Utilisation de WHERE BETWEEN

```
$users = User::whereBetween('id', [2, 5])->get();  
#SELECT * FROM `users` where 'id' BETWEEN 2 and 5;
```

### Spécification d'une clause SELECT

```
$ids = User::select('id')->get();  
#SELECT id FROM `users`;
```

### Utilisation de WHERE NULL pour trouver des valeurs non définies

```
$users = User::whereNull('email')->get();  
#SELECT * FROM `users` where 'email' IS NULL;
```

### Offset & Limite

Vous pouvez utiliser les méthodes **skip** et **take** pour limiter le nombre de résultats renvoyés par la requête ou pour ignorer un nombre donné de résultats dans la requête :

```
$users = User::skip(1)->take(1)->get();  
#SELECT * FROM `users` LIMIT 1 OFFSET 1;
```

Alternativement, vous pouvez utiliser les méthodes **limit** et **offset**. Ces méthodes sont fonctionnellement équivalentes aux méthodes **take** et **skip**, respectivement :

```
$users = User::offset(1)->limit(1)->get();  
#SELECT * FROM `users` LIMIT 1 OFFSET 1;
```

### Agrégats

Le Query Builder fournit naturellement des méthodes d'agrégats, tel que **count**, **max**, **min**, **avg**, et **sum**.

```
$nombre = User::count();  
#SELECT count(*) FROM `users`;  
$max = User::max('salaire');  
#SELECT max('salaire') FROM `users`;  
$min = User::min('salaire');  
#SELECT min('salaire') FROM `users`;  
$total = User::sum('salaire');  
#SELECT sum('salaire') FROM `users`;  
$moyenne = User::avg('salaire');  
#SELECT avg('salaire') FROM `users`;
```



## 10. Gérer les relations

Les tables de base de données sont souvent liées les unes aux autres. Par exemple, un article de blog peut contenir de nombreux commentaires ou une commande peut être liée à l'utilisateur qui l'a passée. Eloquent facilite la gestion et l'utilisation de ces relations et prend en charge une variété de relations communes :

### 10.1. La relation 1..1

Une relation un à un (one-to-one) est un type très basique de relation de base de données mais moins utile. Il s'agit d'une relation unique entre deux entités.

Par exemple, si deux entités **Fournisseur** et **Adresse** existent, on peut déclarer qu'un fournisseur possède une et une seule adresse et s'une adresse appartient à un et un fournisseur. Il est donc de type un à un.

#### Création des modèles

Pour commencer, créer les modèles et leurs migrations associées :

```
php artisan make:model Adresse --migration
```

```
php artisan make:model Fournisseur --migration
```

#### Structure des tables

Ensuite, il faut créer les migrations pour déclarer les structures des tables.

Dans la table **Fournisseurs**, ajoutez simplement un identifiant, un nom, les timestamps (created\_at et update\_at)

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateFournisseursTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up() {
        Schema::create('fournisseurs', function (Blueprint $table) {
            $table->increments('id');
            $table->string('nom');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     */
}
```



```

        * @return void
        */
    public function down() {
        Schema::dropIfExists('fournisseurs');
    }
}

```

Dans la table **Adresses**, ajoutez un identifiant, l'adresse, le code postal, la ville, le pays, les timestamps (created\_at et updade\_at) et une référence à la table Fournisseur à partir de la colonne fournisseur\_id.

```

<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateAdressesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up() {
        Schema::create('adresses', function (Blueprint $table) {
            $table->increments('id');
            $table->string('adresse');
            $table->string('codePostal');
            $table->string('ville');
            $table->string('pays');
            $table->unsignedInteger('fournisseur_id');
            $table->foreign('fournisseur_id')->references('id')-
>on('fournisseurs');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down() {
        Schema::dropIfExists('adresses');
    }
}

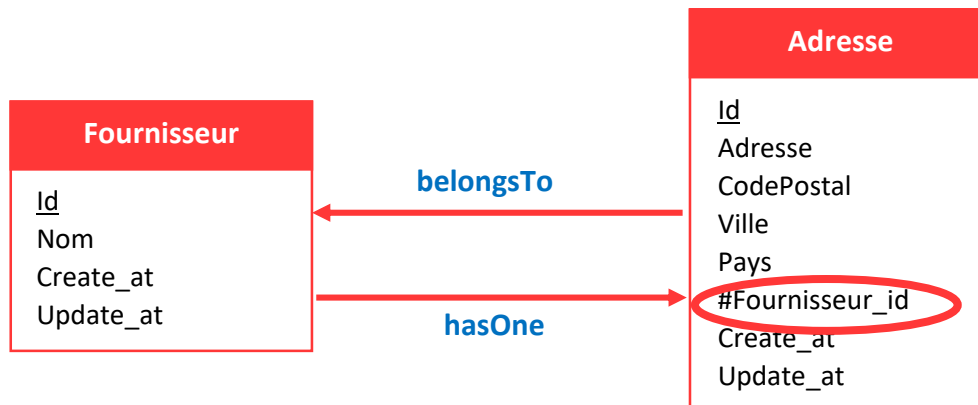
```

## Lancer les migrations

### php artisan migrate

## Déclarer les relations

Maintenant, il faut déclarer les relations dans les classes des modèles.



Je suis dans la table **Fournisseur**. Je regarde la table **adresse** et je me dis « j'ai là une **adresse** ». En langage Eloquent on traduit ça par « **Fournisseur hasOne Adresse** ».

Cette fois je me place dans la table des adresses. Là je me dis « j'ai une adresse qui appartient à un fournisseur ». En langage Eloquent on traduit ça par « **Adresse belongsTo Fournisseur** ».

Pour traduire cela de façon concrète on crée dans le modèle **Fournisseur** la méthode :

```

public function adresse() {
    //Un fournisseur possède une adresse
    return $this->hasOne(Adresse::class);
}
  
```

Et dans le modèle **Adresse** la méthode :

```

public function fournisseur() {
    //Une adresse appartient à un fournisseur
    return $this->belongsTo(Fournisseur::class);
}
  
```

## Insérer une relation

```

$fournisseur = new Fournisseur();
$fournisseur->nom = "Copag";
$fournisseur->save();

$adresse = new Adresse();
$adresse->adresse = "12, Hay El Farah";
$adresse->codePostal = "80000";
$adresse->ville = "Taroudant";
$adresse->pays = "Maroc";
$fournisseur->adresse()->save($adresse);
  
```





### Accéder aux relations

```
$fournisseur = Fournisseur::find(1);  
$adresse = $fournisseur->adresse;  
//Ensuite, on peut manipuler les adresses  
$codePostal = $adresse->codePostal;  
$ville = $adresse->ville;
```

## 10.2. La relation 1..n

Une relation un-à-plusieurs (one-to-many) est utilisée pour définir des relations dans lesquelles un seul modèle est le parent d'un ou plusieurs modèles enfants.

Par exemple, un **article** peut avoir un nombre infini de **commentaires**, par contre un commentaire est forcément lié dans un seul article.

### Création des modèles

Pour commencer, créer les modèles et leurs migrations associées :

```
php artisan make:model Article --migration
```

```
php artisan make:model Commentaire --migration
```

### Structure des tables

Ensuite, il faut créer les migrations pour déclarer les structures des tables.

Dans la table **Article**, ajoutez simplement un identifiant, un titre, un contenu, les timestamps (created\_at et update\_at)

```
<?php  
use Illuminate\Database\Migrations\Migration;  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Support\Facades\Schema;  
  
class CreateArticlesTable extends Migration  
{  
    /**  
     * Run the migrations.  
     *  
     * @return void  
     */  
    public function up()  
    {  
        Schema::create('articles', function (Blueprint $table) {  
            $table->id();  
            $table->string('titre', 255);  
            $table->text('contenu');  
            $table->timestamps();  
        });  
    }  
}
```

```
/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down() {
    Schema::dropIfExists('articles');
}
```

Dans la table **Commentaire**, ajoutez un identifiant, un texte, les timestamps (created\_at et updated\_at) et une référence à la table Article à partir de la colonne article\_id.

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateCommentairesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('commentaires', function (Blueprint $table) {
            $table->id();
            $table->text('text');
            $table->unsignedInteger('article_id');
            $table->foreign('article_id')->references('id')-
            >on('articles');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('commentaires');
    }
}
```

## Lancer les migrations

### php artisan migrate

### Déclarer les relations

Maintenant, il faut déclarer les relations dans les classes des modèles.



Je suis dans la table **Article**. Je regarde la table adresse et je me dis « j'ai beaucoup des **commentaires** ». En langage Eloquent on traduit ça par « **Article hasMany Commentaires** ».

Cette fois je me place dans la table des commentaires. Là je me dis « j'appartiens à un seul article ». En langage Eloquent on traduit ça par « **Commentaire belongsTo Article** ».

Pour traduire cela de façon concrète on crée dans le modèle **Article** la méthode :

```

public function commentaires() {
    return $this->hasMany(Commentaire::class);
}
  
```

Et dans le modèle **Commentaire** la méthode :

```

public function article() {
    return $this->belongsTo(Article::class);
}
  
```

### Insérer une relation

```

$article = new Article();
$article->titre = "Article 1";
$article->contenu = "Contenu de l'article 1";
$article->save();

$commentaire1 = new Commentaire();
$commentaire1->text = 'Parfait';
$commentaire2 = new Commentaire();
$commentaire2->text = 'Moyen';

$article->commentaires()->saveMany([ $commentaire1, $commentaire2 ]);
  
```



### Accéder aux relations

```
$articles = Article::all();  
foreach($articles as $article) {  
    $commentaires = $article->commentaires;  
    foreach($commentaires as $commentaire) {  
        $text = $commentaire->text;  
    }  
}
```

### Utilisation de where

```
$article = Article::first();  
$commentaires = $article->commentaires()->where('text','LIKE',  
    '%Parfait%')->get();
```

## 10.3. La relation n..n

Une relation plusieurs-à-plusieurs (many-to-many) permet de définir une relation entre plusieurs objets d'un côté et plusieurs objet de l'autre.

Par exemple, un **article** peut avoir plusieurs **catégories** et même catégorie peut avoir plusieurs articles.

### Création des modèles

Pour commencer, créer les modèles et leurs migrations associées :

```
php artisan make:model Article --migration
```

```
php artisan make:model Catégorie --migration
```

### Structure des tables

La relation n..n nécessite une table « pivot » qui contient une référence aux clés primaires des deux autres table.

Il faut donc créer une migration pour la table `article_categorie` :

```
php artisan make:migration create_article_categorie_table
```

Ensuite, il faut créer les migrations pour déclarer les structures des tables.

Dans la table **Article**, ajoutez simplement un identifiant, un titre, un contenu, les timestamps (`created_at` et `update_at`)

```
<?php  
use Illuminate\Database\Migrations\Migration;  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Support\Facades\Schema;
```



```
class CreateArticlesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up() {
        Schema::create('articles', function (Blueprint $table) {
            $table->id();
            $table->string('titre', 255);
            $table->text('contenu');
            $table->timestamps();
        });
    }
    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down() {
        Schema::dropIfExists('articles');
    }
}
```

Dans la table **Categorie**, ajoutez un identifiant, un titre les timestamps (created\_at et update\_at).

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateCategoriesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('categories', function (Blueprint $table) {
            $table->id();
            $table->text('titre');
            $table->timestamps();
        });
    }
}
```

```
/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
    Schema::dropIfExists('categories');
}
}
```

Dans la table **article\_categorie**, ajoutez une référence à la table **article** et une autre référence à la table **categorie**. Ensuite ajouter une clé primaire composée :

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateArticleCategorieTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up() {
        Schema::create('article_categorie', function (Blueprint
$table) {
            $table->unsignedInteger('article_id');
            $table->foreign('article_id')->references('id')-
>on('articles');
            $table->unsignedInteger('categorie_id');
            $table->foreign('categorie_id')->references('id')-
>on('categories');
            $table->primary(['article_id', 'categorie_id']);
        });
    }

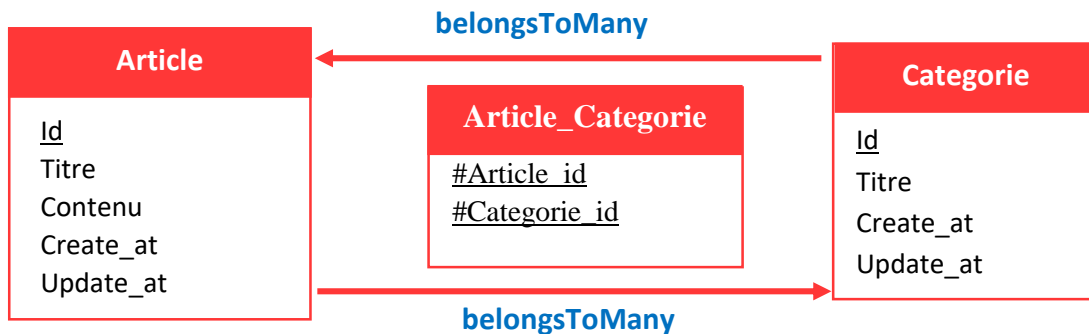
    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down() {
        Schema::dropIfExists('article_categorie');
    }
}
```

## Lancer les migrations

### php artisan migrate

### Déclarer les relations

Maintenant, il faut déclarer les relations dans les classes des modèles.



Je suis dans la table **Article**. Je regarde la table adresse et je me dis « j'appartiens à plusieurs catégories ». En langage Eloquent on traduit ça par « **Article belongsMany Catégorie** ».

Cette fois je me place dans la table des catégories. Là je me dis « j'appartiens à plusieurs articles ». En langage Eloquent on traduit ça par « **Catégorie belongsTo Article** ».

Pour traduire cela de façon concrète on crée dans le modèle **Article** la méthode :

```

public function categories() {
    return $this->belongsToMany(Catégorie::class);
}

```

Et dans le modèle **Catégorie** la méthode :

```

public function articles() {
    return $this->belongsToMany(Article::class);
}

```

### Insérer une relation

```

$article = Article::first();

$categorie1 = new Catégorie();
$categorie1->titre = "Politique";

$categorie2 = new Catégorie();
$categorie2->titre = "Sport";

$categorie3 = new Catégorie();
$categorie3->titre = "Musique";
$article->categories()->save($categorie1);

```



```
$article->categories()->saveMany([  
    $categorie2,  
    $categorie3  
]);
```

### Accéder aux relations

```
//Récupérer les catégories de premier article  
$article = Article::first();  
$categories = $article->categories;  
//Récupérer les articles liés à la première catégorie  
$categorie = Categorie::first();  
$articles = $categorie->articles;
```

### Associer et désassocier des modèles déjà existants

Dans le cas des relations n..n, il est possible d'utiliser les méthodes **attach** et **detach** qui permettent d'associer et désassocier des modèles.

```
$article = Article::first();  
$article->categories()->detach(1);  
$article->categories()->detach([2, 3]);
```

```
$article->categories()->attach(1);  
$article->categories()->attach([2, 3]);
```





## 11. Traiter les formulaires

Les formulaires sont une fonctionnalité essentielle lors du développement d'une application Web. Il est même difficile de penser à une application Web sans avoir de formulaire. La validation est un facteur critique lors de la création d'un formulaire, et sans validation appropriée, ils sont inutiles.

Un formulaire est cependant un vecteur d'attaque potentiel pour une personne malveillante. Laravel propose par défaut un mécanisme de défense face aux attaques de type CSRF (Cross-Site Request Forgery).

### 11.1. Sécuriser les formulaires

Laravel génère automatiquement un jeton de sécurité. Ce jeton CSRF est utilisé pour protéger l'application Web contre les attaques CSRF. Ces jetons contiennent une valeur unique générée par le côté serveur de l'application, qui est envoyée au côté client de l'application. En outre, cela permet de vérifier si un utilisateur authentifié envoie la demande à l'application.

Vous pouvez utiliser la directive Blade `@csrf` pour générer le champ token :

```
<form method="POST" action="/articles/create">  
    @csrf  
</form>
```

### 11.2. Valider les données

Dans une application web, la validation des données est une étape indispensable dès qu'une interaction a lieu avec les utilisateurs. Par défaut, les classes de contrôleurs de Laravel utilisent le trait **ValidatesRequest** qui met à disposition une méthode **validate** pour vérifier les données. Il est également possible et parfois plus simple d'accéder aux méthodes de validation directement via l'objet **\$request**. Dans tous les cas, on peut valider en quelques lignes les données qui proviennent d'un formulaire en écrivant quelques règles de validation.

#### Ecrire les règles de validation

Un formulaire nécessite généralement deux routes. Le premier va afficher le formulaire HTML (vue) et le deuxième traite son enregistrement et rediriger vers une vue avec un message de succès.

Dans le cas d'un contrôleur ressource, la méthode **create** est chargée d'afficher le formulaire tandis que la méthode **store** est chargée d'effectuer plusieurs actions :

- Valider les données
- Rediriger vers le formulaire en cas d'erreur et afficher les erreurs.
- Enregistrer les données s'ils sont valides dans la base de données
- Rediriger vers une page avec un message de succès.

## Exemple

```
/**
 * Store a newly created resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    $validated = $request->validate([
        'title' => 'required|max:255',
        'body' => 'required',
    ]);
}
```

Comme vous pouvez le voir, les règles de validation sont passées dans la méthode **validate** sous forme d'un tableau qui contient une liste des clés-valeurs.

Une règle de validation s'écrit en suivant les règles suivantes :

- Chaque élément de règle est séparé par le caractère « | »
- L'ensemble des éléments doit être valide pour que la règle soit valide.
- Un élément peut avoir un ou plusieurs paramètres.
- Les paramètres sont séparés par des virgules.

Exemple de règle de validation :

`required|string|email|between:10,255|unique:users`

La règle donnée en exemple indique que le champ est obligatoire, il doit être une chaîne de caractères, il doit être un e-mail, il doit faire entre 10 et 255 caractères et il doit être unique dans la table utilisateurs.

## Les règles disponibles

Règle	Leur fonction
<b>accepted</b>	Le champ sous validation doit être <i>yes</i> , <i>on</i> , ou <i>1</i> .
<b>after:date</b>	Le champ sous validation doit être après une date donnée.
<b>alpha</b>	Le champ sous validation peut uniquement contenir des lettres.
<b>alpha_num</b>	Le champ sous validation peut uniquement contenir des caractères alpha-numériques.
<b>array</b>	Le champ sous validation doit être un tableau PHP.
<b>before:date</b>	Le champ sous validation doit être une date avant la date donnée.



<b>between: <i>min,max</i></b>	Le champ sous validation doit avoir une taille entre <i>min</i> et <i>max</i> .
<b>boolean</b>	Le champ sous validation doit pouvoir être un booléen.
<b>date</b>	Le champ sous validation doit être une date valide selon la fonction PHP <code>strtotime</code> .
<b>email</b>	Le champ sous validation doit être une adresse e-mail correcte.
<b>exists: <i>table,column</i></b>	Le champ sous validation doit exister dans la base de données.
<b>image</b>	Le fichier sous validation doit être une image (jpeg, png, bmp, ou gif).
<b>in: <i>foo,bar,...</i></b>	Le champ sous validation doit être inclus dans la liste donnée de valeurs.
<b>integer</b>	Le champ sous validation doit être un entier.
<b>ip</b>	Le champ sous validation doit être une adresse IP.
<b>nullable</b>	Le champ sous validation peut être null.
<b>numeric</b>	Le champ sous validation doit être numérique .
<b>password</b>	Le champ sous validation doit correspondre au mot de passe de l'utilisateur authentifié.
<b>regex: <i>pattern</i></b>	Le filtre sous validation doit correspondre à l'expression régulière donnée.
<b>required</b>	Le champ sous validation doit être présent dans les données.
<b>size: <i>value</i></b>	Le champ sous validation doit avoir une taille correspondant à la valeur <i>value</i> .
<b>string</b>	Le champ sous validation doit être une chaîne de caractères.
<b>unique: <i>table,column, except,idColumn</i></b>	Le champ sous validation doit être unique dans la table de la base de donnée.
<b>url</b>	Le champ sous validation doit être formé comme une URL.

### Enregistrement en base de données

Après la validation des données, deux cas sont possibles. Elle peut avoir échoué ou réussi. Si la validation a réussi, il faut enregistrer les informations du formulaire en base de données.

Pour cela, on peut utiliser les données validées en récupérant le résultat de la méthode **validate**.



```
$data = $request->validate([  
    'title' => 'required|max:100',  
    'body' => 'required',  
]);  
Article::create($data);
```

Remarque : Pour rappel, la méthode **create** d'un modèle accepte uniquement les éléments qui ont été renseignés dans la propriété **\$fillable** de la classe.

### Afficher les messages d'erreur

Il existe un moyen rapide pour afficher les erreurs de la validation d'un formulaire dans la vue. Il s'agit de la variable **\$errors** qui est partagée avec toutes les vues de votre application.

```
@if ($errors->any())  
    <div class="alert alert-danger">  
        <ul>  
            @foreach ($errors->all() as $error)  
                <li>{{ $error }}</li>  
            @endforeach  
        </ul>  
    </div>  
@endif
```

### 11.3. Formulaires PATCH, PUT et DELETE

Quand on crée une API REST, on s'efforce d'utiliser les méthodes HTTP adaptées. Par exemple, pour répondre à une action de suppression d'une ressource, on préférera la méthode DELETE à la méthode POST.

Les formulaires HTML ne prennent pas en charge les actions PUT, PATCH ou DELETE. Ainsi, lors de la définition de routes PUT, PATCH ou DELETE appelées à partir d'un formulaire HTML, il faut ajouter un champ `_method` masqué au formulaire. La valeur envoyée avec le champ `_method` sera utilisée comme méthode de requête HTTP.

Laravel fournit pour cela une fonction assistante **method\_field** qui crée automatiquement un champ **input** caché avec la valeur souhaitée.

```
<form method="POST" action="/articles/{{ $article->id }}">  
    @csrf  
    @method('DELETE')  
    <button type="submit">Supprimer</button>  
</form>
```



## 11.4. Uploader de fichiers

La création d'un formulaire d'upload coté vue est identique à celle d'un formulaire classique, en utilisant le champ **input** de type **file**.

N'oubliez pas d'ajouter l'attribut **enctype="multipart/form-data"** à la balise **form** et la directive **@csrf** dans le formulaire.

### Valider les fichiers

Selon le type de fichier, vous pouvez le valider dans le contrôleur.

Valider un fichier :

```
$data = $request->validate([  
    'cv' => 'required|file',  
]);
```

Valider une image

```
$data = $request->validate([  
    'photo' =>  
    'required|image|dimensions:max_width=320,max_height=200',  
]);
```

Valider une vidéo

```
$data = $request->validate([  
    'video' => 'mimetypes:video/avi,video/mpeg',  
]);
```

La suite : En cours de réalisation