Université IBN ZOHR

Faculté des Sciences Agadir

Programmation orientée objet avancée

Meryem SOUAIDI
Department of Computer Science
Faculty of Sciences-Agadir

PHP Method chaining

- Chainer des méthodes nous permet d'écrire plusieurs méthodes à la suite les unes des autres, « en chaine ». En pratique, On écrira quelque chose de la forme $objet->methode1()->methode2().
- Cependant, pour pouvoir utiliser le chainage de méthodes, il va falloir que nos méthodes chainées retournent notre objet afin de pouvoir exécuter la méthode suivante. Dans le cas contraire, une erreur sera renvoyée.
- nous allons nous appuyer sur notre classe Fruit créée dans la partie précédente et à laquelle nous allons ajouter deux méthodes.

```php
<?php
class Fruit {
  public $name;
  public $color;
protected  $x = 0;
  public function __construct($name, $color) {
        $this->name = $name;
        $this->color = $color;
  }
public function plusUn(){
        $this->x++;
        echo '$x vaut ' .$this->x. '<br>';
        return $this;
    }
    public function moinsUn(){
        $this->x--;
        echo '$x vaut ' .$this->x. '<br>';
        return $this;
    }
}
$mango = new Fruit("Strawberry", "red");
$mango->plusUn()->plusUn()->plusUn()->moinsUn();
?>
```

```
$x vaut 1
$x vaut 2
$x vaut 3
$x vaut 2
```

Anonymous functions in PHP object

- Les fonctions anonymes, qu'on appelle également des closures, sont des fonctions qui ne possèdent pas de nom.
- Les fonctions anonymes ont été utilisées en tant que fonction de rappel. Une fonction de rappel est une fonction qui va être appelée par une autre fonction.
- Depuis PHP 7, il existe trois grands moyens simples d'appeler une fonction anonyme :
  - En les auto-invoquant de manière similaire au langage JavaScript ;

```php
<?php
    (function(){
     echo 'Fonction anonyme bien exécutée';
     })();
 ?>
```

  - En les utilisant comme valeurs de variables.

```php
<?php
 $txt = function(){
 echo 'Fonction anonyme bien exécutée';
  };
  $squ = function(float $x){
  return 'Le carré de ' .$x. ' est ' .$x**2;
  };
  $txt();
  echo '<br>';
  echo $squ(3);
    ?>
```

Fonction anonyme bien exécutée
Le carré de 3 est 9

  - En les utilisant comme fonctions de rappel ;

```php
<?php
/*L'anonyme  accepte un nombre en argument et retourne son carré*/
$squ = function(float $x){
   return $x**2;    };
    //Définition d'un tableau
 $tb = [1, 2, 3, 4, 5];
//array_map() exécute l'anonyme sur chaque élément du tableau
 $tb_squ = array_map($squ, $tb);
 print_r($tb_squ);
 ?>
```

```
Array ( [0] => 1
        [1] => 4
        [2] => 9
        [3] => 16
        [4] => 25)
```

Anonymous classes in PHP object

- Les classes anonymes, tout comme les fonctions anonymes, sont des classes qui ne possèdent pas de nom.
- On va pouvoir passer des arguments aux classes anonymes via la méthode constructeur et celles-ci vont pouvoir étendre d'autres classes ou encore implémenter des interfaces et utiliser des traits comme le ferait une classe ordinaire.
- Notez qu'on va également pouvoir imbriquer une classe anonyme à l'intérieur d'une autre classe. Toutefois, on n'aura dans ce cas pas accès aux méthodes ou propriétés privées ou protégées de la classe contenante.
- Pour utiliser des méthodes ou propriétés protégées de la classe contenante, la classe anonyme doit étendre celle-ci. Pour utiliser les propriétés privées de la classe contenant dans la classe anonyme, il faudra les passer via le constructeur.
- Dans cet exemple nous avons créer et manipuler une classe anonyme

```php
<?php
  //On crée une classe anonyme qu'on
  *stocke dans une variable (objet)*/
$anonyme = new class{
        public $user_name;
        public const BONJOUR = 'Bonjour ';
        public function setNom($n) {
        $this->user_name = $n;
           }
         public function getNom(){
              return $this->user_name;
         }
   };
$anonyme->setNom('Pierre');
echo $anonyme::BONJOUR;
echo $anonyme->getNom();
echo '<br><br>';
//Affiche les infos de $anonyme
var_dump($anonyme);
?>
```

```
Bonjour Pierre
object(class@anonymous)#1 (1) { ["user_name"]=> string(6) "Pierre" }
```

Anonymous classes in PHP object

- On peut encore assigner une classe anonyme à une variable en passant par une fonction.

```php
<?php
 //On crée une classe anonyme qu'on stocke dans une variable objet
        function anonyme(){
           return new class {
              public $user_name;
              public const BONJOUR = 'Bonjour ';
              public function setNom($n){
                 $this->user_name = $n;
              }
              public function getNom(){
                 return $this->user_name;
              } };
        }
        $anonyme = anonyme();
        $anonyme->setNom('Pierre');
        echo $anonyme::BONJOUR;
        echo $anonyme->getNom();
        echo '<br><br>
        //Affiche les infos de $anonyme
        var_dump($anonyme);
    ?>
```

- utiliser un constructeur pour passer des arguments à une classe anonyme lors de sa création

```php
<?php
 //On crée une fonction qui retourne une classe anonyme
        function anonyme($n){
           return new class($n){
              public $user_name;
              public const BONJOUR = 'Bonjour ';

              public function __construct($n){
                 $this->user_name = $n;
              }
              public function getNom(){
                 return $this->user_name;
              } };
        }
//On stocke le résultat de la fonction dans une variable objet
        $anonyme = anonyme('Pierre');
        echo $anonyme::BONJOUR;
        echo $anonyme->getNom();
        echo '<br><br>';
 //Affiche les infos de $anonyme
        var_dump($anonyme);
 ?>
```

Anonymous classes in PHP object

- Si la classe anonyme est **imbriquée** dans une autre classe, la classe anonyme doit l'**étendre** afin de pouvoir utiliser ses propriétés et méthodes protégées.
- Pour utiliser ses méthodes et propriétés privées, alors il faudra également les passer via le constructeur.
- Dans cet exemple, la classe Externe contient également une méthode qui retourne une classe anonyme. On va vouloir utiliser les propriétés de code>Externe dans la classe anonyme. Pour cela, on passe notre variable protégée dans la définition de la classe et dans le constructeur.

```php
<?php
class Externe{
  private $age = 29;
  protected $nom = 'Pierre';
  public function anonyme(){
    return new class($this->age) extends Externe{

        private $a;
        private $n;

        public function __construct($age){
            $this->a = $age;
        }
        public function getNomAge(){
      return 'Nom : ' .$this->nom. ', âge : ' .$this->a;
        }
      };
    }
}
//Instanciation et appel
$obj = new Externe;
 echo $obj->anonyme()->getNomAge();
?>
```

Nom : Pierre, âge : 29

PHP Autoloading classes

- Nous avons un moyen en PHP de charger (c'est-à-dire d'inclure) automatiquement nos classes d'un seul coup dans un fichier au lieu d'avoir à écrire de longues séries d'inclusion de classes dans nos scripts.
- Pour cela, nous allons pouvoir utiliser la fonction spl_autoload_register(). Cette fonction nous permet d'enregistrer une ou plusieurs fonctions qui vont être mises dans une file d'attente et que le PHP va appeler automatiquement dès qu'on va essayer d'instancier une classe

- Dans cet exemple, on va pouvoir ici soit utiliser une fonction nommée, soit idéalement créer une fonction anonyme :

- Ici, la fonction `spl_autoload_register()` va donc tenter d'inclure les fichiers `Fruit.php` et `Strawberry.php` situés dans un dossier «**test_php**».

```php
<?php
// Strawberry is inherited from Fruit
class Strawberry extends Fruit
{
}
?>
```

```php
<?php
class Fruit {
  public $name;
  function __construct($name)
{
        $this->name = $name;
    }
  function get_name() {
    return $this->name;
    }}
?>
```

```php
<?php
spl_autoload_register(function($classe){
require '../test_php/' .$classe. '.php';
  });

 $apple = new Fruit("Apple");
  $strawberry = new Strawberry("Strawberry");

  echo $apple->get_name();
  echo "<br>";
  echo $strawberry->get_name();
  echo "<br>";
 ?>
```

PHP Passing variables by value or by “reference” (alias)

- On pouvait soit passer une variable par valeur (ce qui est le comportement par défaut), soit par référence en utilisant le symbole & devant le nom de la variable.
- Dans le **passage par valeur**, on travaille avec une « copie » de la variable de départ. Les deux copies sont alors indépendantes et lorsqu’on modifie le contenu de la copie, le contenu de la variable d’origine n’est pas modifié.
- Dans le **passage par référence**, on modifie également la valeur de la variable de base puisque ces deux éléments partagent la même valeur.
- Dans l'exemple ci-dessous on passe l’argument dans notre fonction parValeur() par valeur et on passe l’argument de la fonction parReference() par référence

```php
<?php
$x = 1;
$y = $x;
$z = &$y;
$y = 2;
echo 'Valeur de $x : ' .$x. '<br>
   Valeur de $y : ' .$y. '<br>
   Valeur de $z : ' .$z. '<br>';
   $a = 1;
   $b = 2;
function parValeur($valeur){
$valeur = 5;
 echo 'Valeur dans la fonction : ' .$valeur. '<br>';
}
parValeur($a);
echo 'Valeur de $a : ' .$a. '<br>';

 function parReference(&$reference){
 $reference = 10;
 echo 'Valeur dans la fonction : ' .$reference. '<br>';
       }
 parReference($b);
 echo 'Valeur de $b : ' .$b. '<br>';
 ?>
```

```
Valeur de $x : 1
Valeur de $y : 2
Valeur de $z : 2
Valeur dans la fonction : 5
Valeur de $a : 1
Valeur dans la fonction : 10
Valeur de $b : 10
```

PHP Passing objects: identifiers and references

- Lorsqu'on assigne le résultat de l'instanciation d'une classe dans une variable, vous devez savoir qu'on n'assigne pas véritablement l'objet en soi à notre variable objet mais simplement un identifiant d'objet qu'on appelle également parfois un « pointeur ».
- Lorsqu'on passe une variable objet en argument d'une fonction, ou lorsqu'on demande à une fonction de retourner une variable objet, ou encore lorsqu'on assigne une variable objet à une autre variable objet, ce sont des copies de l'identifiant pointant vers le même objet qui vont être passés par valeur.

- passage par valeur via un identifiant

```php
<?php
class Fruit {
  protected $name;
  function __construct($name) {
        $this->name = $name;
    }
function set_name($n) {
return $this->name = $n;
    }
  function get_name() {
    return $this->name;
    }
  }
$apple = new Fruit("Apple");
$banana=$apple;
$banana->set_name("banana");
echo "<br>";
echo $apple->get_name();
?>
```

banana

- passage par référence via un identifiant

```php
<?php
class Fruit{
public $x = 1;
public function modif(){
      $this->x = 2;  } }
function tesZero($obj){
   $obj = 0;   }
function tesVraimentZero(&$obj){
    $obj = 0;  }
$apple= new Fruit();
$apple->modif();
echo 'Après modif() : ' ;
var_dump($apple);
tesZero($apple);
echo '<br>Après tesZero() : ' ;
var_dump($apple);
tesVraimentZero($apple);
echo '<br>Après tesVraimentZero() : ' ;
var_dump($apple);
      ?>
```

```
Après modif() : object(Fruit)#1 (1) { ["x"]=> int(2) }
Après tesZero() : object(Fruit)#1 (1) { ["x"]=> int(2) }
Après tesVraimentZero() : int(0)
```

PHP Object cloning

- Si on assigne le contenu de notre variable objet dans une nouvelle variable, on ne va créer qu'une copie de l'identifiant qui va continuer à pointer vers le même objet.
- Différemment, le **clonage** d'objet consiste à « copier » un objet afin de manipuler une copie indépendante plutôt que l'objet original.
- Pour cela, on va utiliser le mot clef clone qui va faire appel à la méthode magique __clone() de l'objet si celle-ci a été définie. Notez qu'on ne peut pas directement appeler la méthode __clone().
- Dans cet exemple, Le rôle de la méthode __clone() est ici de modifier la valeur stockée dans $name pour le clone en lui ajoutant « (clone) ».

```
object(Fruit)#1 (1) { ["name":protected]=> string(5) "Apple" }
object(Fruit)#2 (1) { ["name":protected]=> string(13) "Apple (clone)" }
Apple
Apple (clone)
```

```php
<?php
    class Fruit{
    protected $name;

public function __construct($n){
      $this->name = $n;
}

public function __clone(){
      $this->name = $this->name. ' (clone)';
}
 public function getNom(){
        echo $this->name;
        }
    }
    $apple = new Fruit('Apple');
    $banana = clone $apple;

    var_dump($apple);
    echo '<br>';
    var_dump($banana);
    echo '<br>';
    $apple->getNom();
    echo '<br>';
    $banana->getNom();
 ?>
```

PHP object comparison

- En utilisant l'opérateur de comparaison simple ==, les objets vont être considérés comme égaux s'ils possèdent les mêmes attributs et valeurs (valeurs qui vont être comparées à nouveau avec == et si ce sont des instances de la même classe.
- En utilisant l'opérateur d'identité ===, en revanche, les objets ne vont être considérés comme égaux que s'ils font référence à la même instance de la même classe.

```
Egalité simple entre $apple et $apple2 ? bool(true)
Identité entre $apple et $apple2 ? bool(false)

Egalité simple entre $apple et $banana ? bool(true)
Identité entre $apple et $banana ? bool(true)

Egalité simple entre $apple et $kiwi? bool(false)
Identité entre $apple et $kiwi ? bool(false)
```

```php
<?php
class Fruit{
protected $name;
public function __construct($n){
        $this->name = $n;
}
public function __clone(){
        $this->name = $this->name. ' (clone)';     }
public function getNom(){
        echo $this->name;   }
 }
$apple = new Fruit('Apple');
$apple2 = new Fruit('Apple');
$banana = $apple;
$kiwi = clone $apple;

echo 'Egalité simple entre $apple et $apple2 ? ';
var_dump($apple == $apple2);
echo '<br>Identité entre $apple et $apple2  ? ';
 var_dump($apple === $apple2);
echo '<br><br>Egalité simple entre $apple et $banana ? ';
 var_dump($apple == $banana);
echo '<br>Identité entre $apple et $banana ? ';
var_dump($apple === $banana);
echo '<br><br>Egalité simple entre $apple et $kiwi? ';
var_dump($apple == $kiwi);
echo '<br>Identité entre $apple et $kiwi ? ';
  var_dump($apple === $kiwi);
  ?>
```

PHP Design Patterns

- Les **design patterns** sont des solutions typiques à des problèmes communs en développement logiciel : ils ne sont pas une implémentation concrète d'une solution à un problème, mais plutôt une stratégie à appliquer pour le résoudre de façon élégante et maintenable.

- Common patterns in php :

- **Factory** : crée des objets sans avoir à instancier les classes directement.
- **Singleton** : crée un objet sans instanciation et n'autorise pas plus d'une instance de self. Les singletons garantissent qu'il n'y a qu'une seule instance d'un objet à un moment donné.
- **Delegate** : Utilisable si l'objet utilise des fonctionnalités d'autres classes.
- **Decorator** : Les décorateurs ajoutent des fonctionnalités à un objet sans modifier son comportement
- **Strategy** : Le modèle de stratégie définit une famille d'algorithmes, encapsule chacun d'entre eux et les rend interchangeables.
- **Observer** : Les objets (sujets) enregistrent d'autres objets (observateurs) qui réagissent aux changements d'état de leur sujet.
- **Adapter**
- **State**
- **Iterator**
- **Front Controller :** Un objet qui fournit une interface simple pour accéder à un système complexe.
- **MVC :** Un pattern qui gère l'interaction entre une application et un utilisateur (très utilisé pour le web)
- **Active Record**

PHP Design Patterns

- Cas pratique 1 : **Construire un singleton**

1. Créez un fichier test_singleton.php
   - Dans ce fichier, créer une classe 'User_Singleton' contenant une propriété privée 'name' et ses deux accesseurs
2. Transformation en singleton
   - En utilisant __construct, faites en sorte que l'on ne puisse pas instancier la classe
   - En utilisant __clone, faites en sorte qu'on ne puisse pas cloner l'objet
   - Déclarez une méthode publique 'getInstance()' qui renvoie l'objet unique de la classe 'User_Singleton'

- Cas pratique 2 : **Construire un factory**

1. Créez un fichier test_factory.php
   - Dans ce fichier, créer une classe 'SMSFactory' contenant une fonction create_messenger() et une classe 'SMSMessage' contenant un constructeur et une fonction send_message qui recevera le message en argument et le renvoie via SMS.
2. Transformation en factory
   - La méthode publique 'create_messenger()' renvoie un objet de la classe 'SMSFactory'

PHP Design Patterns

- Cas pratique1: Solution

```php
<?php
class User_Singleton{
  private $name;
  public static $instance;
  public function setNom($n){
    $this->name=$n;
  }
  public function getNom(){
    return $this->name;
  }
  private function __construct(){}
  private function __clone(){}
  public static function getInstance(){
    if(!(self::$instance instanceof self)){
      self::$instance= new self();
    }
    return self::$instance;
    }
  }
  ?>
```

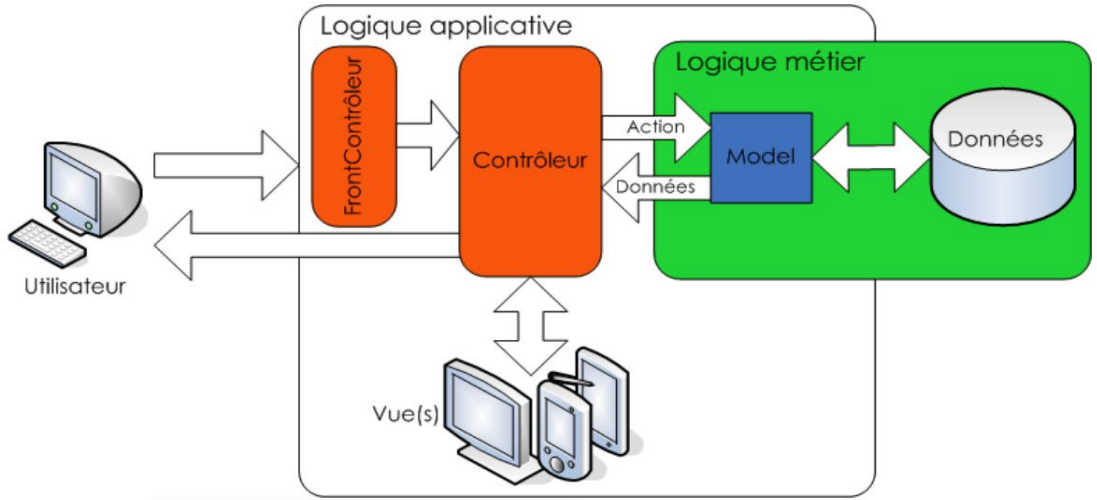
```php
//Returns instance of SomeSingleton
  $var =User_Singleton::getInstance();
  $var->setNom("Ahmed");
  echo $var->getNom();
  $var =new User_Singleton(); //Fatal
*error call to private */
User_Singleton::__construct()
```

- Cas pratique2: Solution

```php
<?php
class SMSFactory{
  public function create_messenger(){
    return new SMSMessage();
  }
}
class SMSMessage{
  public function __construct(){}
  public function send_message($Mess){
    return $Mess;
  }
}
$var=new SMSFactory();
$message =$var->create_messenger();
echo $message->send_message('Hello
world!');
  ?>
```

PHP Design Patterns : MVC

- Le MVC est très utilisé dans les applications web. Il est décomposé en 3 parties:
  - **Le contrôleur** : qui gère les échanges avec l'utilisateur. Le contrôleur reçoit des requêtes de l'utilisateur et demande au modèle d'effectuer certaines actions et de lui renvoyer les résultats. Enfin, il va renvoyer la nouvelle page HTML, générée par la vue, à l'utilisateur.
  - **Le modèle** : qui gère la structuration et l'accès aux données. Mais aussi tout le code qui prend des décisions autour de ces données
  - **La vue** : qui gère la partie 'présentation' (visible). Elle ne fait presque aucun calcul et se contente de récupérer des variables pour savoir ce qu'elle doit afficher.



MVC : model and view separation

- Le code de cette page tient dans un fichier **index.php**

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Les visiteurs</title>
    <!--  <link href="style.css" rel="stylesheet" />-->
</head>
<body>
    <h1>Les informations des visiteurs</h1>
    <p>Date de dernier visite :</p>
    <?php
    // Connexion à la base de données
    try
    {
 $bdd = new PDO('mysql:host=localhost;dbname=mydb;charset=utf8',
'root', ''); }
    catch(Exception $e){
         die( 'Erreur : '.$e->getMessage()   );
        }
// On récupère les 5 derniers visite
    $req = $bdd->query('SELECT id, firstname, lastname,
DATE_FORMAT(reg_date, \'%d/%m/%Y à %Hh%imin%ss\') AS
date_visite_fr FROM myguests ORDER BY reg_date DESC LIMIT 0, 5');
    while ($donnees = $req->fetch())
    {
    ?>
    <div class="news">
        <h3>
            <?php echo
htmlspecialchars($donnees['firstname']); ?>
            <?php
      // On affiche le prenom
              echo  nl2br ( htmlspecialchars(
$donnees['lastname']));
      ?>
            <em>le <?php echo $donnees['date_visite_fr'];
?></em>
        </h3>
        <p>

            <br />
            <em><a href="#">Commentaires</a></em>
        </p>
    </div>
    <?php
    } // Fin de la boucle des visites
    $req->closeCursor();
    ?>
</body>
</html>
```

MVC : model and view separation

- Affichage :

Les informations des visiteurs

Date de dernier visite :

John Doe ***le 17/11/2023 à 23h19min23s***
Commentaires

Mary Moe ***le 17/11/2023 à 23h19min23s***
Commentaires

Julie Dooley ***le 17/11/2023 à 23h19min23s***
Commentaires

John Doe ***le 17/11/2023 à 23h17min27s***
Commentaires

John Doe ***le 17/11/2023 à 23h14min59s***
Commentaires

- Il y a des **défauts** qui devraient vous sauter aux yeux. 👀
  - Il est beaucoup plus difficile de lire – et de maintenir – du code qui ne respecte pas une convention de codage.
  - Coder toute son application dans un seul fichier pose de gros problèmes de travail en équipe.
  - Plus le projet est grand, plus les problèmes de code sont impactants.
  - Même si il fonctionne, si vous allez avoir besoin d'ajouter de nouvelles requêtes SQL au milieu. Très vite, le fichier va grossir et faire plusieurs centaines de lignes.

MVC : model and view separation

- Solution :

- On sépare le code en trois fichiers :
  1. **`src/model.php (Le modèle)`** : se connecte à la base de données et récupère les visites.
  2. **`templates/homepage.php (La vue)`** : affiche les informations de la page en les formatant. Étant donné qu'on dispose de variables de types simples, il est assez facile de les formater comme on le souhaite (avec les fonctions **`nl2br()`** et **`htmlspecialchars()`**, par exemple).
  3. **`index.php (Le contrôleur)`**: fait le lien entre le modèle et l'affichage.

- <u>3 étapes toutes simples</u>

1. Notre nouveau fichier **`src/model.php`** contient une seule fonction **`getPosts()`** qui renvoie la liste des visites. On charge le fichier du modèle. Il ne se passe rien pour l'instant.
2. On appelle la fonction, ce qui exécute le code à l'intérieur de **`src/model.php`**. On y récupère la liste des visites dans la variable **`$posts`**.
3. On charge le fichier de la vue **`templates/homepage.php via index.php`** (l'affichage), qui va présenter les informations dans une page HTML

contrôleur

vue

modèle

index.php

```php
<?php
require('src/model.php');
$posts = getPosts();
require('templates/homepage.php');
?>
```

templates/homepage.php

```php
<!DOCTYPE html>
<html> <head>
   <meta charset="utf-8" />
   <title>Les visiteurs</title>
   <!--  <link href="style.css" rel="stylesheet" />-->
</head><body>
   <h1>Les informations des visiteurs</h1>
   <p>Date de dernier visite :</p>
   <?php
      foreach ($posts as $post) {
      ?>
   <div class="news">
      <h3>
         <?php echo htmlspecialchars($post['firstname']);
         // We display the post content.
         echo nl2br(htmlspecialchars($post['lastname'])); ?>
         <em>le <?php echo $post['french_visite_date']; ?></em>
      </h3>
      <p> <br />  <em><a href="#">Commentaires</a></em> </p>
   </div>
   <?php  } // The end of the posts loop.
      ?>
</body></html>
```

src/model.php

```php
<?php
function getPosts() {
   // We connect to the database.
   try
   {
 $bdd = new PDO('mysql:host=localhost;dbname=mydb;charset=utf8', 'root',
'');  }
   catch(Exception $e){
        die( 'Erreur : '.$e->getMessage()   ); }
    // On récupère les 5 derniers visite
    $statement = $bdd->query('SELECT id, firstname, lastname,
DATE_FORMAT(reg_date, \'%d/%m/%Y à %Hh%imin%ss\') AS
date_visite_fr FROM myguests ORDER BY reg_date DESC LIMIT 0, 5');
   $posts = [];
   while (($row = $statement->fetch())) {
      $post = [
         'firstname' => $row['firstname'],
         'lastname' => $row['lastname'],
         'french_visite_date' => $row['date_visite_fr'], ];
      $posts[] = $post; }
   return $posts;
}
?>
```

MVC : Exercice Pratique

1. On suppose on a deja créer en utilisant **PDO:**
   * une Base de données nommée ”MyDataBase”.
   * Créer une table SQL ”**MyGuests**” qui contiendra les champs suivants:
   – Id INT(6) clé primaire et auto increment
   – firstname VARCHAR(30) not null
   – lastname VARCHAR(30) not null
   – email VARCHAR(50)
   – reg_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
   * 3 fichiers index.php, src/model.php et templates/homepage.php (code source Slide 19)

- On aimerait pouvoir afficher une page avec les commentaires de chaque visite

2. Créer la vue **templates/post.php**
   Dans cette vue, on affiche :

   - les infos des visiteurs : on utilise une variable tableau $post, avec les firstname,lastname et french_visite_date;
   - On a besoin d'une nouvelle propriété **identifier** au niveau de chaque $post de notre page d'accueil. Et on va directement demander à notre modèle de nous la donner.

MVC : Exercice Pratique

- les commentaires : un tableau de tableaux, avec les firstname, french_visite_date et comment.

3. Nous avions déjà écrit un contrôleur **index.php** pour gérer la liste des premiers visites. Je vous propose d'écrire un autre contrôleur **post.php** qui affiche un post et ses commentaires.

- Quand vous cliquez sur les liens "Commentaires" de la page d'accueil, vous accédez dorénavant à votre nouvelle page. Modifions maintenant le contrôleur **post.php** pour prendre en compte ce paramètre GET :

- Il fait un test, un contrôle : il vérifie qu'on a bien reçu en paramètre un id dans l'URL (**$_GET['id']**).
- Ensuite, il appelle les 2 fonctions du modèle dont on va avoir besoin :**getPost** et **getComment**. On récupère ça dans nos deux variables. Bien sûr, on attend du modèle qu'il nous renvoie les mêmes propriétés que celles qu'on avait définies :
  - **firstname ,lastname et french_visite_date** pour les visites ;
  - **firstname ,french_visite_date** et **comment** pour les commentaires.

MVC : Exercice Pratique

4. Mettez à jour le modèle (**PDO**) en ajoutons une **table comments** pour gérer les commentaires dans la base. Elle aura cette structure

comments	
id	integer
post_id	integer
firstname	varchar
comment	text
comment_date	date/time

notre fichier **src/model.php** ne contient qu'une seule fonction **getPosts** qui récupère tous les derniers posts de blog.

- On va écrire 2 nouvelles fonctions :
- **getPost**(au singulier !), qui récupère un post précis en fonction de son ID ;
- **getComments**, qui récupère les commentaires associés à un ID de post.

Ces deux nouvelles fonctions prennent un paramètre : l'identifiant du visite qu'on recherche. Cela nous permet notamment de ne sélectionner que les commentaires liés au post concerné.

- créer une fonction **dbConnect()** qui va renvoyer la connexion à la base de données (sert à éviter la répétition de connexion à la base de données avec les blocs try/catch)