RÉSUMÉ DE PROJET BOUTIQUE EN LIGNE

Sommaire

I. INTRODUCTION

Le projet Boutique en ligne concerne la création d'un site de vente de vêtements. Il s'agit du projet le plus complet regroupant le plus de compétences requises et de technologies.
Dans ce projet nous devrons respecter ces consignes :

- Maquetter l'application
- Concevoir une base de données avec la méthode Merise (MCD/MLD/MPD)
- Faire de la POO (Programmation Orientée Objet) : Utiliser les classes
- Détailler un parcours utilisateur sur une fonctionnalité "métier" de votre site
- (action d'achat ...)
- Structurer un projet et penser son architecture
- Faire de l'asynchrone avec JS
- Pitcher un projet : expression orale / réalisation de slides de présentation

Dans ce site, il y a la possibilité de consulter les produits disponibles pour n'importe quelle personne dans la page d'accueil. Il existe en plus des fonctionnalités accessibles uniquement aux administrateurs et aux utilisateurs, à savoir :

- pour **l'admin** :
  1) La possibilité de gérer les produits, c'est-à-dire les mettre à jour, les supprimer ou encore en ajouter.
- pour **l'utilisateur** (qui doit être connecté) :
  1) L'ajout de produits en choisissant la quantité dans un panier qui leur est à disposition et qui enregistre chaque ajout. Dans ce même panier il peut procéder à l'achat ou supprimer un produit ajouté s'il le souhaite.
  2) La modification de ses coordonnées
  3) Regarder son historique d'achat

Dans ce compte rendu, j'expliquerai en détails avec quels moyens technologiques j'ai pu réaliser ce projet.

II. CONCEPTION BASE DE DONNÉES

1) MCD

La première étape de ce projet concernait la conception de la base de données. Et donc pour ce faire, j'ai opté pour la méthode MERISE en modélisant le MCD (modèle conceptuel de données) et le MLD (modèle logique de données).
Commençons par expliquer le MCD.
Il a fallu tout d'abord identifier les ***entités*** du site, en déduire ensuite les ***relations*** qui existent entre elles.

Pour cette application, j'en ai déduit 5 :

- Produits
- Admins
- Utilisateurs
- Paniers
- Items panier
- Achats

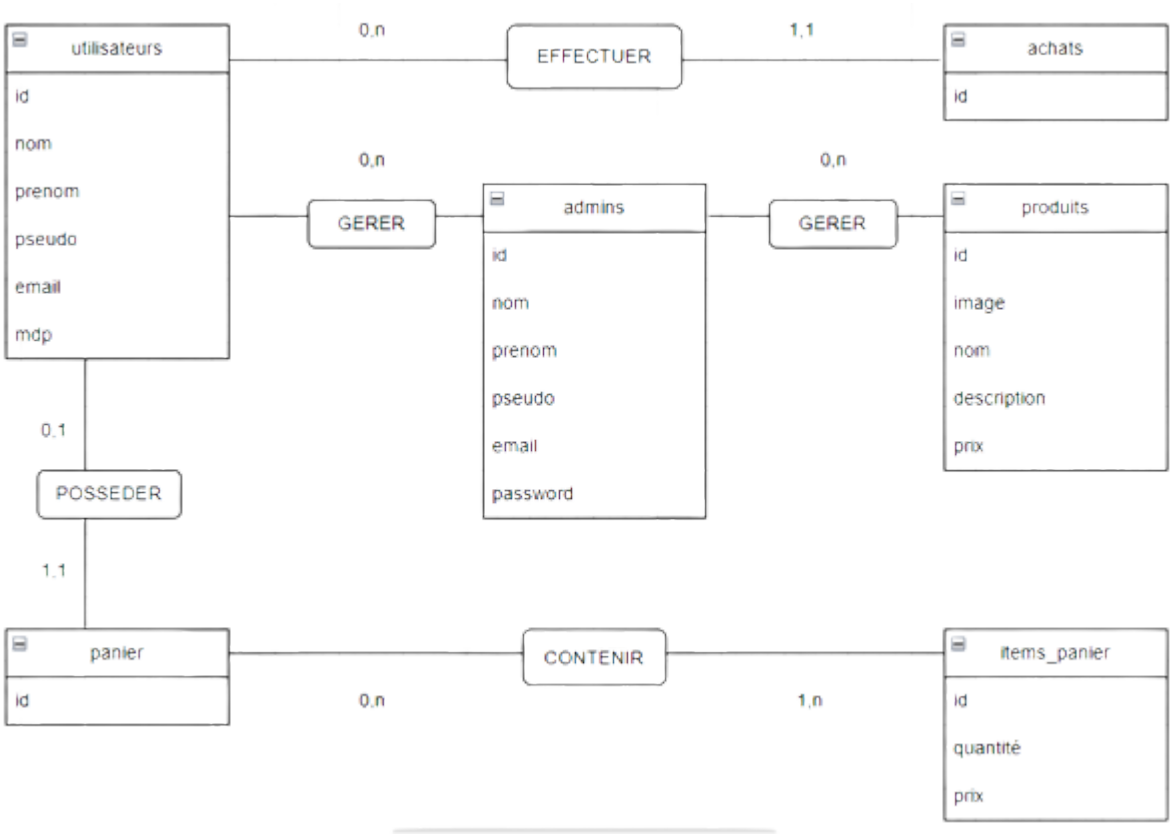
Maintenant que les entités sont définies, il a fallu définir chacune de leurs attributs :

- l'entité Produits possède un identifiant, une image, un nom, une description et un prix
- l'entité Admins possède un identifiant, un prénom, un nom (de famille), un pseudo, un email et un mot de passe.
- l'entité Utilisateurs possède un identifiant, un prénom, un nom (de famille), un pseudo, un email et un mot de passe.
- l'entité Paniers possède un identifiant, cela sert à identifier le panier disponible pour chaque utilisateur.
- l'entité Items Panier concerne les produits contenus dans un panier et possède comme attributs un identifiant, une quantité (du produit ajouté) et un prix (total)
- l'entité Achats qui possède un identifiant et une date (qui correspond à la date de l'achat)

Ensuite on définit les relations entre ces entités, l'action qui détermine la relation, ainsi que les cardinalités. En ce qui concerne les relations, il y en a 5 :

- La relation Utilisateurs -> Achats (un utilisateur **effectue** un ou plusieurs achats (1,n) et un achat est effectué par un seul utilisateur (1,1))
- La relation Admins -> Utilisateurs et Admins -> Produits (admin gère un ou plusieurs utilisateurs (1,n) et un admin gère un ou plusieurs produits (1,n))
- La relation Panier -> Items panier (un panier **possède** un ou plusieurs items (1,n) et un item_panier appartient à un seul panier)
- La relation Utilisateurs -> Paniers où un utilisateur possède un seul et unique panier (1,1) et un panier possède un seul et unique utilisateur (1,1).
-

<u>Modèle conceptuel de données</u>



2) MLD

A partir de là nous allons déduire le MLD (modèle conceptuel de données), en définissant les clés primaires et étrangères de chacune de nos entités ainsi que le type de chaque attribut. Egalement, nous allons simplifier les relations entre les entités ce qui nous donne ceci :

- Un produit peut être acheté 1 ou n fois
- Un produit peut-être ajouté 1 ou n fois dans les items du panier
- Un utilisateur peut effectuer 1 ou n achats.
- Un panier possède 1 ou n items
- Un utilisateur possède 1 et 1 seul panier.

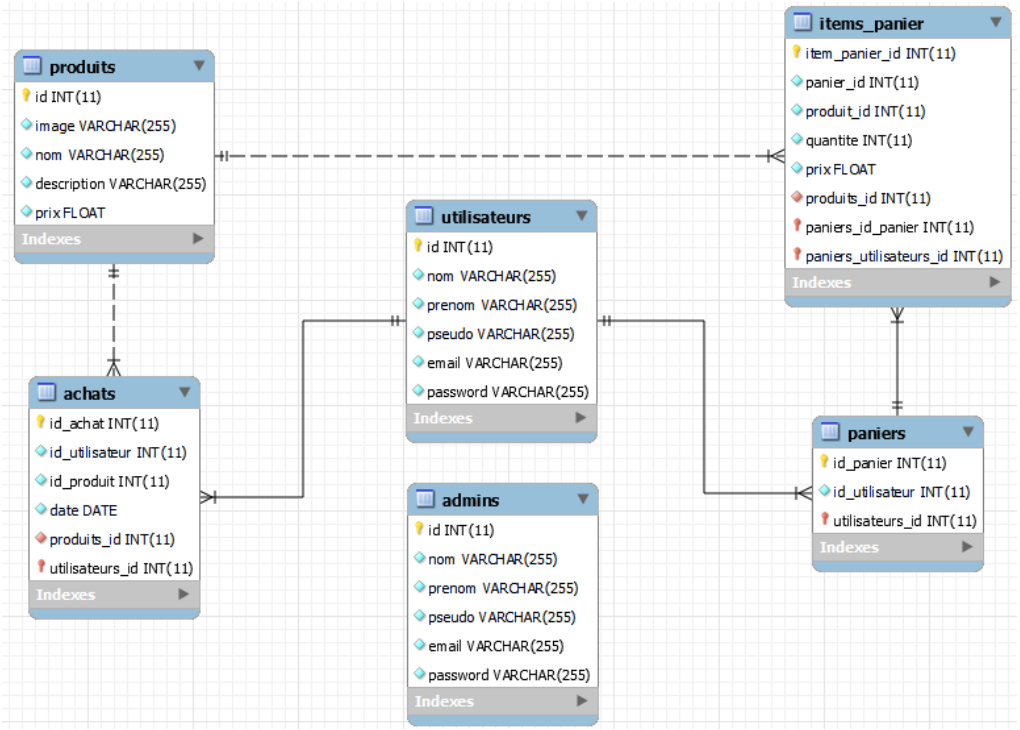
En ce qui concerne les clés primaires/étrangères (de type INT) on a dans chaque entité :

- Produits : id (clé primaire)
- Achats : id_achat (clé primaire), id_utilisateur (clé étrangère), id_produit (clé étrangère)
- Admins : id (clé primaire)
- Items Panier : items_panier_id (clé primaire), panier_id (clé étrangère), produit_id (clé étrangère)
- Paniers : id_panier (clé primaire), id_utilisateur (clé étrangère)
- Utilisateurs : id (clé primaire)

Ensuite on déduit les types de chaque attribut de nos entités qui deviennent des tables :

- Utilisateurs : id(INT), image(VARCHAR), nom(VARCHAR), prenom(VARCHAR), pseudo(VARCHAR), email(VARCHAR), password(VARCHAR)
- Admins : id(INT), image(VARCHAR), nom(VARCHAR), prenom(VARCHAR), pseudo(VARCHAR), email(VARCHAR), password(VARCHAR)
- Produits : id(INT), image(VARCHAR), nom(VARCHAR), description(VARCHAR), prix(FLOAT)
- Paniers : id_panier(INT), id_utilisateur(INT)
- Items_Panier : item_panier_id(INT), panier_id(INT), produit_id(INT), quantité(INT), prix(FLOAT)

- Achats : id_achat(INT), id_utilisateur(INT), int_produit(INT), date(DATE)

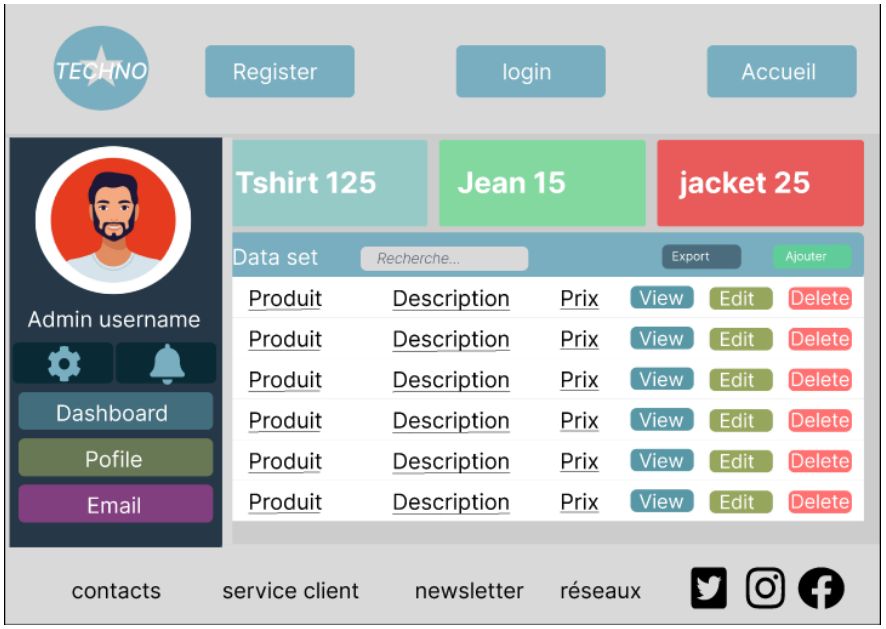


Modèle logique de données

III. MODELISATION DU SITE

On passe maintenant à la modélisation du site, et ce en utilisant Figma. Figma est un site internet qui nous permet de maquetter notre site. Cela sert à avoir une base côté front : une interface utilisateur stable au niveau des couleurs, du footer, du header etc que je pourrai utiliser dans mon code. Ci-dessous quelques exemples :

Page admin :



Page d'inscription :

IV. ARCHITECTURES ET TECHNOLOGIES UTILISÉES

Après avoir conçu la base de données et maquetté le site, on peut désormais passer au code.

On va tout d'abord définir ce qui a été utilisé pour le code :

Côté front j'ai opté pour HTML et CSS pour l'interface utilisateur/admin, JS a été utilisé pour le traitement asynchrone des formulaires, quelques évènements ainsi que l'autocompletion le tout avec fetch()

Back : PHP pour la gestion des sessions, les vérifications des formulaires en back, le CRUD de l'admin et SQL pour l'exécution des requêtes pour que ce soit la récupération des données. Le tout en optant pour la POO.

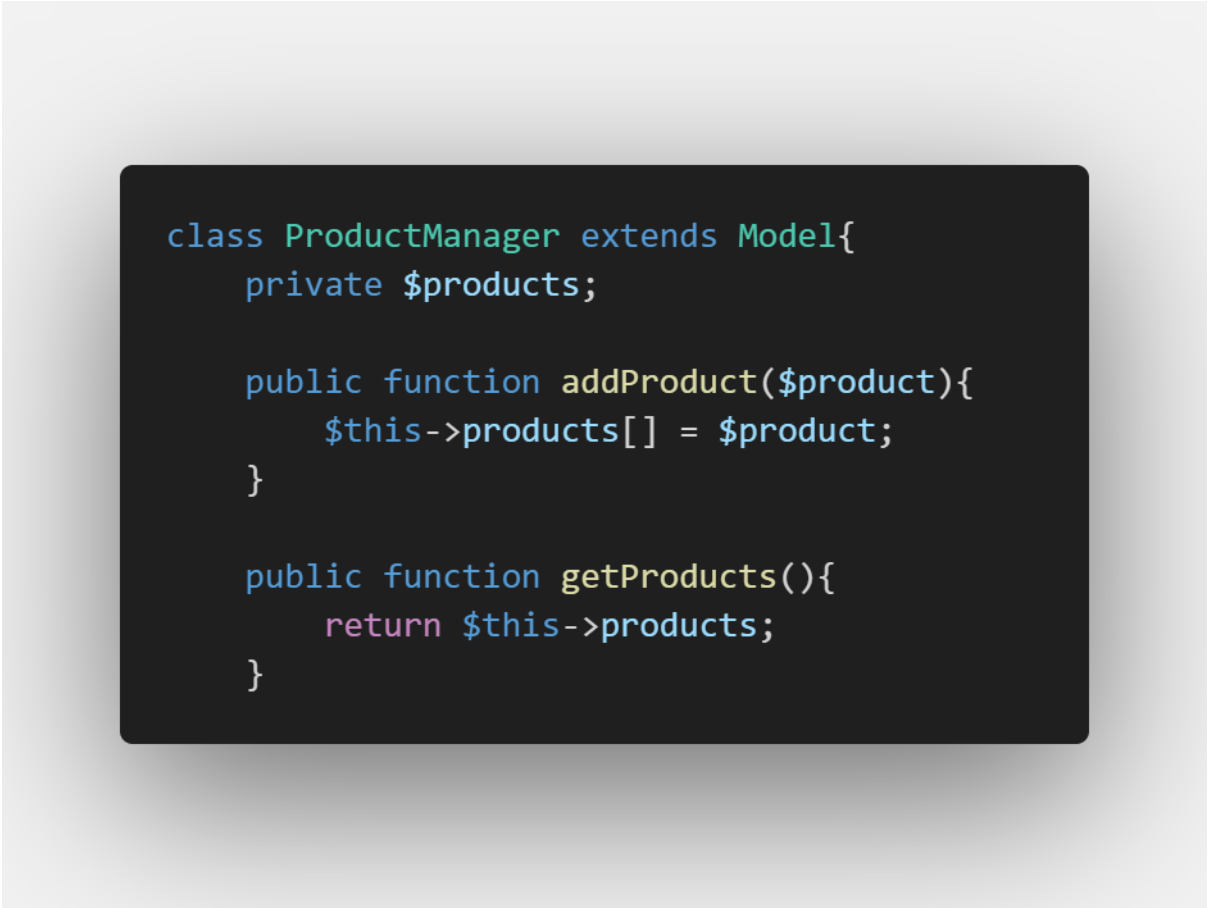
SGBD : MySQL pour la gestion de la base de données

La définition des tables m'a permis de structurer mon code dans l'usage des classes en langage ***PHP***. En effet j'ai implémenter deux classes par table (exemple pour la table "produits", deux classes :

- Classe *Product* : qui initialise les attributs de la classe (id, nom, image, description, prix) avec un constructeur. Il y a aussi un getter et un setter pour chaque attribut.

```php
class Product implements JsonSerializable {
    private $id;
    private $image;
    private $name;
    private $description;
    private $price;
```

- Classe *ProductManager* : qui comporte tout d'abord toutes les méthodes que je décris comme étant "*élémentaires*". En général, ce sont des méthodes qui concernent la gestion de l'entité en question. On peut donc par exemple avoir des méthodes qui ajoutent un produit en base de données, une méthode qui retourne sous forme d'un tableau tous les produits disponibles, une méthode qui retourne un produit en fonction de son id etc.



```php
class ProductManager extends Model{
    private $products;

    public function addProduct($product){
        $this->products[] = $product;
    }

    public function getProducts(){
        return $this->products;
    }
```

Ici la méthode addProduct ajoute un produit dans le tableau contenant tous les produits.
J'ai opté pour une architecture MVC (Modèle, Vue, Contrôleur) dans mon code pour avoir un code plus structuré et propre. Désormais par exemple, les traitements de formulaires (via les requêtes/vérifications de champs) ne se feront pas dans la même page de vue.
L'avantage en outre est que cela permet de mieux localiser les problèmes de code s'il y en a.

Exemple d'usage du MVC :

Cette vue, concerne la page admin où on ajoute un produit. On voit où est-ce qu'on est redirigé dès qu'on envoie le formulaire avec action c'est-à-dire admin/ev (on en reparlera plus bas avec les routes).

```
<form enctype="multipart/form-data" method="post" action="<?= URL ?>admin/ev">
```

Ici le contrôleur fait un appel à une méthode du modèle (ProductManager.php) prend en paramètre l'id du produit et redirige vers la page de modification.

```
public function editProduct($id){
        $product = $this->productManager->getProductbyId($id);
        require "views/admin/editProduct.view.php";
    }
```

Ici la méthode du modèle, qui récupère l'id d'un produit :

```
public function getProductbyId($id){
        for($i=0; count($this->products); $i++){
            if($this->products[$i]->getId() === $id){
                return $this->products[$i];
            }
        }
    }
```

V. GESTION DES ROUTES

L’ architecture MVC existe avec également la gestion des routes. En gros on réécrit les URLs tapés et on assigne pour certaines URLs spécifiques comme plus ‘admin/ev’ une fonction du contrôleur. La réécriture s’active grâce au fichier .htaccess qui active le module de réécriture :

```
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

RewriteRule ^(.*)$ index.php?page=$1
```

Que veulent dire ces lignes ?

Les lignes “ Rewrite Cond “ utilisent “ %{REQUEST_FILENAME} ” qui est une variable d'environnement qui contient le chemin du fichier demandé dans la requête HTTP actuelle. La condition "!-f" vérifie si le fichier n'existe pas. Donc, la règle de réécriture de la dernière ligne ne sera appliquée que si le fichier demandé n'existe pas sur le serveur.

La dernière ligne redirige enfin toutes les requêtes (sauf celles pointant vers un fichier existant) vers "index.php".

Maintenant qu’on a géré les requêtes qui par défaut ne correspondent pas à une route spécifique, il faut gérer les URL qui sont valides.
Pour ce faire, on définit la constante URL dans index.php, et on gère les requêtes GET.

```
define("URL", str_replace("index.php","",(isset($_SERVER['HTTPS']) ? "https" : "http").
"://$_SERVER[HTTP_HOST]$_SERVER[PHP_SELF]"));
```

- define("URL", ...) : Cela définit une constante appelée "URL" qui va stocker la valeur calculée à droite de l'expression

- str_replace("index.php","", ...) : Cette fonction recherche le texte "index.php" dans la chaîne et le remplace par une chaîne vide (""). Cela permet de supprimer le "index.php" de l'URL, s'il est présent.

- (isset($_SERVER['HTTPS']) ? "https" : "http") : C'est une expression conditionnelle qui vérifie si la variable $_SERVER['HTTPS'] est définie. Si c'est le cas, on utilise le protocole HTTPS, donc "https" est renvoyé, sinon "http" est renvoyé.

- "://$_SERVER[HTTP_HOST]$_SERVER[PHP_SELF]" : Cette partie de la chaîne construit l'URL de base en utilisant des variables du serveur PHP. $_SERVER['HTTP_HOST'] contient le nom de domaine de la requête actuelle, et $_SERVER['PHP_SELF'] contient le chemin vers le script PHP actuel.

Ainsi en réalisant chacune de ces étapes, ce bout de code calcule l'URL de base du site en supprimant éventuellement "index.php" du chemin et en déterminant le protocole utilisé (HTTP ou HTTPS) et le nom de domaine du serveur. Le résultat est ensuite stocké dans la constante

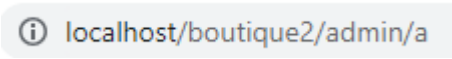
"URL" afin que l’on puisse l’utiliser dans le code du site, comme dans la définition des liens <a href>.

```php
try{
    if(empty($_GET['page'])){
        require "views/home.view.php";
    }
        else {
            $url = explode("/", filter_var($_GET['page']), FILTER_SANITIZE_URL);
```

Ce code gère les demandes d'URL pour une application web. S'il n'y a pas de paramètre 'page' dans l'URL, il affiche la page d'accueil. Sinon, il traite l'URL en utilisant "explode()" et "filter_var()" pour effectuer une action spécifique basée sur la valeur du paramètre 'page'.

Le rendu dans le site est que l’on peut de ce fait directement accéder à certaines pages en tapant des mots clés dans la barre de recherche comme sur l’exemple ci- dessous :

Si je tape “a” alors le contrôleur va réaliser la méthode addProduct() qui consiste à nous rediriger vers la page d’ajout d’un article côté administrateur.



Barre URL

Méthode du contrôleur qui redirige vers la page d'ajout d'un produit

Maintenant que nous avons détaillé le modèle MVC et comment compléter cela avec l'usage d'un routeur, on va montrer quelques fonctionnalités du site ainsi que la façon dont elles fonctionnent au niveau du code.

VI. FONCTIONNALITÉS DU SITE

1) CRUD Admin

Un CRUD (Create Read Update Delete) est mis en place dans le but de gérer les produits chez l'administrateur. Pour ce faire, il faut d'abord définir ces produits en utilisant la POO (programmation orientée objet).

Ainsi avec la classe ProductManager.php nous allons taper des méthodes qui serviront de se connecter à notre base de données, le tout grâce à la méthode Model qui fait office de classe abstraite :

```php
abstract class Model{
    private static $pdo;

    private static function setDb(){
        self::$pdo = new PDO("mysql:host=localhost; dbname=shop; charset=utf8", "root",
"");
        self::$pdo->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_WARNING);
    }

    protected function getDb(){
        if(self::$pdo === null){
            self::setDb();
        }
        return self::$pdo;
    }
}
```

Il suffira donc d'ajouter un extends Model dès la déclaration d'une classe où on voudra se connecter dans notre base de données.

Dans le cas d'une modification d'un produit, après avoir effectué en amont les vérifications côté contrôleur (est-ce que les champs sont bien remplis etc) et que nous sommes redirigé vers le modèle voici ce qui va se passer :

```php
public function editProductDb($id, $image, $name, $description, $price){
        $query = "UPDATE produits SET image = :image, nom = :name, description = :
description, prix = :price WHERE id = :id";
        $stmt = $this->getDb()->prepare($query);
        $stmt->bindValue(":id", $id,PDO::PARAM_INT);
        $stmt->bindValue(":image", $image,PDO::PARAM_STR);
        $stmt->bindValue(":name", $name,PDO::PARAM_STR);
        $stmt->bindValue(":description", $description,PDO::PARAM_STR);
        $stmt->bindValue(":price", $price,PDO::PARAM_INT);
        $result = $stmt->execute();
        $stmt->closeCursor();

        if ($result > 0) {
            $this->getProductbyId($id)->setImage($image);
            $this->getProductbyId($id)->setName($name);
            $this->getProductbyId($id)->setDescription($description);
            $this->getProductbyId($id)->setPrice($price);
        }
    }
```

Nous faisons une requête qui met à jour la valeur tapée de chaque champ et ensuite si la requête (changement) est effectuée on change les valeurs au sein même de la variable globale product.

Toujours dans le CRUD on emploie la même logique qui consiste à faire un appel au contrôleur dès que l'admin réalise une action sur sa vue, pour qu'ensuite le contrôleur utilise une méthode du modèle pour modifier ou supprimer un produit.

Edit Product View :

```php
<form enctype="multipart/form-data" method="post" action="<?= URL ?>admin/ev">
```

Edit Product Index :

```
else if($url[1] === "e"){
                        $productController->editProduct($url[2]);
                    }
```

Edit Product Controller :

```
public function editProduct($id){
        $product = $this->productManager->getProductbyId($id);
        require "views/admin/editProduct.view.php";
    }
```

Edit Product Model :

```
public function editProductDb($id, $image, $name, $description, $price){
        $query = "UPDATE produits SET image = :image, nom = :name, description = :
description, prix = :price WHERE id = :id";
        $stmt = $this->getDb()->prepare($query);
        $stmt->bindValue(":id", $id,PDO::PARAM_INT);
        $stmt->bindValue(":image", $image,PDO::PARAM_STR);
        $stmt->bindValue(":name", $name,PDO::PARAM_STR);
        $stmt->bindValue(":description", $description,PDO::PARAM_STR);
        $stmt->bindValue(":price", $price,PDO::PARAM_INT);
        $result = $stmt->execute();
        $stmt->closeCursor();

        if ($result > 0) {
            $this->getProductbyId($id)->setImage($image);
            $this->getProductbyId($id)->setName($name);
            $this->getProductbyId($id)->setDescription($description);
            $this->getProductbyId($id)->setPrice($price);
        }
    }
```

2) Autocomplétion

L'utilisateur peut rechercher les produits disponibles dans la page boutique via une barre de recherche. Avec JavaScript, on créé un évènement input et on utilise fetch pour récupérer la route user/sa qui enclenche cette méthode côté contrôleur :

Tout d'abord on récupère les variables dont on a besoin :

```
const searchInput = document.getElementById('search-input');
const autoCompletionMsg = document.getElementById('autocompletion-msg');
const bar = document.getElementById('bar')
const container = document.querySelector('.container');
const originalTableContent = container.innerHTML;
```

Puis on récupère la variable contenant les données des produits qui sont récupérés sous format JSON :

```
public function autoCompletionProducts(){
        $products = $this->productManager->getProducts();

        header('Content-Type: application/json');
        echo json_encode($products);
    }
```

Ensuite on code l'évènement (on remarque que productsData contient justement les données des produits que l'on souhaite récupérer) :

```
searchInput.addEventListener('input', function() {
    const searchTerm = searchInput.value.trim().toLowerCase();
    fetch('http://localhost/boutique2/user/sa')
        .then(response => response.json())
        .then(productsData => {
            const matchedProducts = productsData.filter(product => {
                const productName = product.name.toLowerCase();
                return productName.includes(searchTerm);
            });
            console.log(matchedProducts);
```

Ici les produits recherchés sont affichés en console, pour les afficher sur la page on manipule le DOM pour réécrire les nouveaux résultats tout en appliquant le CSS :

```
if (matchedProductsLength > 1) {
                autoCompletionMsg.innerHTML = matchedProductsLength + ' résultats trouvés'
;
                console.log('Produits trouvés: ' + matchedProductsLength);

                matchedProducts.forEach(product => {
                    const productName = product.name;
                    const productDescription = product.description;
                    const productPrice = product.price;


                    const table = document.createElement('table');
                    table.classList.add('table', 'product-table');
```

S'il y a plus d'un produit trouvé, on ajoute une ligne qui affiche le nombre de produits trouvés et pour chaque produit trouvé - avec forEach car les produits sont affichés avec une boucle for - on ajoute une table contenant le nom, prix et description du produit.
On utilise une table car les produits ajoutés dans la vue sont originellement sous forme de table.

```
container.innerHTML = '';
            bar.classList.add('bar')
            container.appendChild(bar);
            container.appendChild(autoCompletionMsg)

            if (searchTerm === '') {
                autoCompletionMsg.innerHTML = '';
            }

            else if (searchTerm !== '') {

            }
```

Ici on recharge les données de la table à chaque nouvelle recherche

```
table.appendChild(newRow);

                container.appendChild(table);
```

```
const newRow = document.createElement('tr');
                newRow.classList.add('table-content');
                newRow.innerHTML = `
```

On ajoute ici le nouveau tableau contenant tous les produits dans le container HTML tout en oubliant pas d'appliquer le CSS pour garder le même design avec **classList.add**

3) Gestion des paniers

Tout d'abord nous avons une méthode dans notre contrôleur qui récupère les données du panier et de ses produits :

```php
public function getUserCart() {
        if (!empty($_SESSION['username'])) {

            $userCart = $this->cartManager->getCartByUserId($_SESSION['id']);

            $cartItems = $this->cartItemsManager->getCartItemsByCartId($userCart->getCartId());
            $products = [];
```

Ici on retourne les données des produits pour chaque utilisateur.

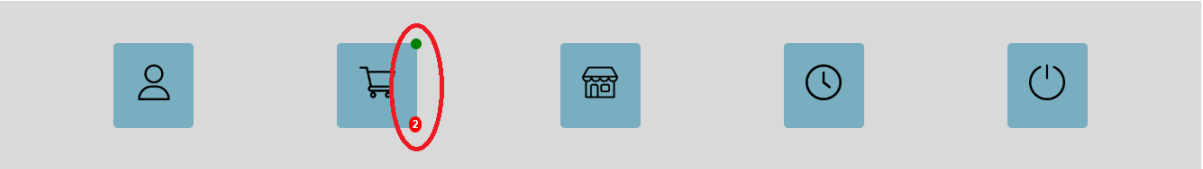
Si un produit est trouvé on récupère son Id et la quantité choisie :

```php
foreach ($cartItems as $cartItem) {
    $product = $this->productManager->getProductById($cartItem->getProductId());

    if ($product) {
        $productData = [
            'productId' => $product->getId(),
            'quantity' => $cartItem->getQuantity()
        ];

        $products[] = $productData;
    }
}
```

Ensuite on retourne le panier de l'utilisateur avec ses items et on convertit cela en JSON pour manipuler les données en JS :

```php
$responseData2 = [
    'userCart' => $userCart,
    'products' => $products
];

header('Content-Type: application/json');
echo json_encode($responseData2);
exit();
```

On manipule nos données en JS car on veut, de façon dynamique, mettre à jour l'état du panier : il y a un bouton rouge qui signale que le panier est vide qui s'allume en vert s'il ne l'est plus et il y a un deuxième bouton qui représente le nombre de produits ajoutés :



Les deux boutons sont activés grâce à ces deux fonctions JS qui avec fetch récupère les données de la méthode présentée en haut :

```javascript
const incrementRoundBtns = (products) => {
  Array.from(BookmarkBtns).forEach(btn => {
    const productId = btn.dataset.productId;
    const cartItem = products.find(item => item.productId === productId);
    if (cartItem) {
      const quantity = cartItem.quantity;
      const roundBtn = btn.nextElementSibling.nextElementSibling;
      roundBtn.textContent = quantity;
      roundBtn.style.display = 'block';
    }
  });
};


const updateItemCount = (count) => {
  ItemCount.textContent = count.toString();
};
```

```
const RoundBtnCart = document.getElementById("round-btn");


const checkUserCart = () => {
  fetch('http://localhost/boutique2/user/gc')
    .then(response => response.json())
    .then(userCart => {
      RoundBtnCart.style.backgroundColor = userCart ? 'green' : 'red';
    })
    .catch(error => {
      console.log('Error:', error);
    });
};
```

4) Ajout de produits dans le panier

Lorsque l'on ajoute un produit au panier, les boutons sont également mis à jour :

```
.then(response => response.json())
.then(result => {
  if (result.success) {
    const count = parseInt(ItemCount.textContent) + 1;
    updateItemCount(count);
    RoundBtnCart.style.backgroundColor = 'green';
  } else {
    console.log('Ajout non réalisé.');
  }
})
```

```
Array.from(BookmarkBtns).forEach(btn => {
  btn.addEventListener('click', () => {
    const productId = btn.dataset.productId;
    const price = btn.dataset.price;
    addProductToCart(productId, price);
  });
```

Ensuite on ajoute l'Id du produit et le prix qu'on cible avec HTML.

Maintenant comment cela se passe avec le MVC, étant donné que l'on veut ajouter les items en base de données ?

Méthode du modèle où on prend 4 arguments qui correspondent à des données récupérées côté serveur.

```
private function insertCartItem($cartId, $productId, $quantity, $price) {
        $query = "INSERT INTO items_panier (panier_id, produit_id, quantite, prix) VALUES
 (:cartId, :productId, :quantity, :price)";
        $stmt = $this->getDb()->prepare($query);
        $stmt->bindValue(":cartId", $cartId, PDO::PARAM_INT);
        $stmt->bindValue(":productId", $productId, PDO::PARAM_INT);
        $stmt->bindValue(":quantity", $quantity, PDO::PARAM_INT);
        $stmt->bindValue(":price", $price, PDO::PARAM_INT);
        $result = $stmt->execute();

        if ($result > 0) {
            $cartItem = new CartItems($this->getDb()->lastInsertId(), $cartId, $productId, $quantity, $price);
            $this->addCartItem($cartItem);
        }
    }
```

```
function registerCartDb($user_id){
        $query = "INSERT INTO paniers (id_utilisateur) values (:id_utilisateur)";

        $stmt = $this->getDb()->prepare($query);
        $stmt->bindValue(":id_utilisateur", $user_id, PDO::PARAM_INT);
        $result = $stmt->execute();

        if ($result > 0){
            $cart = new Cart($this->getDb()->lastInsertId(), $user_id);
            $this->addCart($cart);
        }
    }
```

Ici l’ajout d’un nouveau panier en base de données.

```
else if($url[1] === "ac") {
                    $cartItemsController->addProductToCartItems();
                }
```

Et là, la route qui redirige vers le contrôleur des items panier qui ajoute un produit dans le panier, contrôleur qui comprend cette méthode :

```
public function addProductToCartItems() {

    $cart = $this->cartManager->getCartbyUserId($_SESSION['id']);

    if ((empty($cart))){
      $this->cartManager->registerCartDb($_SESSION['id']);
    }

    $cartId = $cart->getCartId();
    $productId = $_POST['productId'];
    $quantity = $_POST['quantity'];
    $price = $_POST['price'];

    $this->cartItemsManager->addItemToCart($cartId, $productId, $quantity, $price);

    $response = [
        'success' => true,
        'cartId' => $cart,
    ];

    header('Content-Type: application/json');
    echo json_encode($response);
  }
```

Ici on récupère l’Id du panier et via un formulaire et avec la méthode POST on récupère l’id du produit, la quantité et le prix.

VII. SÉCURITÉ ET OPTIMISATION

Dans une application, il faut avoir des mesures de sécurité contre de potentielles attaques XSS ou d'injections SQL. Les injections SQL se font via des requêtes dans l'URL ou les entrées du site. Le pirate génère un code SQL qui lui permet d'accéder à nos données ce qui nous expose à la récupération ou la suppression de données personnelles entre autres.

Les attaques XSS consistent à taper côté serveur un script en JS afin que le client l'exécute ensuite à son insu. Dès lors, le pirate commence par écrire un bout de code JavaScript afin de le rediriger vers un site pirate.

Au niveau de la sécurité, j'ai effectué quelques ajouts pour me protéger le plus possible :

- Au niveau des requêtes SQL j'ai utilisé bindValue et prepare et j'ai hashé les mots de passe côté serveur. Le but est de me protéger des injections
- Pour éviter les attaques XSS, j'ai utilisé html_special_chars() pour enlever les caractères spéciaux en HTML
- Au niveau des sessions, une condition dans les pages concernés qui empêchent l'accès à certaines pages

```php
if (!$_SESSION['username']){
        header('location: ../home');
    }
```

- Au niveau des URL filter_sanitize supprime les caractères inutiles tapés dans la barre URL

```php
$url = explode("/", filter_var($_GET['pag
e']), FILTER_SANITIZE_URL);
```