

# Développement Web en PHP

Cours de développement d'application web orienté pages en PHP

- [Mettre en place un environnement de développement PHP + MySQL](#)
- [Bases de PHP](#)
  - [Développer un formulaire d'enregistrement en MVC](#)
  - [Mettre en place un architecture MVC avec PHP](#)
  - [Traiter une soumission de formulaire en PHP](#)
- [Framework Symfony](#)
  - [Installation du framework Symfony](#)
  - [Doctrine ORM](#)
  - [Les Controller Symfony](#)
  - [Les Templates Twig](#)
  - [Premiers pas avec symfony](#)

# Mettre en place un environnement de développement PHP + MySQL

## Installer Laragon

Laragon est un outil qui package plusieurs outils pour le développement web. Nous allons l'utiliser principalement pour l'interpréteur PHP et la base de donnée MySQL. Téléchargez Laragon via [ce lien](#), puis exécutez l'installateur. Suivez ensuite les instructions d'installation.

## Configurer Laragon

Lancez Laragon puis faites *clic droit sur le fond > MySQL > Change root password*. Mettez un mot de passe et conservez le bien il servira à se connecter à la base de donnée. Lancez ensuite le serveur en faisant *clic droit sur le fond > MySQL > Start MySQL*

## Créer une base de donnée

Vous pouvez créer une base de donnée pour chacun de vos projet en faisant : *clic droit sur le fond > MySQL > Create Database*, puis donnez lui un nom.

## Installer JetBrains PhpStorm et Datagrip

### Licence

PhpStorm et Datagrip sont des outils professionnels sous licence. Heureusement, JetBrains permet aux étudiants de bénéficier de licences gratuites pour tous ses outils. Elles sont accessible via le [Github Student Pack](#).

## Avec JetBrains Toolbox

Jetbrains Toolbox est un petit utilitaire très pratique qui permet d'installer et mettre à jours les logiciels JetBrains en un clic.

## Installer JetBrains Toolbox

Téléchargez et installez [Jetbrains Toolbox](#)

## Installer PhpStorm

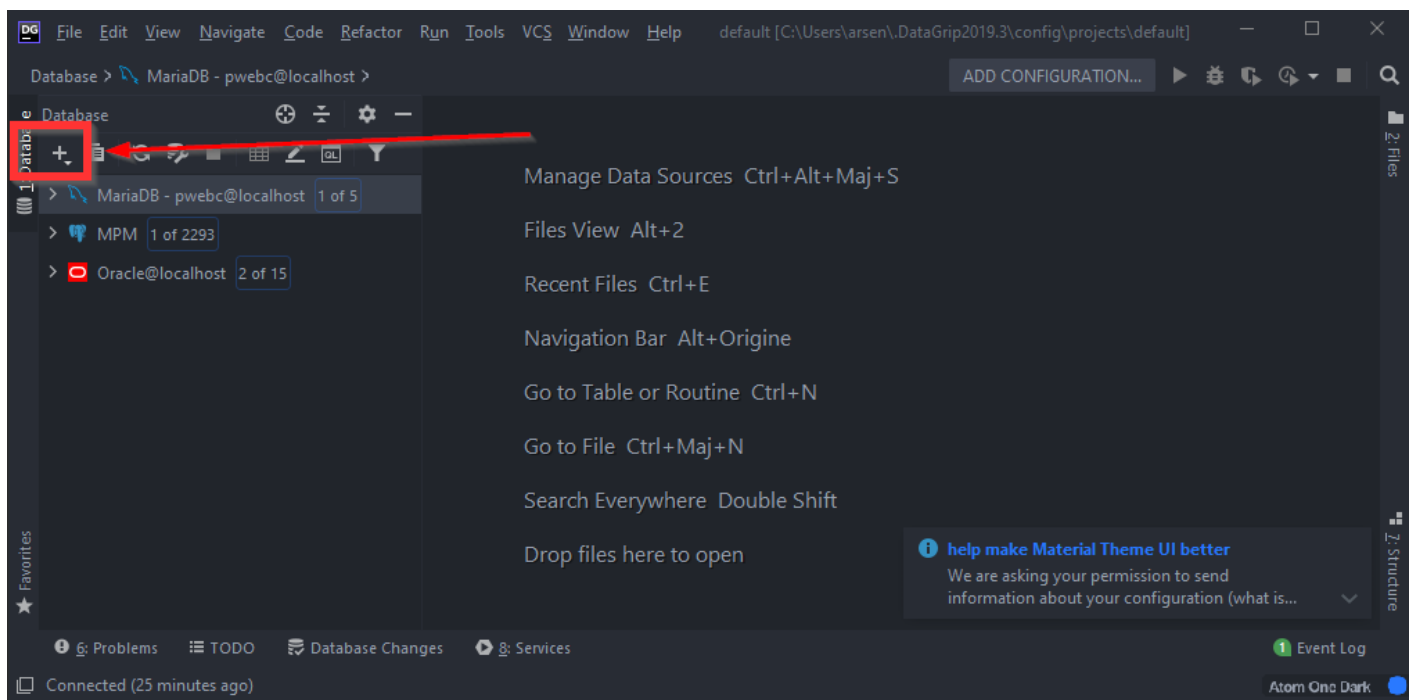
Exécutez JetBrains Toolbox puis faites un clic droit sur son icône dans la barre d'état système de Windows. Trouvez ensuite PhpStorm et Datagrip dans la liste des applications proposée. Cliquez sur "Install" et attendez la fin du téléchargement.

## Sans JetBrains ToolBox

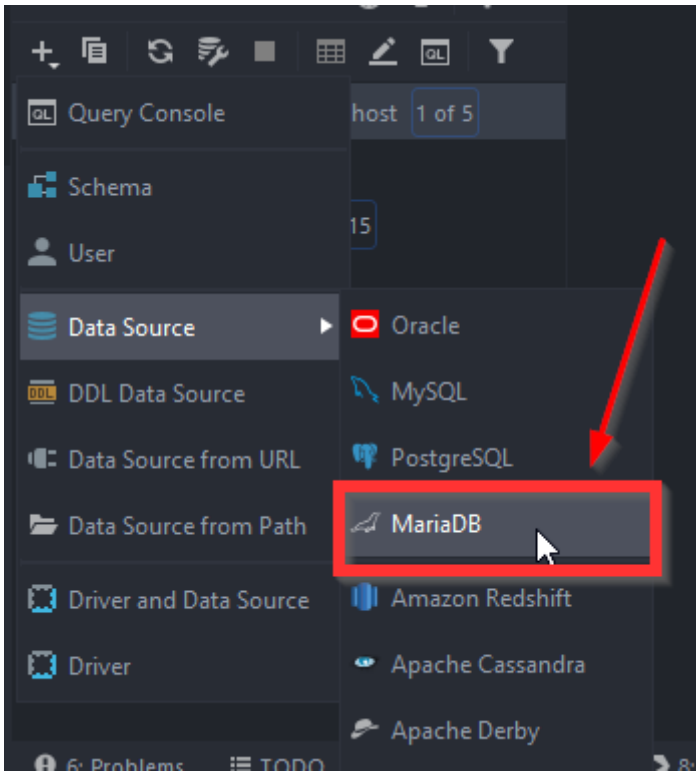
Téléchargez PHP Storm via [ce lien](#) et datagrip via [ce lien](#), puis exécutez l'installateur. Suivez ensuite les instructions d'installation.

# Se connecter à une base de donnée avec Datagrip

Lancer Datagrip puis cliquer sur le bouton "+" :



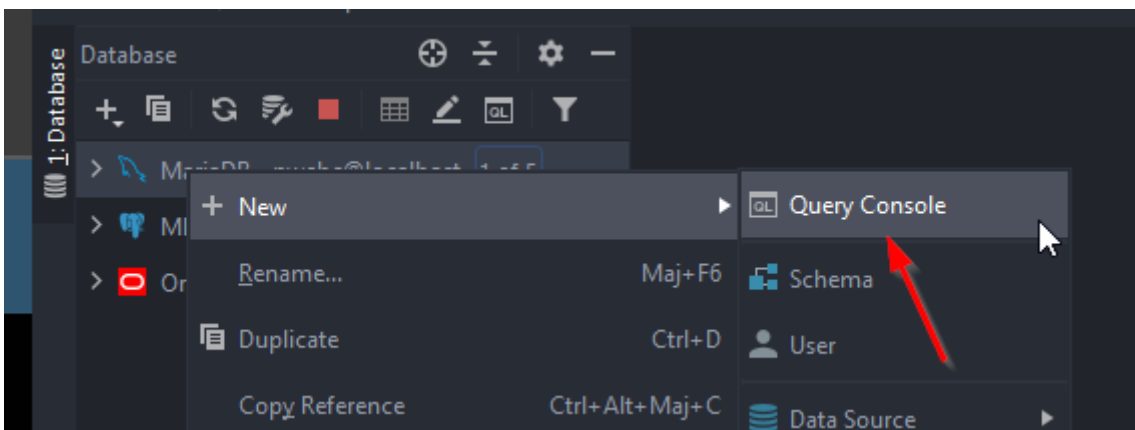
Et choisissez "Datasource > MariaDB" :



Au bas du formulaire cliquez sur "*Download missing driver file*". Remplissez ensuite le formulaire comme suit :

- Name : nom de votre projet
- Host : laissez
- port : laissez
- User :
- Password : Le mot de passe que vous avez configuré sur la base de donnée MySQL de Laragon
- Database : Le nom de la base de donnée que vous avez créé sur Laragon

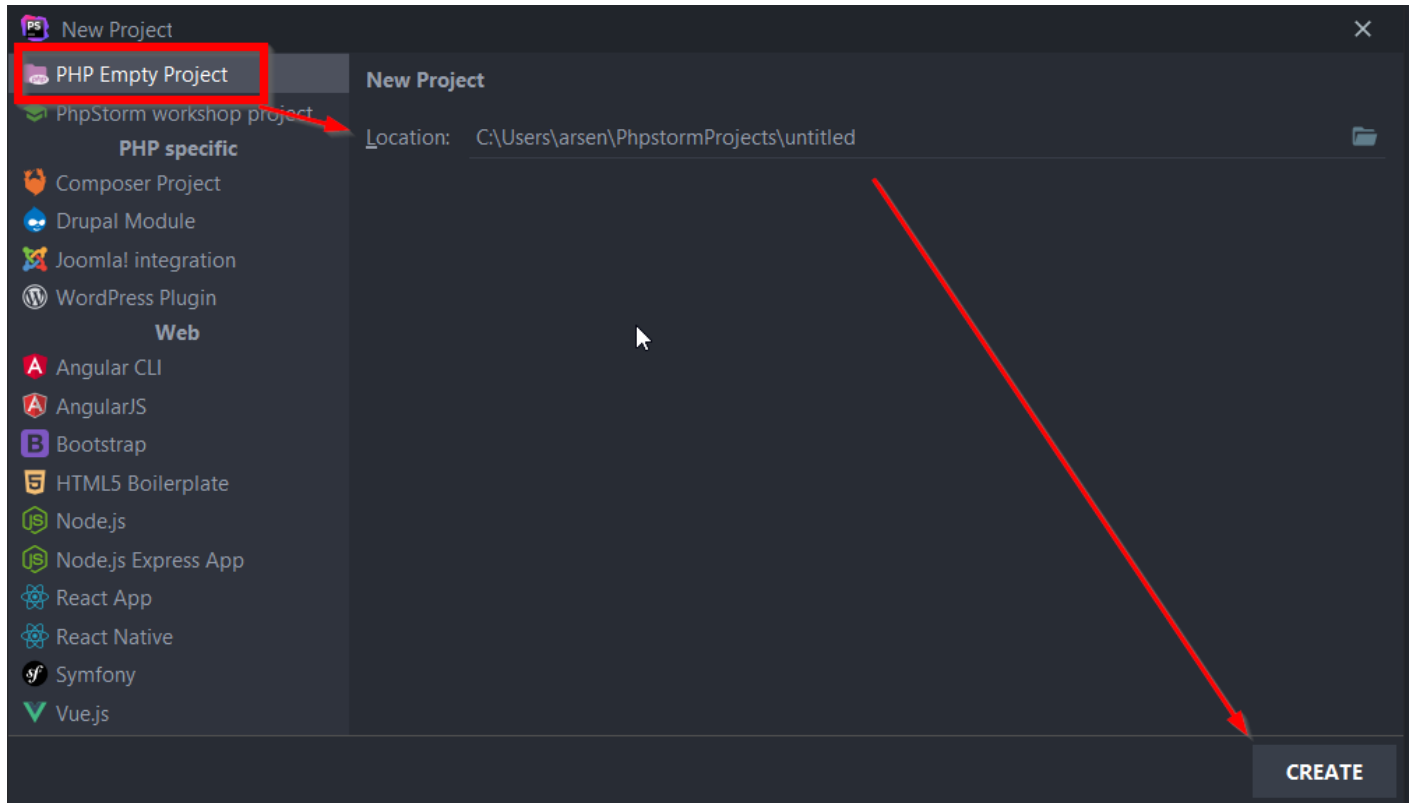
Faites ensuite "*Test Connection*" afin de voir si tout est bien configuré, puis fait "*Apply*". Votre base de donnée est apparue dans la liste des bases de donnée à gauche de l'interface. Pour ouvrir une session SQL faites *Clic droit* sur le nom de votre base > *New* > *Query Console* :



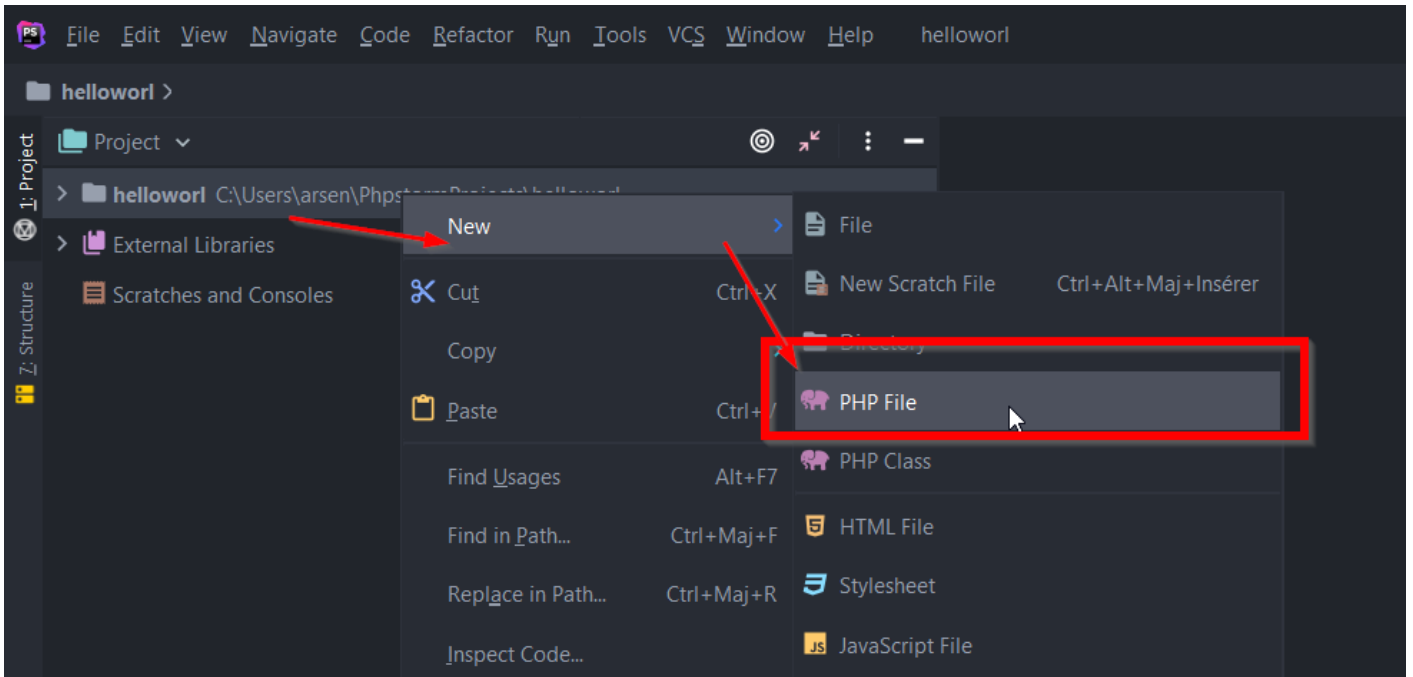
Vous avez donc une console dans laquelle vous pouvez écrire des scripts SQL. `CTRL + ENTER` pour exécuter votre script.

# Créer et configurer un projet sur PhpStorm

Lancez PhpStorm et sélectionnez "Create new Project" puis choisissez "PHP Empty Project", paramétrez l'emplacement de votre projet puis faites "Create" :



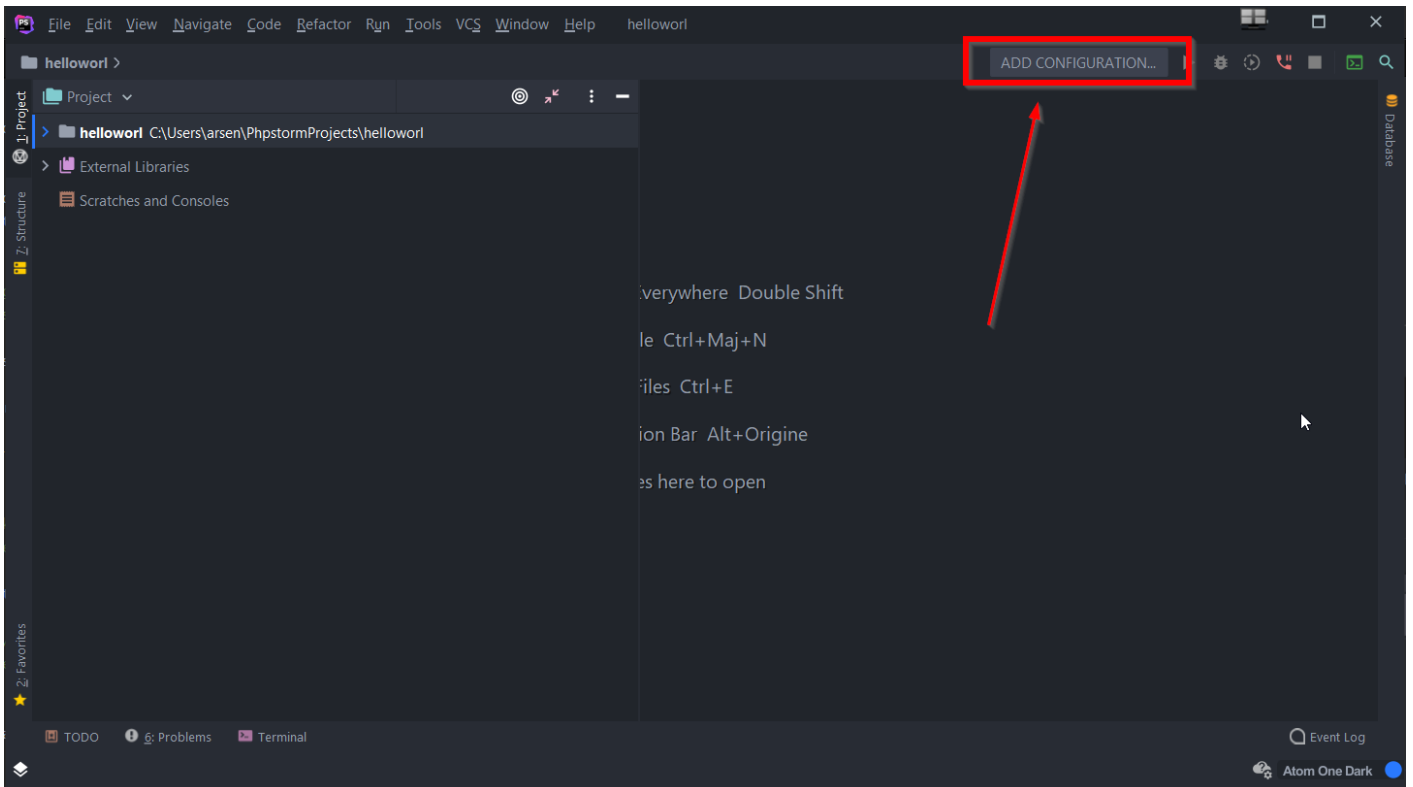
Ensuite ajoutez un fichier PHP à votre projet :



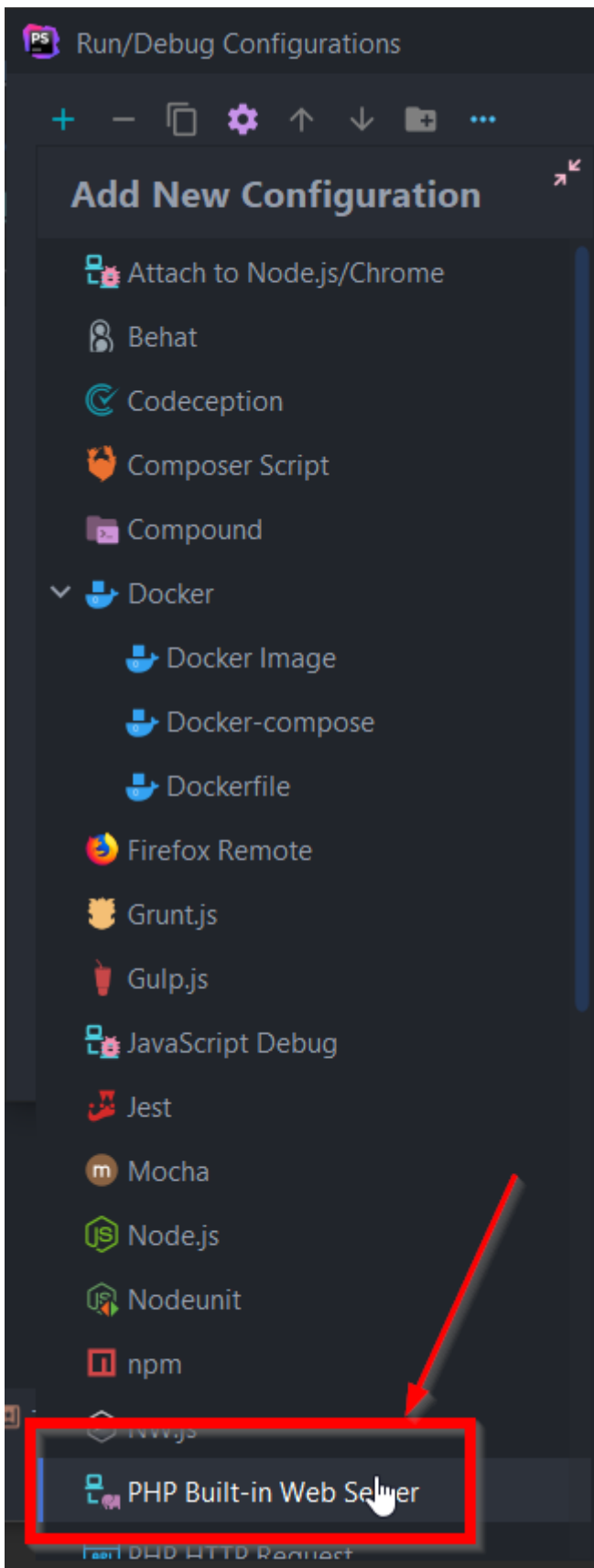
Appellez le `index`, et écrivez le code suivant :

```
echo "Hello, World !";
```

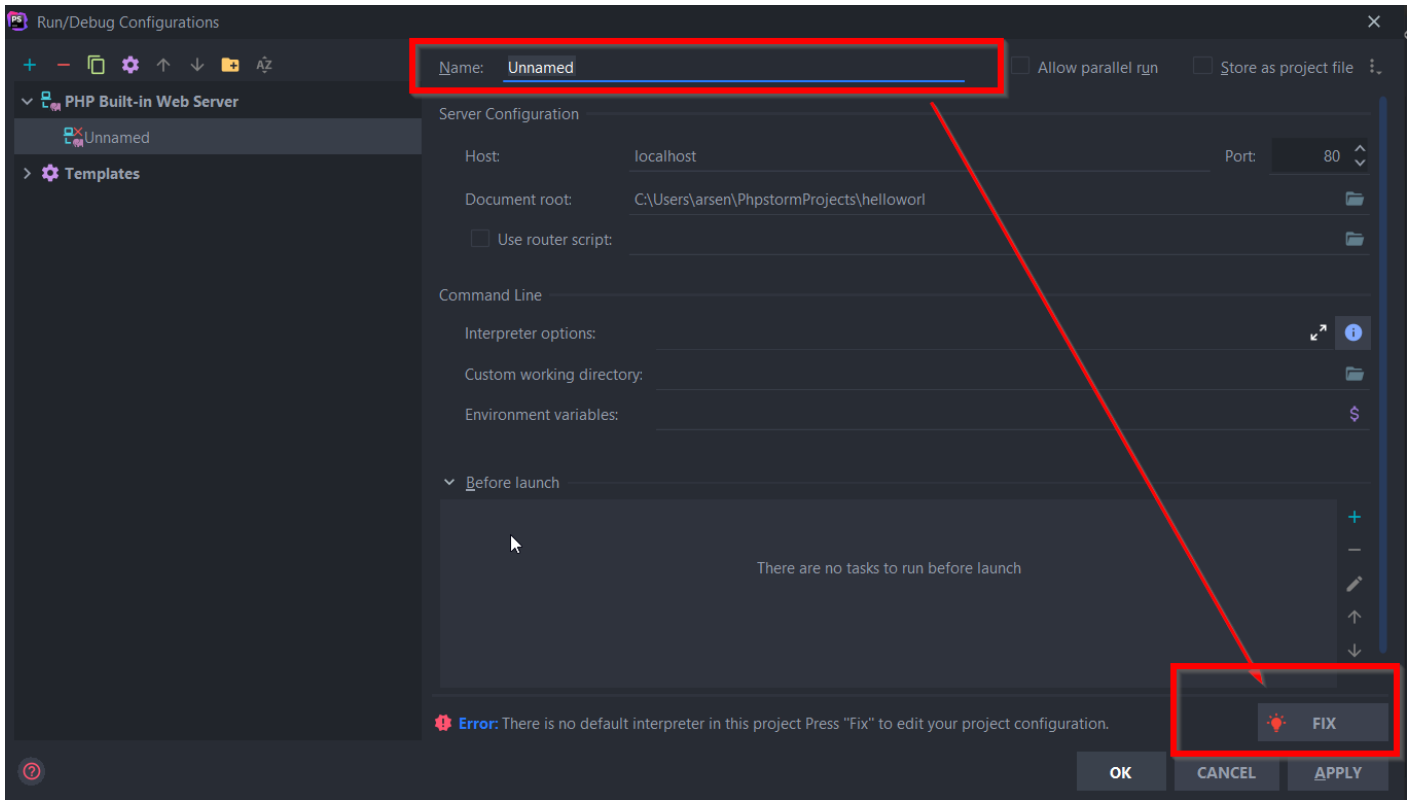
Ensuite pour exécuter votre code, créez une configuration en appuyant sur "Add Configuration" :



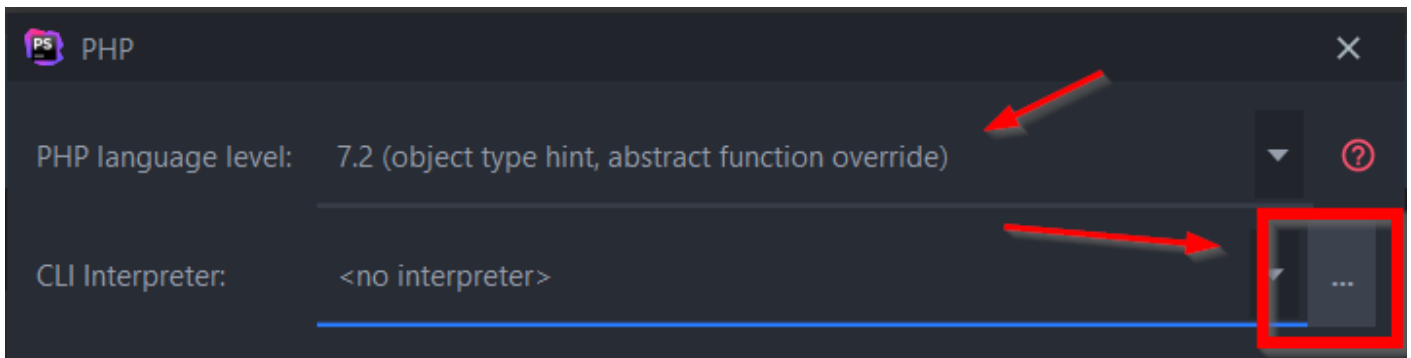
Et choisissez "PHP Built-In Server" :



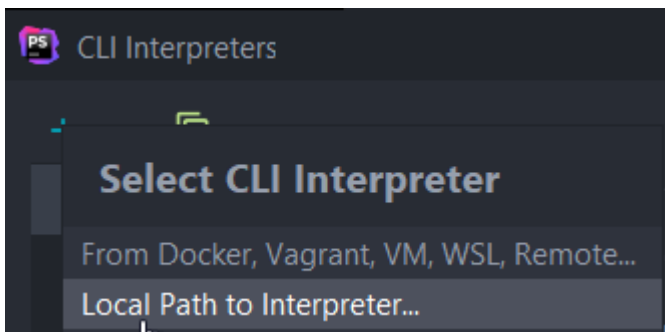
Donnez un nom (peu importe lequel) à votre configuration dans le premier champs du formulaire, puis appuyez sur "Fix" pour configurer l'interpréteur PHP :



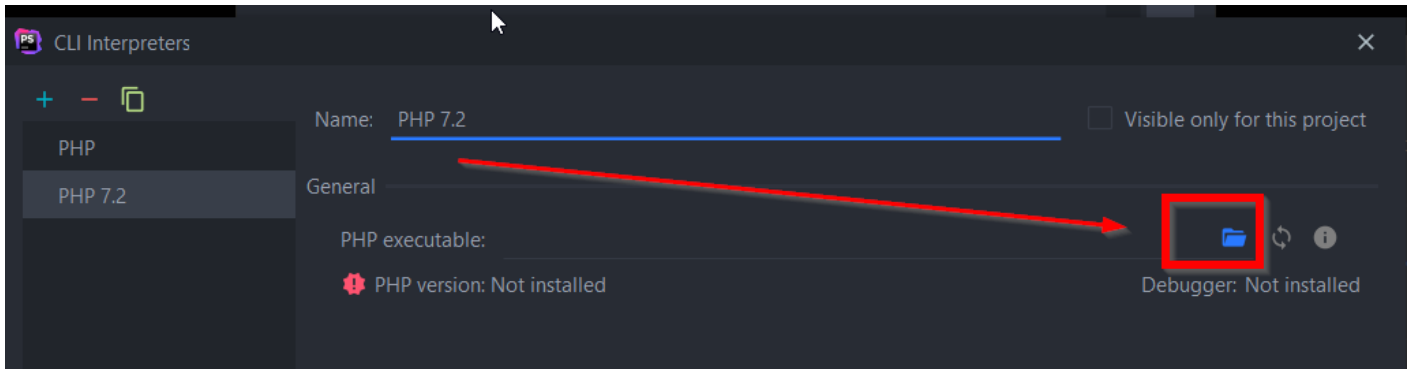
Paramétrez la version de PHP sur 7.2, puis sur "... " pour créer un nouvel interpréteur avec cette version :



Cliquez sur "+" puis sélectionnez l'option "Local Path to Interpreter" :



Ensuite, donnez un nom à votre Interpréteur (peu importe lequel) puis cliquez sur l'icone de dossier pour fournir le chemin de l'interpréteur :



Le chemin à partir du répertoire d'installation de Laragon est : `bin/php/php.exe` . Par défaut Laragon est installé dans `C:/laragon` donc le chemin par défaut est : `c:/laragon/bin/php/php.exe` . Faites ensuite "Apply" et "Ok" sur toutes les fenêtrte ouvertes par dessus PhpStorm.

Vous pouvez ensuite faire `Shift + F10` pour démarrer le serveur PHP ou alors utilisez le bouton Run. Ceci fait, ouvrez votre navigateur à l'adresse `http://localhost` . Si vous constatez une page blanche avec "Hello World !" écrit, c'est que tout marche bien et que votre code s'est bien exécuté !

# Bases de PHP

# Développer un formulaire d'enregistrement en MVC

Nous allons maintenant utiliser l'architecture MVC pour développer une fonctionnalité simple d'enregistrement d'un utilisateur dans une base de donnée d'après un formulaire.

## Vue

Commençons par créer notre vue, le formulaire, dans un fichier `register.tpl` dans le dossier `view` :

```
<form method="post" action="index.php?controller=registerController&action=register">

    <label for="lastname">Nom :
        <input type="text" name="lastname" required/>
    </label>
    <br>
    <label for="firstname">Prénom :
        <input type="text" name="firstname" required/>
    </label>
    <br>
    <label for="number">Numéro :
        <input type="text" name="number" required/>
    </label>
    <br>
    <label for="role">Rôle :
        <input type="text" name="role" required/>
    </label>
    <br>
    <label for="email">Email :
        <input type="text" name="email" required/>
    </label>

    <button type="submit">Inscription</button>
</form>
```

# Modèle

On va maintenant créer un service de données : `registerService.php` dans le dossier `model`. Dans ce script, on va créer une fonction `registerUser` dans lequel on importe le script de connexion à la base de donnée pour accéder à l'objet PDO via notre fonction `connect()`. La fonction `registerUser` prend en paramètre les données correspondant au formulaire pour les insérer dans la base de donnée.

```
function registerUser($firstname, $lastname, $number, $role, $email)
{
    require("connectDb.php");
    $db = connect();
    $request = $db->prepare("INSERT INTO utilisateur(nom,prenom,num,role,email) value
(:firstname, :lastname, :number,:role, :email)");
    $request->bindParam(":firstname", $firstname);
    $request->bindParam(":lastname", $lastname);
    $request->bindParam(":number", $number);
    $request->bindParam(":role", $role);
    $request->bindParam(":email", $email);

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

# Controleur

On va finalement créer un fichier `registerController.php` dans le dossier `controller`. Dans ce fichier, une fonction `register` va gérer l'affichage de l'interface ainsi que la soumission du formulaire.

```
function register()
{
    if (isset($_POST["lastname"]) && isset($_POST["firstname"]) && isset($_POST["number"]) &&
isset($_POST["role"]) && isset($_POST["email"])) {
        $lastname = $_POST["lastname"];
        $firstname = $_POST["firstname"];
        $number = $_POST["number"];
        $role = $_POST["role"];
        $email = $_POST["email"];

        require("../model/registerService.php");
    }
}
```

```
        registerUser($firstname,$lastname, $number,$role,$email);  
    }  
  
    require("./view/register.tpl");  
}
```

Si les données sont présentes, c'est que c'est une soumission du formulaire, sinon c'est simplement une requête de la page du formulaire, on doit donc faire une condition pour vérifier ça.

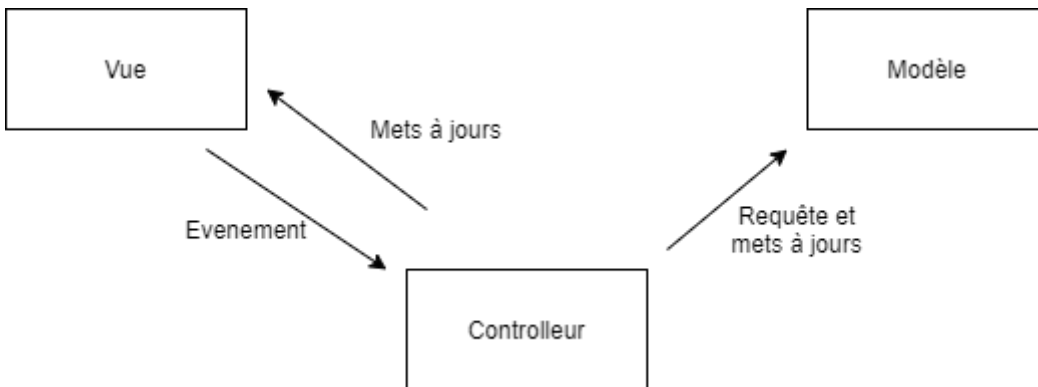
Voilà, vous avez un formulaire fonctionnel avec votre architecture MVC, félicitation

# Mettre en place un architecture MVC avec PHP

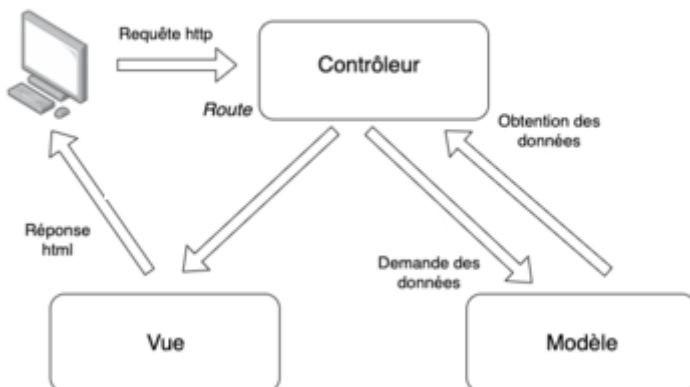
## Le Modèle MVC

Le modèle MVC pour : Model - View - Controller est un Design Pattern visant à séparer une application en trois modules :

- Modèle : les données
- Contrôleur : traite les données et mets à jours la vue
- Vue : Définit l'interface utilisateur



Si on applique au modèle du Web cela donne le schéma suivant :



## MCV en PHP Vanilla

Commeçons par créer une arborescence de dossiers pour ranger nos scripts en fonction des différents modules de MVC :

- Un dossier `model` : contient des scripts qui contiennent des fonctions de requêtes SQL ainsi qu'un script de connexion à la base de donnée.
- Un Dossier `view` : va contenir des fichier `.tpl` qui à partir desquels seront générées des pages HTML.
- Un dossier `controller` : contient des scripts qui vont faire appel aux fonctions du `model` afin de récupérer des données, les traiter, puis faire appel à la vue correspondante.

et un fichier `index.php` pour le routage.

## Le routage

Pour accéder aux service des controleurs on va passer dans chaque requêtes une variable `controller` et une variable `action` en paramètre d'URL. Exemple :

`http://localhost/index.html&controller=user&action=login`. Si bien qu'on fera toujours appel à `index.html`, qui lui s'occupera d'appeler la bonne action du bon controleur. Le code qui s'en occupe est le suivant :

```
if (isset ( $_GET ['controller'] ) & isset ( $_GET ['action'] )) {
    []$controle = $_GET ['controller'];
    []$action = $_GET ['action'];
} else { // absence de paramètres : prévoir des valeurs par défaut
    []$controle = "user";
    []$action = "login";
}

// Inclure le fichier php de contrôle
// Et lancer la fonction-action issue de ce fichier.

require ('./controller/' . $controle . '.php');
$action ();
```

## Connexion à la base de donnée

On va créer un script de connexion à la base de donnée `connectDb` dans le dossier `model` et on y écrit la fonction suivante :

```
function connect(){
    $hostname = "localhost";
    $base = "nom de la base de donnée";
    $loginBD = "root";
    $passBD = "mot de passe de la base de donnée";
```

```
try {
    $bdd = new PDO ( "mysql:server=$hostname; dbname=$base", "$loginBD", "$passBD", array
(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8'));
    return $bdd;
}
catch ( PDOException $e ) {
    echo "Echec de connexion à la base de donnée";
    die ( "Echec de connexion : " . $e->getMessage () . "\n" );
}
}
...
```

# Traiter une soumission de formulaire en PHP

En PHP, la manière principale de communiquer des données au serveur est pas la soumission de formulaire. Voyons un exemple simple avec un formulaire de connexion.

## Le formulaire

Commençons déjà par écrire notre formulaire dans un document HTML : `login.html`

### Base du formulaire

```
<form method="post" action="login.php">

</form>
```

Nous allons utiliser la méthode HTTP POST car nous devons envoyer des données au serveur. Ensuite dans l'attribut `action` nous mettons le chemin sur le serveur de notre script PHP qui va gérer la soumission du formulaire.

### Les champs

Ajoutons maintenant les champs dont nous avons besoin pour la connexion : un champs de type *text* pour le login et un champs de type *password* pour le mot de passe, ainsi qu'un bouton de soumission.

```
<form method="post" action="login.php">

  <input type="text" name="login" required/>
  <input type="password" name="password" required/>

  <button type="submit">Connexion</button>

</form>
```

L'attribut `required` permet d'empêcher la soumission du formulaire si le champs n'est pas rempli. Chaque champs possède une attribut `name` pour l'identifier lors du traitement dans le script PHP.

Voilà notre formulaire est prêt !

# Le script PHP

Créez maintenant un script `login.php` afin de traiter la soumission du formulaire.

## Récupération des données

Afin de récupérer des données envoyées via une requête POST, il faut faire appel dans le code PHP à la variable globale `$_POST`. `$_POST` (comme `$_GET`) est un dictionnaire clé valeur (ou tableau associatif) qui contient les valeurs des données passées dans le formulaire, avec pour clé, l'attribut `name` du champs dans le formulaire.

Mais avant de récupérer les valeurs, il faut vérifier qu'elles sont bien présentes grâce à la fonction `isset` :

```
if(!(isset($_POST["login"]) && isset($_POST["password"]))){\n    Header("Location: index.html");\n}
```

Si ce n'est pas le cas, on utilise `Header` pour rediriger vers le formulaire. On peut ensuite récupérer les valeurs des champs à partir de leurs noms :

```
$login = $_POST["login"];\n$password = $_POST["password"];
```

Après avoir récupéré ces données, on va pouvoir les valider, mais pour cela, il faut se connecter à une base de donnée !

## Connexion à une base de donnée

Commencez par créer une table et des données sur votre base de donnée en exécutant le script suivant sur datagrip :

```
CREATE TABLE utilisateur (\n    login VARCHAR(255),\n    password VARCHAR(255)\n);\nINSERT INTO utilisateur(login,password) VALUES ("JohnTest","Test@2020");
```

Pour se connecter à une base de donnée MySQL avec PHP, il faut utiliser PDO :

```
$hostname = "localhost";
$base = "nom de la base de donnée";
$loginBD = "root";
$passBD = "mot de passe de la base de donnée";

$bdd = new PDO ( "mysql:server=$hostname; dbname=$base", "$loginBD", "$passBD", array
(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8'));
```

PDO (PHP Database Object) est orienté objet, c'est pour cela qu'on l'instancie avec le mot clé `new` et qu'on va devoir utiliser l'opérateur `->` dessus.

Pour éviter les injections SQL, il faut utiliser une requête préparée. Cela se fait avec la fonction `prepare` en utilisant des étiquettes dans la requête SQL :

```
$requete = $bdd->prepare ( "SELECT * FROM utilisateur WHERE login=:login AND
password=:password" );
```

On peut ensuite lier les paramètres avec la méthode `bindParam` :

```
$requete->bindParam ( ":login", $login );
$requete->bindParam ( ":password", $password );
```

Et enfin, on exécute la requête :

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

On peut ensuite récupérer la première ligne du résultat de la requête sous la forme d'un dictionnaire clé valeur (ou tableau associatif) avec la méthode `fetch()` :

```
$resultat = $resultatRequete->fetch();
```

Si la requête ne contient plus aucune ligne, on obtiens `false`. On peut donc vérifier que notre utilisateur est valide en faisant :

```
if($resultat){
    echo "Vous êtes connecté";
}
else {
    echo "Mauvais identifiants";
}
```

Voilà votre page de connexion est fonctionnelle, félicitations

# Framework Symphony

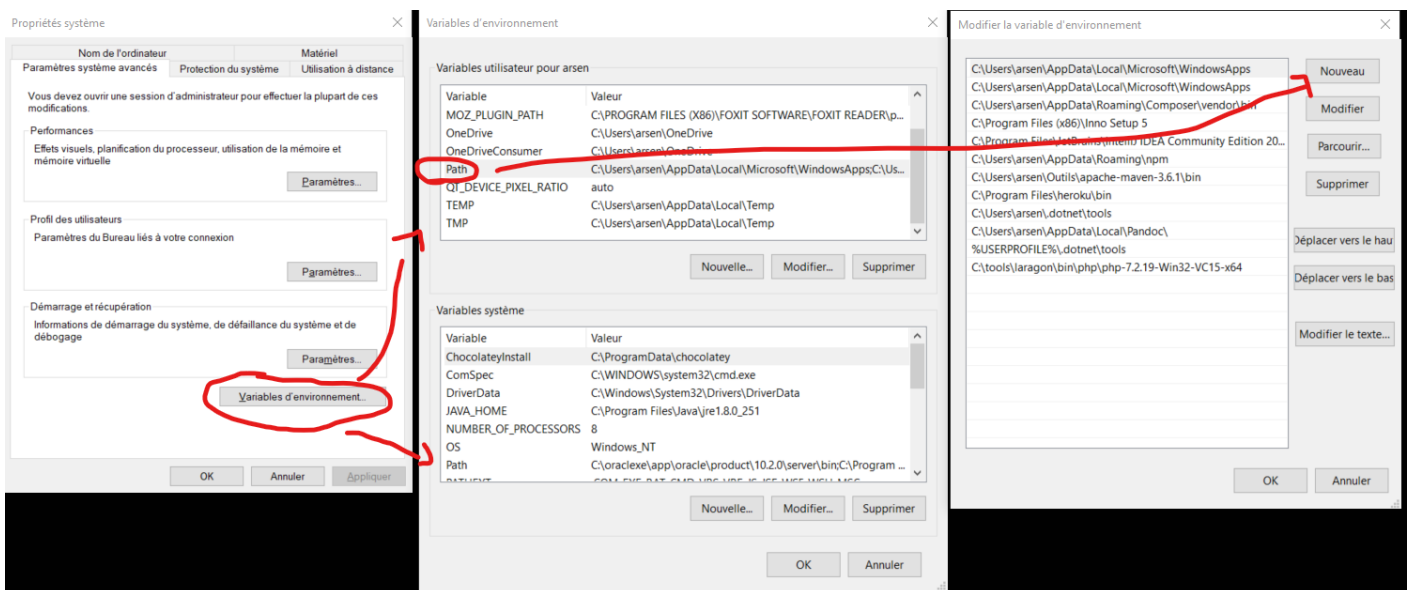
# Installation du framework Symfony

Symfony est un framework d'application moderne qui permet d'écrire des applications web en php, tout en ayant une expérience de développement moderne et agréable.

## Installation

### Ajouter PHP au PATH

Copier le chemin de votre exécutable PHP : `bin\php\php-7.2.19-Win32-VC15-x64` à partir du répertoire d'installation de Laragon. Dans la recherche Windows, rechercher "variables" et ouvrir la première options. Cliquez sur "Variables d'environnement systèmes" et ajoutez le chemin de PHP à la variable PATH pour l'utilisateur et le système.



### Installer Composer & Symfony CLI

Composer est un gestionnaire de package pour PHP, (à la manière de maven / nugget / pip ...). Pour l'installer créer un dossier "Composer" dans votre dossier d'outils (le même où il y a le répertoire d'installation de Laragon, par exemple). Dans ce dossier, ouvrez une invite de commande :

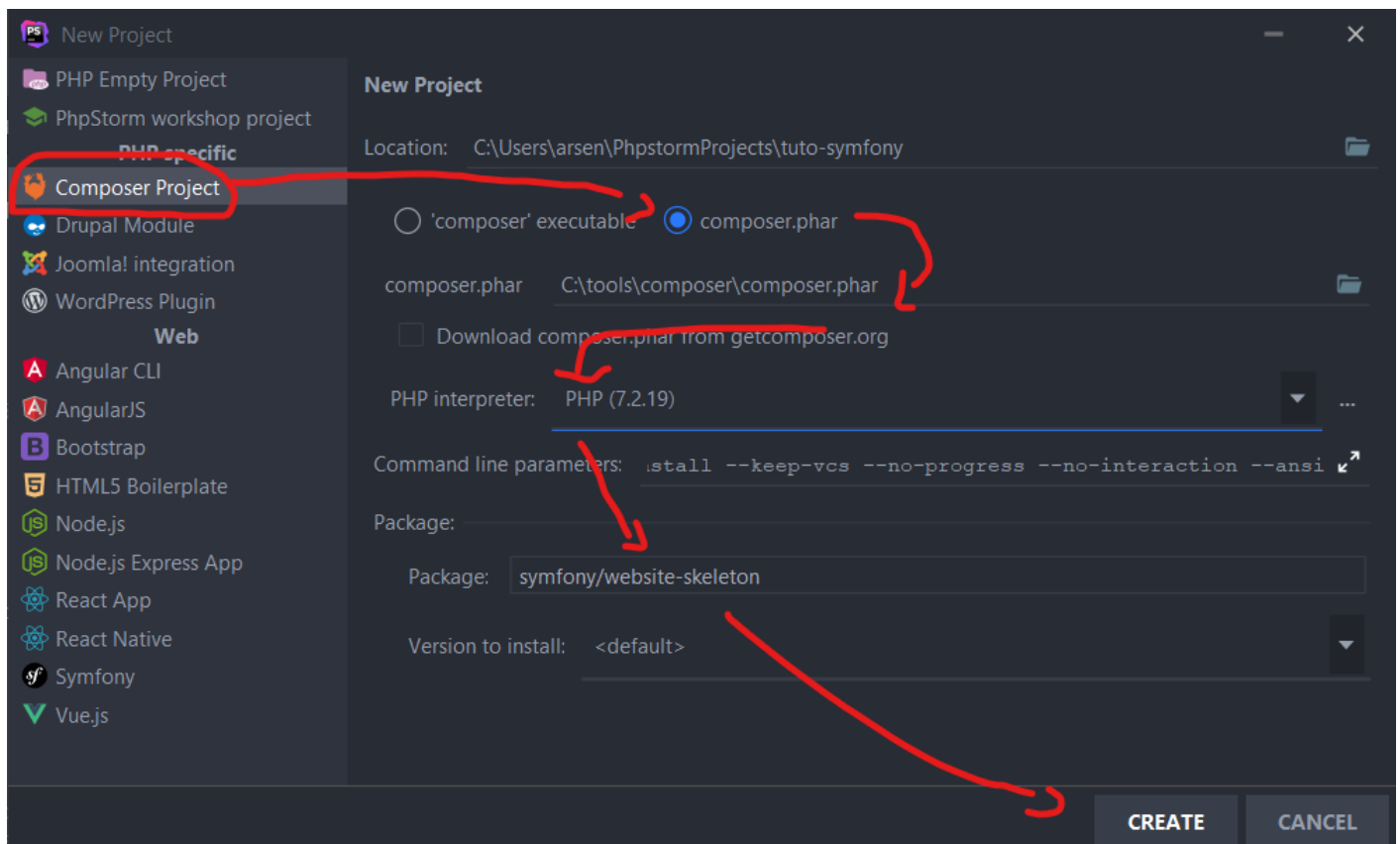
```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') ===
'756890a4488ce9024fc62c56153228907f1545c228516cbf63f885e036d37e9a59d27d63f46af1d4d07ee0f76181c
7d3') { echo 'Installer verified'; } else { echo 'Installer corrupt'; unlink('composer-
setup.php'); } echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
echo @php "%~dp0composer.phar" %*>composer.bat
```

Téléchargez Symfony CLI via [ce lien](#). Installez le.

## Créer un projet Symfony sur PHP Storm

Créer un nouveau projet Composer :

- Choisir "*composer.phar*"
- Renseigner le chemin de votre "*composer.phar*" qui se trouve dans le dossier où vous avez installé Composer
- Renseigner votre interpréteur PHP qui est normalement déjà configuré donc vous n'avez qu'à le choisir dans le menu
- Dans "*Package*" choisissez `symfony/website-skeleton`



Enfin attendez que le projet se génère.

# Run le projet

Ouvrez le Terminal de PHP Storm et tapez la commande :

```
symfony server:start
```

Rendez vous ensuite à l'adresse <http://127.0.0.1:8000> Si une page "welcome to symfony" s'affiche, tout fonctionne correctement, vous êtes prêt à coder !

# Doctrine ORM

Doctrine est un framework de Mapping Relationnel-Objet (ORM en anglais). Une tel framework permet de traduire des données représentée dans un modèle orientée objet (utilisé dans les langages de proframmmation OO) vers un modèle relationnel (utilisée des les systèmes de gestion de bases de données relationnels). Les ORMs permettent de gagner beaucoup de temps dans une application qui manipule des données, en déchargeant le programmeur d'une grande quantité de code très redondant.

## Installation de Doctrine ORM

Installez le package Doctrine via Composer :

```
composer require symfony/orm-pack
composer require --dev symfony/maker-bundle
```

Ensuite, pour configurer les accès à la base de donnée, ouvrez le fichier `.env` à la racine de votre projet et cherchez le paramètre `DATABASE_URL`, et assignez lui la valeur suivante :

```
mysql://db_user:db_password@localhost:3306/db_name?serverVersion=5.7
```

en remplaçant `db_user`, `db_password`, et `db_name` par les valeurs correspondant à votre base.

## Créer une Entity

Une Entity est une classe PHP dont les données seront persistées dans dans la base de donnée par Doctrine. Pour en créer, il existe un utilitaire en ligne de commande. Pour y faire appel, utilisez :

```
php bin/console make:entity
Class name of the entity to create or update:
> Product
```

```
New property name (press <return> to stop adding fields): > name
```

```
Field type (enter ? to see all types) [string]: > string
```

```
Field length [255]: > 255
```

```
Can this field be null in the database (nullable) (yes/no) [no]: > no
```

```
New property name (press <return> to stop adding fields): > price
```

```
Field type (enter ? to see all types) [string]: > integer
```

```
Can this field be null in the database (nullable) (yes/no) [no]: > no
```

```
New property name (press <return> to stop adding fields): > (press enter again to finish)
```

L'outil en ligne de commande va ensuite vous demander les champs de votre entité un par un, ainsi que les informations à propos de vos champs (type, taille, etc ...).

Quand vous aurez fini, l'outil va vous créer une classe sous `src/Entity`:

```
namespace App\Entity;
```

```
use App\Repository\ProductRepository; use Doctrine\ORM\Mapping as ORM;
```

```
/**

@ORM\Entity(repositoryClass=ProductRepository::class) / class Product { /*

@ORM\Id()
@ORM\GeneratedValue()
@ORM\Column(type="integer") */ private $id;

/**

@ORM\Column(type="string", length=255) */ private $name;

/**

@ORM\Column(type="integer") */ private $price;

public function getId(): ?int { return $this->id; }

// ... getter and setter methods }
```

Vous pouvez constater que les infos que vous avez données à l'outil ont été converties en annotations, qui vous ensuite permettent à l'ORM de sérialiser la classe dans une base de données relationnelle.

## Migrations

Pour pouvoir utiliser l'ORM, il faut d'abord synchroniser le schéma de votre base de données avec vos Entity PHP. Pour cela il faut créer puis exécuter une migration.

Créer une migration :

```
php bin/console make:migration
```

Exécuter une migration :

```
php bin/console doctrine:migrations:migrate
```

Au moment de la migration, Symfony va exécuter le SQL nécessaire pour mettre à jours son schéma de façon à gérer les Entities.

## Persister des données

Pour faire appelle à la couche de persistance dans le controller, il faut faire appel à l'Entity Manager de doctrine. Vous pouvez t accéder dans un controlleur comme ceci :

On peut ensuite créer un objet Entity :

```
$product = new Product();  
$product->setName('Vis en acier inoxydable');  
$product->setPrice(1999);  
$product->setDescription('6mm de diamètre');
```

On peut ensuite persister l'Entity :

```
$entityManager->persist($product);
```

Et enfin commiter la transaction :

```
$entityManager->flush();
```

## Requêter des données

Pour récupérer des données via Doctrine, cela se passe avec le Repository de l'Entity, que vous obtenez ainsi :

```
$repository = $this->getDoctrine()->getRepository(Product::class);
```

Vous pouvez ensuite récupérer des Entities par Id :

```
$product = $repository->find($id);
```

Vous pouvez ensuite filtrer sur un ou plusieurs propriétés avec la méthode `findOneBy()` et en lui passant un tableau clé-valeur avec en clé le nom du champs à filtrer et en valeur la valeur sur laquelle filtrer. Exemple :

```
$product = $repository->findOneBy([
    'name' => 'Keyboard',
    'price' => 1999,
]);
```

## Mettre à jours des données

Pour mettre à jours une Entity, il suffit de la récupérer avec le Repository, de la modifier puis de commiter les changements avec `$entityManager->flush()` :

```
$entityManager = $this->getDoctrine()->getManager();
$product = $entityManager->getRepository(Product::class)->find($id);
$product->setName('Nouveau nom');
$entityManager->flush();
```

# Les Controller Symfony

Symfony repose sur le modèle MVC et utilise donc des Controllers. Nous en avons vu les bases dans la partie précédente, mais nous allons revenir dessus dans cette partie pour présenter certaines fonctionnalités.

## Génération d'URLs et Redirections

Dans la partie précédente, nous avons nommé nos routes. C'est ici que cela va devenir utile. En effet on peut utiliser le nom d'une route pour générer une URL de cette route, en remplissant ses paramètres.

Reprenons cet exemple de route paramétrée :

```
□ /**
 * @Route("/blog/{id}", name="blog_show", methods = {"GET"})
 */
public function show(int $id) { ... }
```

On peut donc générer une URL qui correspond à cette route comme ceci :

```
$url = $this->generateUrl('blog_show', ['id' => 10]);
```

On peut également faire des redirections vers des routes :

```
return $this->redirectToRoute('blog_show', ['id' => 10]);
```

Ou vers des URLs :

```
return $this->redirect('http://symfony.com/doc');
```

## Session

On peut également, via un Controller Symfony, accéder à la session et y lire/écrire des données. Pour cela, il suffit de prendre en paramètre de la méthode endpoint un paramètre de type `SessionInterface` qui sera fourni par le framework.

```
□ /**
 * @Route("/auth/login", name="login", methods = {"POST"})
```

```
*/  
public function login(SessionInterface $session) { ... }
```

Il s'agit d'un tableau associatif avec le lequel vous pouvez interagir via les méthodes `set` et `get` :

```
□ /**  
 * @Route("/auth/login", name="login", methods = {"POST"})  
 */  
public function login(SessionInterface $session) {  
    $session->set('key', 'value');  
    $value = $session->get('key');  
}
```

## L'objet Request

L'objet Request qui peut être fourni en paramètre d'une méthode endpoint par le framework, permet d'accéder aux données de la requête :

```
public function myEndpoint(Request $request){ ... }
```

Il permet de récupérer des paramètres du Body de la requête (équivalent `$_POST`) :

```
$request->request->get('myParam');
```

Ou des paramètres GET (équivalent `$_GET`) :

```
$request->query->get('myUrlParam');
```

Vous pouvez aussi récupérer un fichier dans le Body de la requête (équivalent `$_FILES`) :

```
$request->files->get('myFile');
```

## Retourner un fichier

Au lieu de retourner un Template rendu, vous pouvez aussi retourner un fichier qui sera téléchargé au client :

```
return $this->file('/chemin/du/fichier.pdf');
```

# Les Templates Twig

Twig est un moteur de template HTML qui permet d'écrire des templates de pages HTML avec une syntaxe plus légère et plus lisible que du PHP Vanilla.

Les trois syntaxes principales de twig sont :

- `{{ ... }}` pour afficher une variable passée au template ou évaluer une expression
- `{% ... %}` pour afficher du contenu en suivant une logique comme des conditions ou des boucles
- `{# ... #}` pour afficher des commentaires qui ne seront pas rendus dans le HTML

## Afficher des variables

Pour afficher une variable, utilisez la syntaxe `{{ ... }}` :

```
<p> Hello {{ name }}</p>
```

Si votre variable est un objet ou un tableau associatif, vous pouvez utiliser l'opérateur `.` pour accéder au champs d'un objet (publics ou exposé par getter / setter) ou à une valeur d'un tableau associatif :

```
<p> Hello {{ user.name }}</p>
```

Twig vous protège des attaques type faille XSS car il s'occupe d'échapper les caractères HTML.

## Conditions

Pour afficher des conditions, on peut faire un simple if :

```
{% if user.isLoggedIn %}
    Hello {{ user.name }}!
{% else %}
    You are not logged in
{% endif %}
```

## Boucles

On peut afficher le contenu de collections en utilisant des boucles :

```
<ul>
  {% for user in users %}
    <li>{{ user.username|e }}</li>
  {% endfor %}
</ul>
```

## Réutiliser un Template

On peut importer un Template dans un autre pour factoriser du contenu (et éventuellement lui passer des paramètres) :

```
{{ include('block/menu.html.twig', {user: user}) }}
```

## Héritage et Layout

Twig permet à un Template d'hériter d'un autres. Cela est très utile notamment quand on a un squelette de page à répéter sur beaucoup d'autres avec un contenu différent. Le Template parent va donc définir des `block` nommés qui pourront être remplis par les templates enfant.

Par exemple avec le template parents suivant qui correspond à un squelette générique de page HTML :

```
<!DOCTYPE html>
<html>
  <head>
    {% block head %}
      <link rel="stylesheet" href="style.css" />
      <title>{% block title %}{% endblock %} - Website</title>
    {% endblock %}
  </head>
  <body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
      {% block footer %}
        Legal notices : ...
      {% endblock %}
    </div>
  </body>
</html>
```

Un exemple d'enfant :

```
{% extends "base.html" %}

{% block title %}Home{% endblock %}
{% block head %}
    <meta name="description" content="My website ...">
{% endblock %}
{% block content %}
    <h1>Home</h1>
    <p class="important">
        Welcome to my homepage.
    </p>
{% endblock %}
```

Il est également possible pour un block dans un enfant d'appeler le contenu du parent pour l'ajouter à son propre contenu au lieu de le remplacer :

```
{{ parent() }}
```

# Premiers pas avec symfony

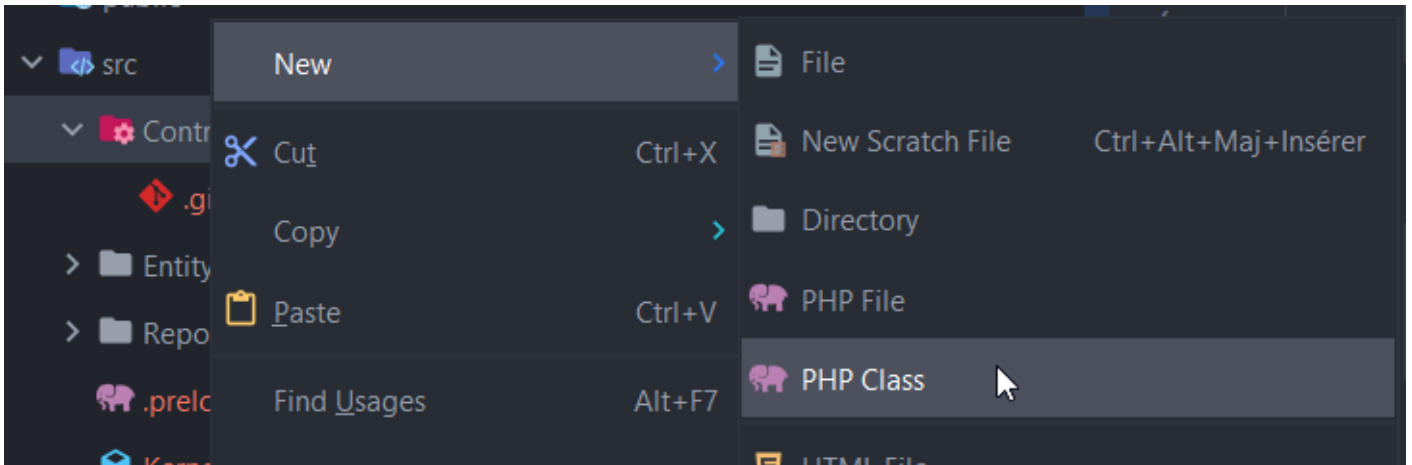
Symfony est un framework MVC et qui utilise le paradigme orienté objet de PHP. Il permet de développer des applications complexes mais maintenables.

## Controllers & Endpoints

Pour répondre à des requêtes HTTP, votre application a besoin de *Controllers* les controllers sont des classes qui contiennent les méthodes endpoints. Une méthode endpoint est une méthode qui va gérer une requête sur une route particulière et avec une méthode HTTP particulière.

## Premier Controller

Pour commencer, créez dans le dossier `src/Controller` de votre projet une nouvelle classe PHP :



Ensuite, pour configurer la route exécutez la commande suivante afin d'installer un package symfony :

```
composer require annotations
```

Vous pouvez ensuite écrire une méthode endpoint dans votre classe :

```
namespace App\Controller;  
  
use Symfony\Component\HttpFoundation\Response;
```

```
use Symfony\Component\Routing\Annotation\Route;

class HelloWorldController
{
    /**
     * @Route("/helloworld")
     */
    public function HelloWorld(){
        return new Response("Hello World from Symfony !");
    }
}
```

L'annotation `@Route` est dans un commentaire mais elle est bien comprise par le framework, elle permet de lier votre méthode endpoint à la route `/helloworld`. Pour essayer tout ça, exécutez :

```
symfony server:start
```

puis rendez vous à : <http://127.0.0.1:8000/helloworld>

## Les Templates

Pour être MVC, il faut dissocier la vue du Controller. Pour cela, nous allons utiliser des Templates afin de décrire les vues.

Dans un premier temps, exécutez la commande suivante pour installer le module de Symfony pour les templates :

```
composer require twig
```

Puis créez dans le dossier `src/templates` un nouveau dossier `hello` et dans ce dossier un fichier `hello.html.twig` avec le contenu suivant :

```
{# templates/lucky/number.html.twig #}
```

Ensuite, changez le code de votre Controller comme ceci, afin de rendre le template :

```
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
```

```
class HelloWorldController extends AbstractController
{
    /**
     * @Route("/helloworld")
     */
    public function HelloWorld(){

        $name = "Shepard";
        return $this->render("hello/hello.html.twig", [
            "name" => $name
        ]);
    }
}
```

Notre Controller étend désormais la classe `AbstractController` afin de pouvoir accéder aux méthodes de rendu de template. On utilise donc la méthode `render` avec en paramètre le nom de notre template ainsi que des données à afficher via un tableau associatif. Enfin, ajouter cette ligne à votre template pour afin d'afficher les données en question :

```
<h1>Hello {{ name }} !</h1>
```

Lancez ensuite votre application et allez sur l'URL de l'endpoint pour constater le rendu de votre template !

## A propos des Routes

### Nommage

Afin de faciliter le debuggage et la génération d'URLs, il est recommandé de nommer ses URLs avec l'attribut `name` :

```
/**
 * @Route("/helloworld", name="hello")
 */
public function HelloWorld(){ ... }
```

### Méthode HTTP

En définissant la Route d'une méthode endpoint, vous pouvez filtrer par méthode HTTP de la façon suivante :

```
/**
 * @Route("/helloworld",name="hello" methods={"GET"})
 */
public function HelloWorld(){ ... }
```

Cela est très utile car cela aide à séparer automatique la logique de présentation et la logique de modification d'un formulaire par exemple.

---

## Paramètres de Route

Il est possible de passer des paramètres dans une route. Attention à ne pas confondre avec les paramètre d'URL qui se trouve après un `?` et sont séparés par des `&`. Un paramètre de route est une partie dynamique de l'URL, que le client utilise pour passer un paramètre au serveur. Souvent, il s'agit du nom ou identifiant de la ressource concernée par la requête. Par exemple dans une application gérant un blog, la route `/posts/6` concernerait le poste de blog avec l'identifiant 6.

Pour déclarer un paramètre de route, procédez ainsi :

```
□ /**
 * @Route("/blog/{id}", name="blog_show", methods = {"GET"})
 */
public function show(int $id) { ... }
```