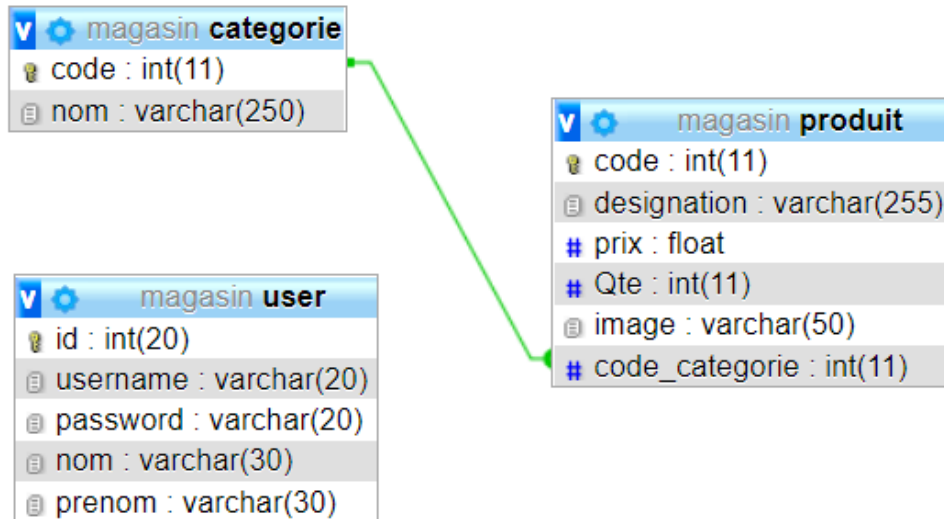


– Architecture MVC en PHP–

Pré-requis

Nous allons partir de la base de données magasin



Et le dernier TP

TP3

- inc
 - css
 - images
 - js
- footer.php
- header.php
- navigation.php
- add.php
- crudProduits.php
- login.php
- logout.php
- register.php
- update.php

My Flowers Shop

Se déconnecter

nouha

Accueil Produits Catégories Utilisateurs

Gestion produits

Search...

Ajouter Produit

| # | Designation | Catégorie | Quantité de stock | Prix unitaire | Image | Actions |
|----|--------------------------|---------------------|-------------------|---------------|-------|---------|
| 27 | Amaryllis | Plantes d'intérieur | 4 | 25 | | |
| 28 | Trio de piments | Plantes d'intérieur | 2 | 15 | | |
| 29 | Jardinière de campanules | Plantes d'intérieur | 15 | 40 | | |

Activer W
Accédez aux p

NB. Il est recommandé d'ajouter un menu de navigation

Code du fichier « navigation.php »

```
<nav class="navbar navbar-expand-sm bg-light navbar-light">
  <ul class="navbar-nav">
    <li class="nav-item">
      <a class="nav-link " href="">Accueil</a>
    </li>
    <li class="nav-item">
      <a class="nav-link active" href="crudProduit.php">Produits</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="">Catégories</a>
    </li>
    <li class="nav-item">
      <a class="nav-link disabled" href="">Utilisateurs</a>
    </li>
  </ul>
</nav>
```













Dans la page crudProduit.php, pour ajouter ce menu, il suffit d'inclure le fichier navigation.php

```
<?php require "inc/header.php";?>
<?php require "inc/navigation.php";?>

.....

<?php require "inc/footer.php"; ?>
```

Nous allons créer l'arborescence suivante :

- ▼ **VERSION_1**
 - ▼ app
 - ▼ controllers
 -  Controller.php
 -  ControllerHome.php
 -  ControllerProduit.php
 - ▼ models
 -  Database.php
 -  Model.php
 -  ModelProduit.php
 - ▼ views
 - > Home
 - ▼ Produit
 -  crud.php
 -  footer.php
 -  header.php
 -  navigation.php
 - ▼ assests
 - ▼ css
 - # bootstrap.min.css
 - # starter-template.css
 - > images
 - ▼ js
 - JS bootstrap.bundle.min.js
 - ▼ config
 -  App.php
 -  index.php

Architecture MVC

Au fur et à mesure que votre site Web grandit, vous allez rencontrer des difficultés à organiser votre code. Cette section vise à vous montrer une bonne façon de concevoir votre site web. On appelle design pattern (patron de conception) une série de bonnes pratiques pour l'organisation de votre site. Un des plus célèbres design patterns s'appelle MVC (Modèle - Vue - Contrôleur).

Le pattern **MVC** permet de bien organiser son code source.

Au début du cours, nous avons programmé de manière monolithique : nos pages Web mélangent **traitement** (PHP), **accès aux données** (SQL) et **présentation** (balises HTML). Nous allons maintenant séparer toutes ces parties pour plus de clarté.

L'objectif est de réorganiser le code pour finalement y rajouter plus facilement de nouvelles fonctionnalités. Nous allons mieux comprendre le fonctionnement sur l'exemple de la gestion des produits.

L'architecture **MVC** est une manière de découper le code en trois bouts M, V et C ayant des fonctions bien précises.

Dans notre exemple, l'ancien fichier `crudProduit.php` qui permet d'afficher la liste des produits dans un tableau HTML va être réparti entre le contrôleur `controller/ControllerProduit.php`, le modèle `model/ModelProduit.php` et la vue `views/produit/crud.php`.

1- M: Le modèle

Le modèle est chargé de la gestion des données, notamment des interactions avec la base de données(persistance). C'est, par exemple, la classe Model vue en cours.

Dans le répertoire model placer les classes vues en cours (exercice refactoring) : Model.php, Database.php et App.php. Attention aux inclusions !

1.1- La classe App (Rappel)

```
<?php
class App{
    const DB_NAME = 'magasin';
    const DB_USER = 'root';
    const DB_PASS = '';
    const DB_HOST = 'localhost';
}
?>
```

Cette classe stocke les paramètres de connexion à la BD dans des constantes de classe.

1.12- La classe Database (Rappel)

```
<?php
include "config\App.php";
class Database {
    private static $instance = null;
    private $pdo;

    private function __construct() {
        // Private constructor to prevent direct instantiation
        try {
            $this->pdo = new PDO("mysql:host=".App::DB_HOST.";dbname=".App::DB_NAME."",
App::DB_USER, App::DB_PASS);
            $this->pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        } catch (PDOException $e) {
            die("Database connection failed: " . $e->getMessage());
        }
    }
    public static function getInstance() {
        if (self::$instance === null) {
            self::$instance = new self();//new Database();
        }
        return self::$instance;
    }
    public function getConnection() {
        return $this->pdo;
    }
}
?>
```

Cette classe permet :

1. La connexion à une BD MySQL en utilisant l'API PDO. Cette connexion se fait une seule fois grâce au pattern singleton. C'est la méthode `getInstance` qui établit la connexion à la BD une seule fois en retournant une instance unique de la classe Database.

Tester les deux classes Database et App. Par exemple ajouter le fichier `test.php` (à la racine) qui contient le code suivant :

```
<?php
require_once "app\models\Database.php";
// Get a reference to the database connection
$db = Database::getInstance()->getConnection();
var_dump($db );
?>
```

1.2- La classe Model (Rappel)

```
<?php
class Model {
    protected $table;
    protected $db;
    public function __construct($db, $table) {
        $this->db = $db;
        $this->table = $table;
    }
    public function save($data) {
        try {
            if (isset($data['code'])) {
                // Update an existing record
                $columns = array_keys($data);
                $sets = [];
                $i=0;
                foreach ($columns as $column) {
                    $sets[$i] = "$column = :$column";
                    $i++;
                }
                $setString = implode(', ', $sets);

                $sql = "UPDATE $this->table SET $setString WHERE code = :code";
            } else {
                // Insert a new record
                $columns = implode(', ', array_keys($data));
                $placeholders = ':' . implode(', ', array_keys($data));
                $sql = "INSERT INTO $this->table ($columns) VALUES ($placeholders)";
            }
            $stmt = $this->db->prepare($sql);
            $stmt->execute($data);
        } catch (PDOException $e) {
            die("Error saving data: " . $e->getMessage());
        }
    }
    public function find($code) {
        try {
            $sql = "SELECT * FROM $this->table WHERE code = :code";
            $stmt = $this->db->prepare($sql);
            $stmt->execute(['code' => $code]);
            return $stmt->fetch(PDO::FETCH_ASSOC);
        } catch (PDOException $e) {
            die("Error finding data: " . $e->getMessage());
        }
    }
}
```

```

public function findAll() {
    try {
        $sql = "SELECT * FROM $this->table";
        $stmt = $this->db->query($sql);
        return $stmt->fetchAll(PDO::FETCH_ASSOC);
    } catch (PDOException $e) {
        die("Error finding data: " . $e->getMessage());
    }
}

public function create($data) {
    try {
        $columns = implode(', ', array_keys($data));
        $placeholders = ':' . implode(', ', array_keys($data));
        $sql = "INSERT INTO $this->table ($columns) VALUES ($placeholders)";

        $stmt = $this->db->prepare($sql);
        $stmt->execute($data);
        return $this->db->lastInsertId();
    } catch (PDOException $e) {
        die("Error creating data: " . $e->getMessage());
    }
}

public function update($code, $data) {
    try {
        $data['code'] = $code;
        $columns = array_keys($data);
        $sets = [];
        $i = 0;
        foreach ($columns as $column) {
            $sets[$i] = "$column = :$column";
            $i++;
        }
        $setString = implode(', ', $sets);
        $sql = "UPDATE $this->table SET $setString WHERE code = :code";
        $stmt = $this->db->prepare($sql);
        $stmt->execute($data);
    } catch (PDOException $e) {
        die("Error updating data: " . $e->getMessage());
    }
}

public function delete($code) {
    try {
        $sql = "DELETE FROM $this->table WHERE code = :code";
        $stmt = $this->db->prepare($sql);
        $stmt->execute(['code' => $code]);
    } catch (PDOException $e) {
        die("Error deleting data: " . $e->getMessage());
    }
}

```

Tester la classe Model. Par exemple, dans la racine, ajouter le fichier test.php suivant :

```
<?php
require_once "app\models\Database.php";
require_once "app\models\Model.php";

// Get a reference to the database connection
$db = Database::getInstance()->getConnection();

// Create an instance of the Generic Model class for the 'Produit' table
$produit = new Model($db, "Produit");
// Call the findAll method to get all the products
$produit_records = $produit->findAll();

var_dump($produit_records );
?>
```

La classe générique Model :

Avantages :

- Réutilisabilité : Une classe générique Model peut être réutilisée pour différentes entités sans avoir besoin de créer des sous-classes pour chaque entité (exp. Produit, user, etc).
- Consistance : Elle impose une structure et un comportement cohérents entre différentes entités.

Inconvénients :

- Moins de spécialisation : Elle peut être moins spécialisée pour des entités spécifiques, et on peut avoir besoin d'ajouter une logique conditionnelle pour gérer les exigences spécifiques à chaque entité au sein d'une seule classe.
- Complexité potentielle : À mesure que l'application se développe, une classe générique Model unique pourrait devenir complexe à gérer.

Compte tenue de ces inconvénients, il serait judicieux d'ajouter quand nécessaire une sous-classe par entité, qui hérite de la classe Model exemple. C'est le cas de la classe ModelProduit:

```
class ModelProduit extends Model {
```

Les sous-classes peuvent avoir des **méthodes** ou des **propriétés spécifiques** aux besoins de chaque entité, offrant une structure plus spécialisée et claire.

1.3- La classe ModelProduit

```
<?php
require_once 'app\models\Model.php';
class ModelProduit extends Model{
    private $data=array();
    // La syntaxe ... = NULL signifie que l'argument est optionnel
    // Si un argument optionnel n'est pas fourni,
    // alors il prend la valeur par défaut, NULL dans notre cas
    public function __construct($db,$code=null,$designation=null, $prix=null,
    $qte=null, $categorie=null){
        parent::__construct($db, 'Produit');
        if (!is_null($designation) && !is_null($prix) && !is_null($qte) &&
        !is_null($categorie)) {
            // Si aucun des paramètre n'est nul,
            // c'est forcément qu'on les a fournis
            // donc on retombe sur le constructeur à 1 argument
            $this->data['code'] = $code;
            $this->data['designation'] = $designation;
            $this->data['prix'] = $prix;
            $this->data['qte'] = $qte;
            $this->data['categorie'] = $categorie;
        }
    }
    public function __get($attr){
        if (!isset($this->data[$attr]))
            return "erreur";
        else return ($this->data[$attr]);
    }
    public function __set($attr,$value) {
        $this->data[$attr] = $value;
    }
    public function findAll() {
        try {
            $sql = "SELECT produit.code, produit.designation, produit.prix,
            produit.Qte, produit.image, categorie.nom as categorie
            FROM produit
            LEFT JOIN categorie
            ON code_categorie = categorie.code";
            $stmt = $this->db->query($sql);
            return $stmt->fetchAll(PDO::FETCH_ASSOC);
        } catch (PDOException $e) {
            die("Error finding data: " . $e->getMessage());
        }
    }
}
```

Tester à chaque étape...

2. V: La vue

Dans la vue sont regroupés toutes les lignes de code qui génèrent la page HTML que l'on va envoyer à l'utilisateur. Les vues sont des fichiers qui ne contiennent quasiment exclusivement que du code HTML, à l'exception de quelques echo permettant d'afficher les variables préremplies par le contrôleur. Une boucle for est toutefois autorisée pour les vues qui affichent une liste d'éléments. **La vue n'effectue pas de traitement/de calculs.**

Dans notre exemple, la première vue serait le fichier `views/Produit/crud.php`.

Le code de ce fichier permet d'afficher une page Web contenant tous les produits contenus dans la variable `$produit`.

Cette page correspond au fichier `crudProduit` ancien mais sans la partie PHP.

N.B les liens `<a href ...>` seront traités ultérieurement.

```
<?php require "app/views/header.php"; ?>
<?php require "app/views/navigation.php"; ?>

<div id="outer" class="container d-flex align-items-center justify-content-center">
<main>
    <div class="d-flex justify-content-between flex-wrap flex-md-nowrap align-items-center pt-3 pb-2 mb-3 border-bottom">
        <div class="table-responsive">
            <div class="table-wrapper">
                <div class="table-title">
                    <div class="row">

                        <div class="col-sm-8"><h2>Gestion <b>produits</b></h2></div>
                        <div class="col-sm-4">
                            <div class="search-box mb-4" >
                                <i class="material-icons">&#xE8B6;</i>
                                <input type="text" class="form-control"
                                                                    placeholder="Search&hellip;" >
                            </div>
                        </div>
                    </div>
                </div>
            </div>
            <div class="row">
                <div class="col-sm-8"></div>
                <div class="col-sm-4 ">
                    <a href="#" class="btn btn-success float-right" data-toggle="modal">
                        <i class="material-icons">&#xE147;</i>
                        <span>Ajouter Produit</span></a>
                </div>
            </div>
        </div>
    </div>
</div>
```

```

    </div>
    <table class="table table-striped table-hover table-bordered">
        <thead>
            <tr>
                <th>#</th>
                <th>Designation </i></th>
                <th>Catégorie</th>
                <th>Quantité de stock</th>
                <th>Prix unitaire</th>
                <th>Image </th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody id="table">
            <?php
            foreach($produits as $produit){
                echo '<tr>';
                echo '<td>'. $produit['code']. '</td>';
                echo '<td>'. $produit['designation'] . '</td>';
                echo '<td>'. $produit['categorie']. '</td>';
                echo '<td>'. $produit['Qte']. '</td>';
                echo '<td>'. $produit['prix']. '</td>';
                echo '<td>  </td>';
            ?>

            <td>
                <a href="" class="edit" title="Edit" data-toggle="tooltip"><i
class="material-icons">&#xE254;</i></a>
                <a href="" class="delete" title="Delete" data-toggle="tooltip"><i
class="material-icons">&#xE872;</i></a>
            </td>
        </tr>
        <?php } ?>

    </tbody>
</table>
</div>
</div>
</div>

<?require "app/views/footer.php";?>

```

3. C: le contrôleur

Le contrôleur gère la logique du code qui prend des décisions. C'est en quelque sorte l'intermédiaire entre le modèle et la vue : le contrôleur va demander au modèle les données, les analyser, prendre des décisions et appelle la vue adéquate en lui donnant le texte à afficher à la vue. **Le contrôleur contient exclusivement du PHP.**

Il existe une multitude d'implémentations du MVC:

- un gros contrôleur unique
- un contrôleur par modèle
- un contrôleur pour chaque action de chaque modèle

Nous choisissons ici la version intermédiaire (un contrôleur par modèle) et commençons à créer un contrôleur Abstrait Controller puis le contrôleur ControllerProduit pour ModelProduit.

```
<?php
abstract class Controller {

    // Abstract methods to be implemented by concrete subclasses
    abstract public function index();

    //abstract public function show($id);

    //abstract public function create();

    //abstract public function edit($id);

    //abstract public function delete($id);
}
```

On appelle **action** une fonction du contrôleur ; **une action correspond généralement à une page Web.**

Dans le contrôleur Controller, nous avons les actions communs à tous les contrôleurs.

Dans le contrôleur ControllerProduit, nous implémentons les actions en fonction de l'entité Produit.

```

<?php
require_once 'app\models\ModelProduit.php';
require_once 'app\models\Model.php';
require_once 'app\models\Database.php';
require_once 'app\controllers\Controller.php';
require_once 'app/views/View.php';

class ControllerProduit extends Controller{
    private $model;

    public function __construct() {
        $db = Database::getInstance()->getConnection();
        $this->model = new ModelProduit($db);
    }

    public function index() {
        $produits = $this->model->findAll();
        // $controller = "Produit";
        include('app/views/Produit/crud.php');
    }
    // Other actions...
}

```

1. afficher la liste des produits : action index()
2. afficher un produit défini par un code particulier: action show()
3. afficher le formulaire d'ajout d'un produit : action create()
4. afficher le formulaire de mise à jour d'un produit :action edit()
5. supprimer un produit et afficher un message de confirmation : action delete()
6. traitement du formulaire: action createProcess ()

Pour le moment, nous avons besoin de l'action show uniquement. Cette action effectue les étapes suivantes :

1. on se sert du modèle pour récupérer le tableau de tous les produits avec
`$produits = $this->model->findAll();`
2. on appelle ensuite la vue qui va nous générer la page Web avec
`include('app/views/Produit/list.php');`

Dans la racine, créer le fichier index.php pour tester

```

<?php
require_once ("app/controllers/ControllerProduit.php");
(new ControllerProduit()->index());

```

4. Le routeur : un autre composant du contrôleur

Le routeur est généralement un composant distinct chargé d'interpréter les demandes entrantes et de les diriger vers le **contrôleur** et l'**action** appropriés. La logique de routage est souvent centralisée et peut résider dans un fichier séparé ou une classe dédiée.

Ajouter le répertoire routing à la racine et les deux fichiers Router.php et routes.php

```

  app
  > controllers
  > models
  > views
  > assests
  > config
  routing
  Router.php
  routes.php
```

Créons une classe Router simple qui gère le routage dans une configuration MVC de base. Cette classe analysera l'URL entrante, déterminera le **contrôleur** et l'**action**, puis enverra le résultat.

```
<?php
class Router {
    protected $routes = [];
    public function addRoute($uri, $controllerAction) {
        $this->routes[$uri] = $controllerAction;
    }

    public function dispatch($uri) {
        $controllerAction = $this->getControllerAction($uri);

        if ($controllerAction === false) {
            // Handle 404: route not found
            echo '404 Not Found';
            return;
        }

        list($controllerName, $actionName) = explode('@', $controllerAction);

        // Include the appropriate controller file
        require_once __DIR__ . '/../app/controllers/' . $controllerName . '.php';

        // Create an instance of the controller
```

```

        $controller = new $controllerName();

        // Call the action method
        $controller->$actionName();
    }

    protected function getControllerAction($uri) {

        return isset($this->routes[$uri]) ? $this->routes[$uri] : false;
    }
}

```

Le fichier `routes.php` définit les routes en utilisant la méthode `addRoute` du routeur.

```

<?php
require "Router.php";
$router = new Router();

// Define routes
$router->addRoute('/', 'ControllerHome@index');
$router->addRoute('/products', 'ControllerProduit@index');
//$router->addRoute('/products/show', 'ControllerProduit@show');
//$router->addRoute('/products/create', 'ControllerProduit@create');
//$router->addRoute('/products/createProcess', 'ControllerProduit@createProcess');
//$router->addRoute('/products/edit', 'ControllerProduit@edit');
//$router->addRoute('/products/delete', 'ControllerProduit@delete');

```

Voilà le nouveau code du fichier `index.php`, qui contient l'appel du routeur. Tester (ça doit afficher la liste des produits)

```

<?php
require 'routing/routes.php'; // ceci implique l'instanciation de Router()

$uri = isset($_GET['url']) ? "/" . $_GET['url'] : '/'; //récupération de l'URI

$router->dispatch($uri);

```

Explication

En fait, le routeur fait partie du contrôleur et s'occupe d'appeler l'action du contrôleur.

Voici le déroulé de l'exécution pour l'URL suivante tapée à la barre de navigation:

`http://localhost/ MVC/Version_2/index.php?url=products`

1. Le client demande l'URL à partir du navigateur web:

`http://localhost/MVC/Version_2/index.php?url=products`

2. Exécution de la page `index.php`.
3. La page `index.php` :
 - a. Récupère l'URI à partir de l'URL. La variable `$uri` contient `'/products'`
 - b. Demande au routeur de dispatcher cette URI :

```
<?php
require 'routing/routes.php'; // ceci implique l'instanciation de Router()

$uri = isset($_GET['url']) ? "/" . $_GET['url'] : '/'; //récupération de l'URI

$router->dispatch($uri);
```

4. Le routeur (à travers la méthode `dispatch`) va donc effectuer les opérations suivantes :
 - a. Identifier la correspondance `$controllerAction` à partir de la variable `$uri`, en utilisant le tableau `$routes`.

```
$controllerAction = $this->getControllerAction($uri);
```

Voici un exemple du tableau `$routes` :

| <code>\$uri</code> | <code>\$controllerAction</code> |
|---------------------------------|---|
| <code>'/'</code> | <code>'ControllerHome@index'</code> |
| <code>'/products'</code> | <code>'ControllerProduit@index'</code> |
| <code>'/products/create'</code> | <code>'ControllerProduit@create'</code> |

Donc `controllerAction` contient `'ControllerProduit@index'`

- b. Identifie le `$ControllerName` et le `$ControllerAction` à partir de `$controllerAction`.

```
list($controllerName, $actionName) = explode('@', $controllerAction);
```

ControllerName contient ControllerProduit et
ControllerAction contient index

c. Il appelle donc l'action index du ControllerProduit

```
// Include the appropriate controller file
require_once __DIR__ . '/../app/controllers/' . $controllerName . '.php';

// Create an instance of the controller
$controller = new $controllerName();

// Call the action method
$controller->$actionName();
```

5. Dans l'action index de ControllerProduit

- a. Appel du modèle pour récupérer le tableau de tous les produits (\$produits)
- b. Appel de la vue qui va nous générer la page Web (crud.php).

```
public function index() {
    $produits = $this->model->findAll();
    $controller = "Produit";
    include('app/views/Produit/crud.php');
}
```

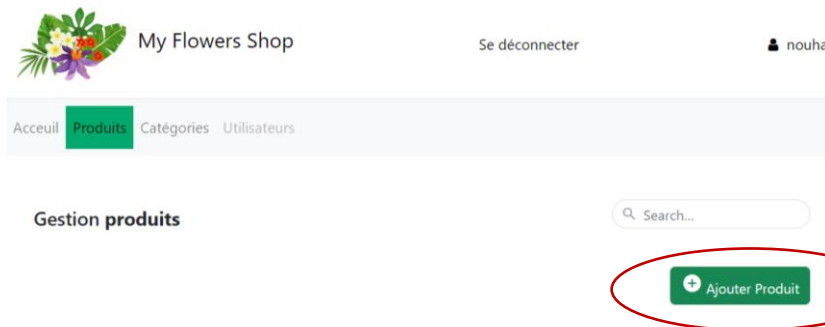
5. Vue "ajout" d'un produit

Nous allons créer deux actions dans le contrôleur `ControllerProduit`: `create` et `createProcess` qui doivent respectivement afficher un formulaire d'ajout d'un produit et effectuer l'enregistrement dans la BD.

1. Commençons par l'action `create` qui affichera le formulaire :
 - i. Créez la vue `/view/Produit/add.php` qui contient le formulaire.
 - ii. Rajoutez une action `create` à `ControllerProduit.php` qui affiche cette vue.
2. Testez votre page en appelant l'action `create` passée au routeur :

`__/_/index.php?url=products/create`

Tester aussi en programmant le bouton « Ajouter Produit »



```
<a href="/index.php?url=products/create" class="btn btn-success float-right" data-toggle="modal"><i class="material-icons">&#xE147;</i><span>Ajouter Produit</span></a>
```

3. Créez l'action `createProcess` dans le contrôleur qui devra principalement
 - i. récupérer les données du produit à partir du formulaire
 - ii. appeler la méthode `save` du modèle,
 - iii. appeler la fonction `index()` du contrôleur (si pas d'erreurs) pour afficher le tableau de tous les produits.
4. Testez le tout, c-à-d. que la création d'un produit depuis le formulaire (action `create`) appelle bien l'action `createProcess` et que le produit est bien ajouté dans la BD.

Vous souhaitez envoyer l'information `"url=products/createProcess"` en plus des informations saisies lors de l'envoi du formulaire => 2 solutions:

- i. rajoutez un champ caché à votre formulaire :

```
<input type='hidden' name='url' value=' products/createProcess'>
```

- ii. ou passez dans l'action du formulaire

```
<form action="index.php?url= products/createProcess" method=POST
>
```

La méthode du formulaire est au choix (Get ou Post).

6. Vue "mise à jour" et suppression d'un produit

Dans la page crud.php, mettre à jour les liens href pour la mise à jour et la suppression en passant dans l'URL, le controller et l'action.

```
<td>
<a href="index.php?url=products/edit&code=<?=$produit['code']?>" class="edit" title="Edit" data-
toggle="tooltip"><i class="material-icons">&#xE254;</i></a>
<a href="index.php?url=products/delete&code=<?=$produit['code']?>" class="delete" title="Delete"
data-toggle="tooltip"><i class="material-icons">&#xE872;</i></a>
</td>
```

Ajouter les actions delete et edit dans le controllerProduit

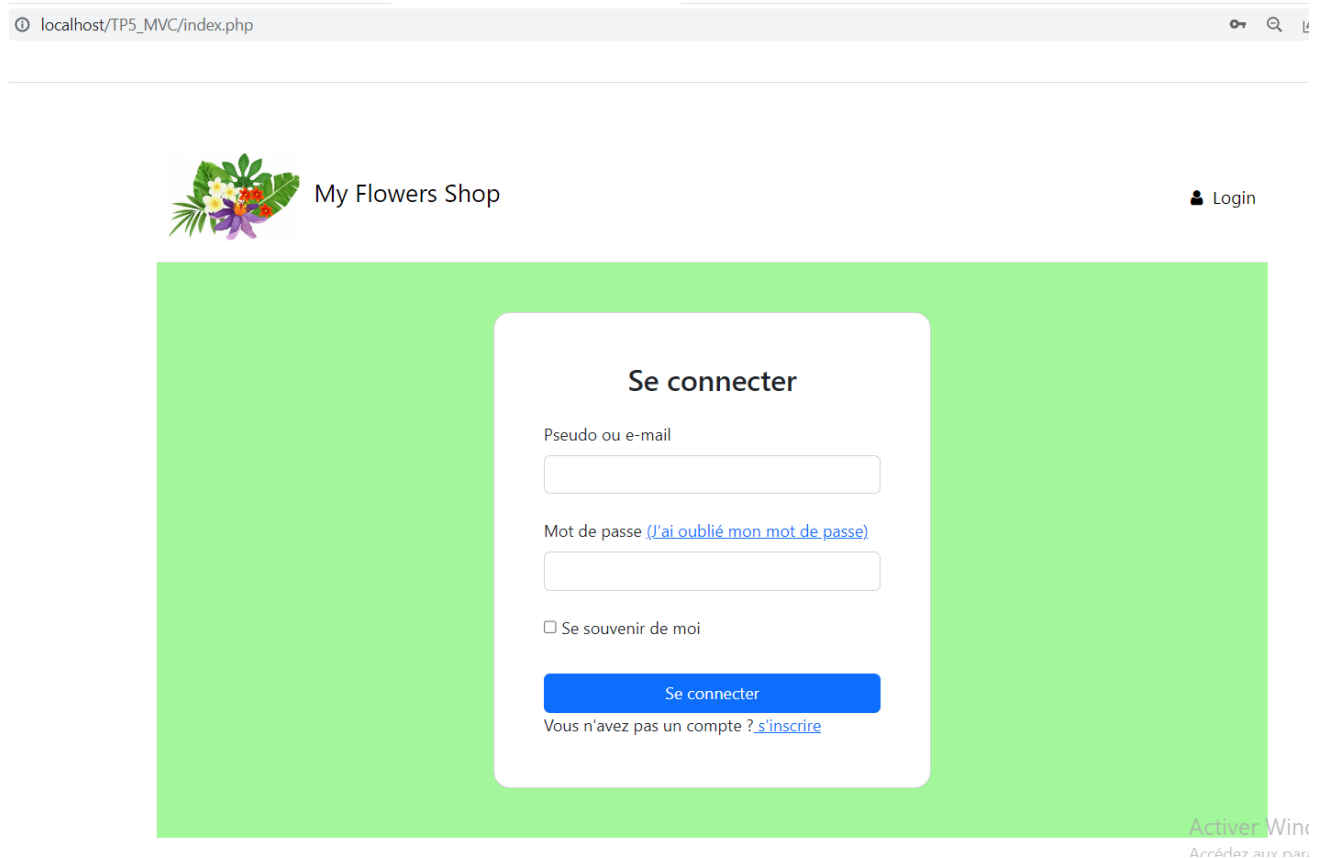
```
public function edit(){
    $db = Database::getInstance()->getConnection();
    $modelCategorie = new Model ($db, "categorie");
    $categories=$modelCategorie->findAll();
    $code = $_GET['code'];
    $p = $this->model->find($code);
    require ('app\views\produit\update.php');//redirige vers la vue
}
public function delete(){
    $code = $_GET['code'];
    $status = $this->model->delete($code);
    if (!$status){
        $message='prodnotfound';
        require ('app\views\produit\error.php');//redirige vers la vue
    }
    else{
        $this->index();
    }
}
}
```

Le traitement du formulaire de MAJ pourra être createProcess (le même que celui pour le formulaire d'ajout) avec quelques rectifications.

7. Vue "accueil" et gestion de la session

Lorsque l'utilisateur tape « index.php » dans la barre de navigation, alors le contrôleur par défaut est **ControllerHome** et l'action est **index**

Dans le ControllerHome, l'action index doit rediriger vers la vue "login.php" si l'utilisateur n'est pas authentifié (cad la session n'est pas encore créée).



Pensez alors à ajouter une méthode `loggedOnly()` appelé au début de l'action index (et aussi à toutes les autres actions de contrôleur afin d'interdire l'accès à une vue sans authentification

```
<?php
require_once 'app\models\ModelUser.php';
require_once 'app\models\Model.php';
require_once 'app\models\Database.php';
require 'app\controllers\Controller.php';
require_once 'app\views\View.php';

class ControllerHome extends Controller{
    private $model;
```

```

public static function loggedOnly(){
    if(session_status() == PHP_SESSION_NONE){
        session_start();
    }
    if(!isset($_SESSION['auth'])){
        require ('app/views/Home/login.php'); //redirige vers la vue
        exit();
    }
}
public function index() {
    self::loggedOnly();
    $controller = "Home";
    include('app/views/Home/acceuil.php');
}
public static function loginProcess(){
    session_start();
    $controller='Home';
    // à compléter

public static function logout(){
    session_start();
    unset($_SESSION['auth']);
    $_SESSION['flash']['success'] = "Vous êtes maintenant déconnecté";
    require ('app/views/Home/login.php');
}

public function create(){}
public function edit(){}
public function delete(){}
}

```

Si l'utilisateur est authentifié, l'action accueil va charger la vue `acceuil.php` :

```

<?php require "app/views/header.php"; ?>
<?php require "app/views/navigation.php"; ?>
<div id="outer" class="container d-flex align-items-center justify-content-center">
    <div id="inner">
</br></br></br></br>
        <h1>Bienvenue <?php echo($_SESSION['auth']['nom']." ".$_SESSION['auth']['prenom']);?></h1>
    </div>
</div>

<?require "app/views/footer.php";?>

```



Bienvenue Mohamed Tounsi

7. AJAX

Penser à ajouter :

1. une action `search()` dans le contrôleur Produit (à compléter)
2. Une méthode `findbyDesignation($c)` dans le model Produit
3. Une vue qui va remplacer l'élément HTML `<tbody>` à chaque fois l'utilisateur tape un caractère dans la zone de recherche
4. La route dans le routeur :

```
$router->addRoute('/products/search', 'ControllerProduit@search');
```