Compte Rendu E-Commerce

Parisot Clément

Table des matières

Chapitre 11 : Ajouter une catégorie et détail d'un produit

Détail d'un Produit

Pour afficher de détail d'un produit, on commence par créer un nouveau Controller ainsi s'une page Twig (sans oublier les routes) :

```php
function showProductController($twig, $db)
{
    include_once '../src/model/ProductModel.php';

    $product = null;

    if (isset($_GET['product'])) {
        $product = getOneProduct($db, $_GET['product']);
    }
    echo $twig->render('showProduct.html.twig', [
        'product' => $product
    ]);
}
```

```twig
<div class="container">
    {% if product %}
    <div class="col-12">
        <h1 class="mb-5">{{ product.label }}</h1>
    </div>
    <div class="row">
        <div class="col-md-4">
            <div class="card p-2">
                <img src="https://fakeimg.pl/350x200/?text=Image" class="img-fluid rounded-start" alt="...">
            </div>
        </div>
        <div class="col-md-5 text-secondary">
            <p>{{ product.description }}</p>
        </div>
        <div class="col-md-3 bg-light text-center p-5">
            <h5 class="h1 text-primary">{{ product.price }} €</h5>
        </div>
    </div>
    {% else %}
    <p>Aucun produit n'est défini !</p>
    {% endif %}
</div>
```

On ajoute le bouton de sélection du produit :

Carte Graphique

ASRock Radeon RX 66!

C'est vraiment une super cart

Voir le produit

Et on a ce résultat-là :

ASRock Radeon RX 6650 XT Challenger D 8GB OC

C'est vraiment une super carte graphique de oufff

600.5 €

Ajouter une Catégorie

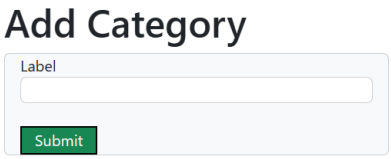
Pour ajouter la possibilité d'ajouter une catégorie, il a suffi de copier tout ce qui a été fait pour le produit et le remplacer tous les « product » par « category ». Cela pour le Controller, le model et le Twig :

```php
function getOneCategory($db, $idCategc
{
    $query = $db->prepare("SELECT id,
    $query->execute([
        'id' => $idCategory
    ]);

    $category = $query->fetch();

    return $category;
}

function getAllCategories($db)
{
    $query = $db->prepare("SELECT id,
    $query->execute([]);

    $category = $query->fetchAll();

    return $category;
}

function saveCategory($db, $label) {
    $query = $db -> prepare("INSERT IN
    return $query -> execute([
        'label' => $label
    ]);
}
```

=>

```php
function getOneProduct($db, $idProduct)
{
    $query = $db->prepare("SELECT id, labe
    $query->execute([
        'id' => $idProduct
    ]);

    $product = $query->fetch();

    return $product;
}

function getAllProducts($db)
{
    $query = $db->prepare("SELECT id, labe
    $query->execute([]);

    $product = $query->fetchAll();

    return $product;
}

function saveProduct($db, $label, $descrip
    $query = $db -> prepare("INSERT INTO p
    return $query -> execute([
        'label' => $label,
        'descr' => $description,
        'price' => $price,
        'idCategory' => $category
    ]);
}
```

On a donc ce résultat où on peut ajouter les catégories :



Pour ensuite avoir dans la page « addProduct » toutes les catégories ajoutées automatiquement, il suffit d'ajouter dans le Controller la commande créée pour récupérer toutes les catégories :

```php
echo $twig->render('addProduct.html.twig', [
    'form' => $form,
    'categories' => getAllCategories($db)
]);
```

Puis, dans le Twig, ajouter une boucle For dans le select :

```twig
<select name="productCategory" id="productCategory" class="form-select">
    {% for category in categories %}
    <option value="{{category.id}}">{{category.label}}</option>

    {% endfor %}
</select>
```

Ceci récupère toutes les catégories et, en passant par toutes les cases du tableau, affiche et propose les catégories tout en n'envoyant que l'ID de la catégorie, comme ce qui avait été fait auparavant.

De même dans le HomeController pour avoir la catégorie qui s'affiche au-dessus du produit :

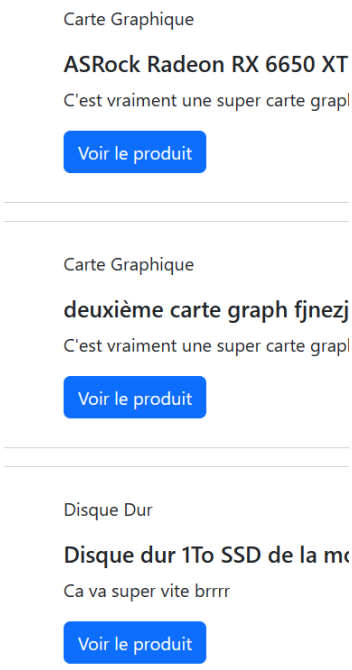
```php
function homeController($twig, $db) {

    include_once("../src/model/ProductModel.php");
    include_once("../src/model/CategoryModel.php");

    echo $twig -> render('home.html.twig', [
        'products' => getAllProducts($db),
        'categories' => getAllCategories($db)
    ]);
}
```

Puis, dans le Twig on rajoute une boucle pour détecter la catégorie du produit :

```twig
<div class="col-md-6 d-flex align-items-center">
    <div class="card-body">
        {% for category in categories %}
            {% if category.id == product.idCategory %}
                <p class="card-text">{{category.label}} </p>
            {% endif %}
        {% endfor %}
        <h5 class="card-title">{{product.label}}</h5>
        <p class="card-text">{{product.description}}</p>

        <a href="?page=showProduct&product={{ product.id }}" class
    </div>
```

En passant par toutes les catégories de « categories », si les ID concordent, afficher le nom de la catégorie au-dessus du nom du produit.

Ce qui donne ce résultat :

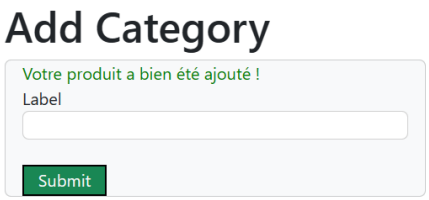


Afin d'avoir les erreurs qui s'affichent dans le Twig, il faut ajouter des conditions dans celui-ci :

```
<div class="card px-3 bg-light" style="width: 25rem;">

    {% if form.state == "success" %}
    <div style="color: green;">{{form.message}} </div>
    {% endif %}
    {% if form.state == "danger" %}
    <div style="color: red;">{{form.message}} </div>
    {% endif %}

    <label for="productLabel">Label</label>
    <input type="text" class="card" name="productLabel" id="pr
```

Si « form » signale un succès, alors afficher le message de succès, et si « form » signale un danger, afficher le message de danger.

Ce qui donne le résultat suivant dans le cas d'un succès :



Chapitre 12 : Update un produit

Afin de modifier un produit, il nous faut une nouvelle page, en commençant par le Twig :

```
</div>

<div class="mb-3">
    <label for="productDescription" class="form-label">Description</label>
    <textarea class="form-control" id="productDescription" name="productDescription"
        rows="3">{{ product.description }}</textarea>
</div>

<div class="mb-3">
    <label for="productPrice" class="form-label">Prix</label>
    <input type="number" class="form-control" id="productPrice" name="productPrice" step="0.01"
        value="{{ product.price }}">
</div>

<div class="mb-3">
    <select class="form-select" name="productCategory">
        {% for category in categories %}
        <option value="{{category.id}}">{{category.label}}</option>

        {% endfor %}
    </select>
</div>

<button type="submit" class="btn btn-primary" name="btnPostProduct">Update</button>
```

Le code permettant d'afficher toutes les catégories a également été ajouté, le tableau « categories » ayant été transmis par le Controller :

```
echo $twig->render('updateProduct.html.twig', [
    'product' => $product,
    'page' => '?page=updateProduct&product=' . $pr
    'categories' => getAllCategories($db)
]);
lse {
```

On ajoute ensuite la fonction qui permet de modifier le produit dans la base de données :

```
function updateOneProduct($db, $id, $label, $description, $price, $category)
{
    $query = $db->prepare("UPDATE product SET label=:label,
    `description`=:descr, price=:price, idCategory=:category WHERE id=:id");
    return $query->execute([
        'id' => $id,
        'label' => $label,
        'descr' => $description,
        'price' => $price,
        'category' => $category,
    ]);
}
```

Avec comme modification le bon nom de la table « product ».

Pour accéder à la page, on ajoute la route :

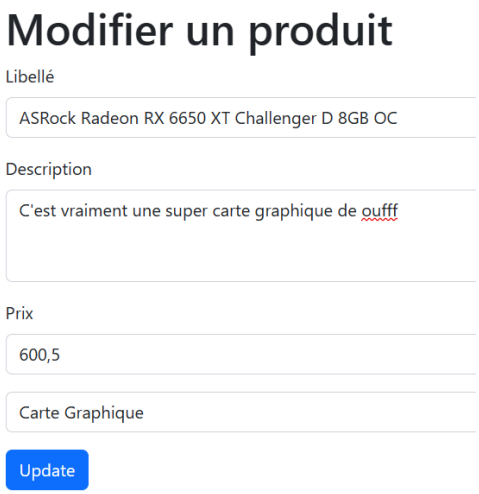
```
'addProduct' => 'addProductController',
'updateProduct' => 'updateProductController',
'addCategory' => 'addCategoryController'
```

Ainsi que le Controller en ajoutant « categories » afin de récupérer les catégories dans le Twig :

```
        $price = $_POST['productPrice'];
        if (empty($price)) {
            $price = 0.00;
        }
        $category = $_POST['productCategory'];
        if (!empty($label) && !empty($description) && !empty($category)) {
            updateOneProduct($db, $product['id'], $label, $description, $price, $cat
        }
    }
    echo $twig->render('updateProduct.html.twig', [
        'product' => $product,
        'page' => '?page=updateProduct&product=' . $product['id'],
        'categories' => getAllCategories($db)
    ]);
} else {
    echo $twig->render('home.html.twig', []);
}
}
```

Grâce à cela, on pourra avoir toutes les catégories dans la barre de sélection de la page de modification.

On se retrouve donc avec ce résultat :



Il ne manque que les affichages d'erreurs

Pour afficher les erreurs sur la page de modification de produit, il faut d'abord savoir quelles erreurs sont possibles d'afficher :

- Succès de la modification (pas vraiment une erreur mais rentre dans la même catégorie)
- Erreur lors de la modification
- Le produit n'existe pas (l'ID ne concorde avec rien qui est dans la base de données)

```php
$form = [];

if (isset($_GET['product'])) {

    $product = getOneProduct($db, $_GET['product']);

    if (isset($_POST['btnPostProduct'])) {
        $label = $_POST['productLabel'];
        $description = $_POST['productDescription'];
        $price = $_POST['productPrice'];
        if (empty($price)) {
            $price = 0.00;
        }
        $category = $_POST['productCategory'];
        if (!empty($label) && !empty($description) && !empty($category)) {
            updateOneProduct($db, $product['id'], $label, $description, $price, $category);
            $form['state'] = 'success';
            $form['message'] = 'Votre produit a bien été modifié !';
        } else {
            $form['state'] = 'danger';
            $form['message'] = 'Veuillez remplir tous les champs !';
```

On commence par ajouter le tableau « form » et les assignations en fonction de la réussite ou de l'échec.

La première association étant pour le succès, la seconde est pour vérifier que tous les champs sont remplis.

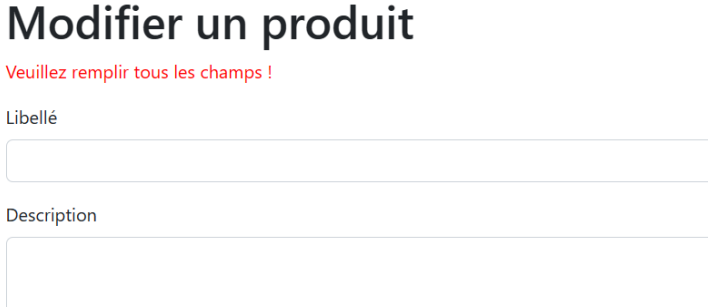
```php
    if ($product == null) {
        $form['state'] = 'non-existent';
        $form['message'] = 'Le produit n\'existe pas !';

        echo $twig->render('updateProduct.html.twig', [
            'form' => $form
        ]);
        return 0;
    }

    echo $twig->render('updateProduct.html.twig', [
        'product' => $product,
        'page' => '?page=updateProduct&product=' . $product['id'],
        'categories' => getAllCategories($db),
        'form' => $form
    ]);
} else {

    $form['state'] = 'non-existent';
    $form['message'] = 'L\'identifiant de produit est erroné !';

    echo $twig->render('updateProduct.html.twig', [
        'form' => $form
    ]);
}
```

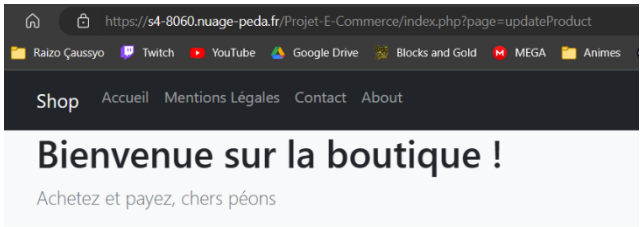
Puis on vérifie si le produit existe dans la base de données, s'il n'existe pas alors on renvoie la page d'erreur disant que le produit n'existe pas. Puis, pour pas que la fonction continue et affiche deux fois la page, on met un return 0.

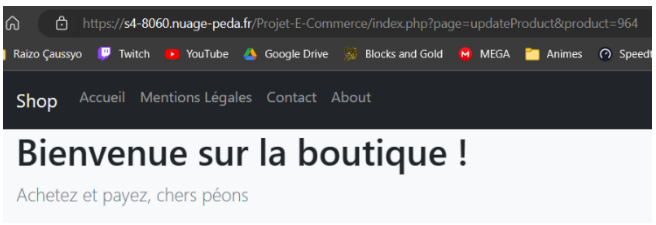
Si le produit existe, le code continue et affiche la page normale avec le produit.

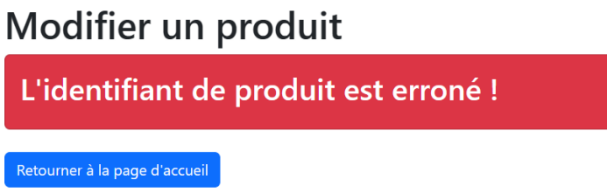
Si la variable « product » dans l'url n'existe pas, il renvoie vers une autre page d'erreur.

Les pages d'erreur « produit n'existe pas » et « url invalide » proposent un bouton renvoyant vers la page d'accueil. Ce qui nous donne ces résultats :









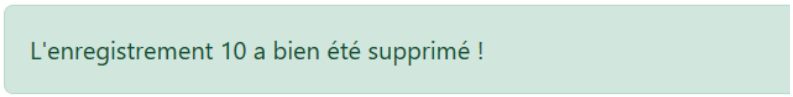
Chapitre 13 : Supprimer un produit

Afin de supprimer un produit, on doit d'abord créer le Controller et la requête SQL :

```php
function deleteOneProduct($db, $id)
{
    $query = $db->prepare("DELETE FROM product WHERE id=:id");
    $query->execute([
        'id' => $id
    ]);
    if ($query->rowCount() > 0) {
        $result = true;
    } else {
        $result = false;
    }
    return $result;
}
```

```php
function deleteProductController($twig, $db)
{

    include_once '../src/model/ProductModel.php';

    $product = deleteOneProduct($db, $_GET['product']);

    if ($product) {
        $form = [
            'state' => 'success',
            'message' => 'L\'enregistrement ' . $_GET['product'] .
        ];
    } else {
        $form = [
            'state' => 'danger',
            'message' => 'L\'enregistrement ' . $_GET['product'] .
        ];
    }
    echo $twig->render('message.html.twig', [
        'form' => $form
    ]);
}
```

En cliquant sur le bouton de suppression d'un produit, on est renvoyé vers une page nous disant si le produit a bien été supprimé :



Chapitre 14 : Envoyer un fichier

Nous allons donc pouvoir envoyer des images, mais cela est source de faille de sécurité si non-traité.

On commence par créer l'input dans le Twig :

```
<div class="mb-3">
    <label for="productImage" class="form-label">Image</label>
    <input type="file" class="form-control" id="productImage" name="productImage">
</div>
```

Puis on ajoute de traitement de l'image et du titre dans le Controller :

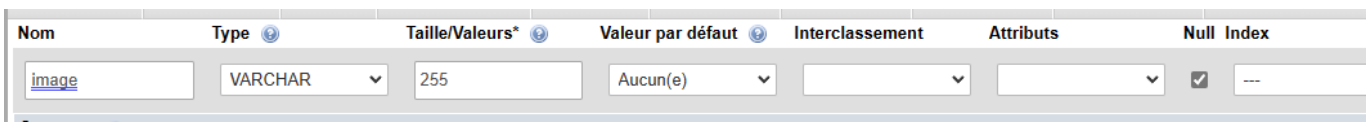
```
if (isset($_FILES["productImage"])) {
    if (!empty($_FILES["productImage"]['name'])) {
        $file_upload = explode(".", $_FILES["productImage"]['name']);
        // Vérification de l'extension
        if (in_array($file_upload[count($file_upload) - 1], $uploads['extensions'])) {
            // Nettoyage des accents
            $file_name = strtr(
                $file_upload[0],
                'ÀÁÂÃÄÅÇÈÉÊËÌÍÎÏÒÓÔÕÖÙÚÛÜÝàáâãäåçèéêëìíîïðòóôõöùúûüýÿ',
                'AAAAAACEEEEIIIIOOOOOUUUUYaaaaaaceeeeiiiioooooouuuuyy'
            );
            // Nettoyage des caractères spéciaux
            $file_name = preg_replace('/([^.a-z0-9]+)/i', '_', $file_name);
            $file_name = $file_name . '.' . $file_upload[count($file_upload) - 1];
            $file_path = $uploads['path'] . $file_name;
            if (!file_exists($file_path)) {
                // Déplacement du fichier
                move_uploaded_file($_FILES['productImage']['tmp_name'], $file_path);
                $uploads['state'] = true;
            } else {
                $msg = "Une image avec ce nom existe déjà !";
            }
        } else {
            $msg = "L'extension du fichier n'est pas acceptée !";
        }
    } else {
        $msg = "Veuillez choisir un fichier !";
    }
}

if (!empty($label) && !empty($description) && !empty($category)) {
    if ($uploads['state']) {
        $form = [
            'state' => 'success',
            'message' => 'Votre produit a bien été ajouté !'
        ];
    } else {
        $form = [
            'state' => 'danger',
            'message' => $msg
        ];
    }
```

On crée un fichier dans public/ qui servira à stocker les uploads, et on donne accès à ce dossier à tout le monde :

```
144  chmod -R 777 uploads/
145  ls -la
```

Il faudra plus tard passer les permissions de ce dossier en 755.

Il faut pouvoir stocker le chemin de l'image dans la base de données donc on ajoute une colonne « image » à la table des produits stockant une chaine de caractères :



Puis on modifie les fonctions Model afin de pouvoir sauvegarder les chemins d'images et de pouvoir les récupérer.

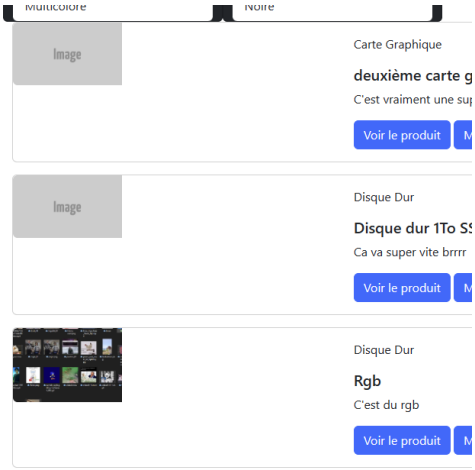
Tout fonctionne lorsque l'on ajoute un nouveau produit avec une image :

☐	Éditer	Copier	Supprimer	2	deuxième carte graphique	C'est vraiment une super carte graphique de oufff	600.5	1	*NULL*
☐	Éditer	Copier	Supprimer	3	Disque dur 1To SSD de la mort qui tue	Ca va super vite brrrr	100.5	2	*NULL*
☐	Éditer	Copier	Supprimer	11	Rgb	C'est du rgb	200	2	Capture_d_Aoy_cran_2023_02_08_204510.png

Tout cocher Avec la sélection : Éditer Copier Supprimer Exporter

On voit que les caractères ont également été retirés.

Enfin, il faut pouvoir afficher l'image du produit en question, pour cela on va ajouter l'affichage de l'image en fonction de la chaine de caractères stockée dans la base de données, et voilà le résultat :



Si dans la base de données, l'image est égale à null, alors on affiche une image par défaut.

Nous pouvons faire la même modification dans le Twig du showProduct :

```
</div>
<div class="row">
    <div class="col-md-4">
        <div class="card p-2">
            <div class="col-md-4">
                {% if product.image %}
                <img src="uploads/{{ product.image }}" class="img-fluid rounded-start" alt="...">
                {% else %}
                <img src="https://fakeimg.pl/350x200/?text=Image" class="img-fluid rounded-start"
                alt="...">
                {% endif %}
                </div>
        </div>
    </div>
    <div class="col-md-5 text-secondary">
        <p>{{ product.description }}</p>
```

Afin de faciliter la réutilisation du code, on crée un dossier « service » qui sert à y poser toutes les fonctions qui peuvent être réutilisées dans plusieurs endroits, et un fichier « Upload » avec dedans comme fonction :

```php
<?php
function VerifyImageName($image, $uploads)
{

    $file_name = null;

    $file_upload = explode(".", $image);
    // Vérification de l'extension
    if (in_array($file_upload[count($file_upload) - 1], $uploads['extensions'])) {
        // Nettoyage des accents
        $file_name = strtr(
            $file_upload[0],
            'ÀÁÂÃÄÅÇÈÉÊËÌÍÎÏÒÓÔÕÖÙÚÛÜÝàáâãäåçèéêëìíîïðòóôõöùúûüýÿ',
            'AAAAAACEEEEIIIIOOOOOUUUUYaaaaaaceeeeiiiioooooouuuuyy'
        );
        // Nettoyage des caractères spéciaux
        $file_name = preg_replace('/([^.a-z0-9]+)/i', '_', $file_name);
        $file_name = $file_name . '.' . $file_upload[count($file_upload) - 1];
        $file_path = $uploads['path'] . $file_name;
        if (!file_exists($file_path)) {
            // Déplacement du fichier
            move_uploaded_file($image['tmp_name'], $file_path);
            $uploads['state'] = true;
        } else {
            $msg['state'] = False;
            $msg['msg'] = "Une image avec ce nom existe déjà !";
            return $msg;
        }
    } else {
        $msg['state'] = False;
        $msg['msg'] = "L'extension du fichier n'est pas acceptée !";
        return $msg;
    }

    $res = [
        "state" => True,
        "file" => $file_name
    ];
    return $res;
}
```

Cette fonction sert à vérifier le nom d'une image afin et de renvoyer l'état de l'upload de l'image, et de l'upload complètement dans le server. Elle renvoie également en cas de réussite le chemin absolu de l'image afin de pouvoir conserver ce chemin dans la base de données.

```php
if (isset($_FILES["productImage"])) {
    if (!empty($_FILES["productImage"]['name'])) {

        $uploads = [
            'extensions' => ['png', 'jpg'],
            'path' => 'uploads/',
            'state' => false
        ];
        // Tu dois mettre la fonction demandée du coup mdrr
        $verify = VerifyImageName($_FILES["productImage"], $uploads);
        $file_name = '';
        $msg = '';
        if ($verify['state']) {
            $file_name = $verify['file'];
        } else $msg = $verify['msg'];

        $uploads['state'] = $verify['state'];
```

Mise en pratique, cette fonction permet de drastiquement réduire la taille du code et de faciliter la compréhension de ce qu'il se passe.

Chapitre 16 : Pages d'administration

Pour avoir des pages d'administrations, il faut commencer par créer ces pages en commençant par les routes :

```
'adminProducts' => 'adminProductsController',
'adminProductCategories' => 'adminProductCategoriesController',
```

Puis on fait les Controller :

```
function adminProductCategoriesController($twig, $db)
{
    include_once '../src/model/ProductModel.php';

    var_dump($_POST);

    echo $twig->render('adminProductCategories.html.twig', []);
}
```

Pour l'instant ils ne renvoient que la page Twig, sans faire de traitement de quoi que ce soit.

Et enfin les Twig avec simplement un titre pour débuter :

```
-E-Commerce > template > adminProductCategories.html.twig
{% extends "layout.html.twig" %}

{% block title %}Gestion des catégories de produit{% endblock %}

{% block content %}
<div class="container">
    <h1>Gestion des catégories de produit</h1>
</div>
{% endblock %}
```

Ensuite il faut récupérer les informations des produits et des catégories dans le Twig dans un tableau :

```
function adminProductsController($twig, $db)
{
    include_once '../src/model/ProductModel.php';
    include_once '../src/model/CategoryModel.php';

    var_dump($_POST);

    echo $twig->render('adminProducts.html.twig', [
        'products' => getAllProducts($db),
        'categories' => getAllCategories($db)
    ]);
}
```

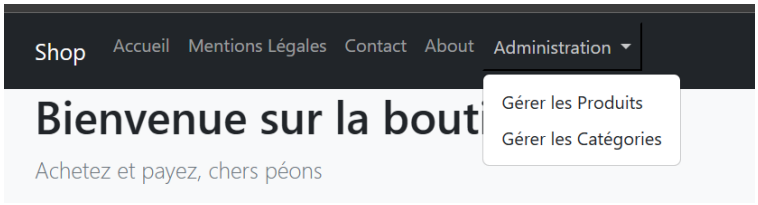
On récupère les informations des produits et des catégories afin d'avoir le nom des catégories liés auc catégories de produit.

On crée ensuite un tableau qui affiche tous les produits ligne par ligne, avec une condition qui récupère le libellé de la catégorie du produit :

```
<table class="table table-bordered table-striped">
  <thead>
    <tr>
      <th scope="col">ID</th>
      <th scope="col">Libellé</th>
      <th scope="col">Prix</th>
      <th scope="col">Catégorie</th>
      <th scope="col">Actions</th>
    </tr>
  </thead>
  <tbody>
    {% for product in products %}


    <tr>
      <th scope="row">{{product.id}}</th>
      <td>{{product.label}}</td>
      <td>{{product.price}}</td>

      {% for category in categories %}
      {% if product.idCategory == category.id %}
      <td>{{category.label}}</td>
      {% endif %}
      {% endfor %}

      <td><a href="?page=showProduct&product={{ product.id }}" class="btn btn-primary">Voir le produit</a>
        <a href="?page=updateProduct&product={{ product.id }}" class="btn btn-primary">Modifier le produit</a>
        <a href="?page=deleteProduct&product={{ product.id }}" class="btn btn-primary">Supprimer le produit</a>
      </td>
    </tr>
```

On se retrouve avec ce résultat :

En plus d'un dropdown menu pour accéder aux pages d'administration :



D'après vous, quelles sont les problématiques que vous pouvez observer à ce niveau du projet ?

On peut accéder aux modifications, ajout et suppression de produits et catégories à partir de n'importe quel appareil, tant qu'il est connecté à Internet et qu'il a l'url d'hébergement du site.

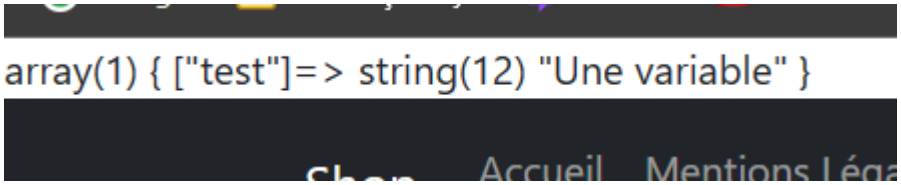
Chapitre 17 : Authentification

Nous allons commencer par la création d'un Controller pour le login afin de découvrir la variable SESSION :

```php
<?php

function loginController($twig, $db)
{
    include_once '../src/model/ProductModel.php';

    var_dump($_SESSION);

    echo $twig->render('login.html.twig', []);
}
```

Il s'agit simplement d'un Controller montrant le contenu de la variable SESSION

On a donc la variable SESSION qui est active et on peut voir que quand on met une variable dedans, elle reste.



Afin d'avoir des utilisateurs qui peuvent se connecter, et de les séparer entre administrateur et client, on a besoin de deux tables :

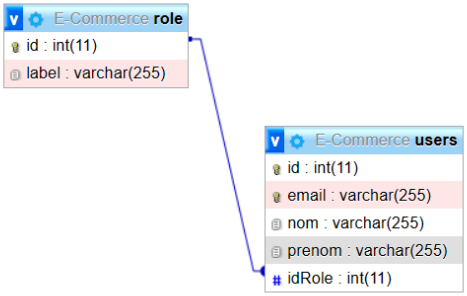
Rôle :

1	**id**	int(11)		Non	*Aucun(e)*	AUTO_INCREMENT
2	**label**	varchar(255)	utf8mb4_general_ci	Non	*Aucun(e)*	

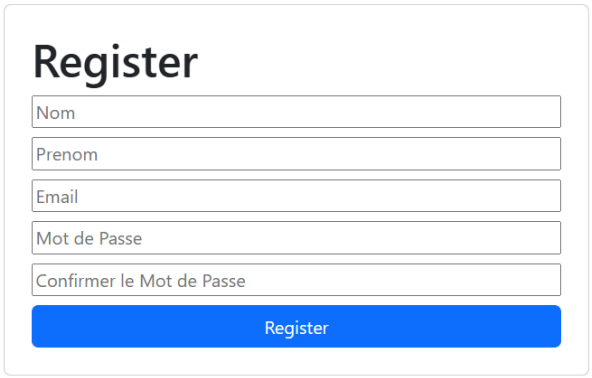
Et Utilisateurs :

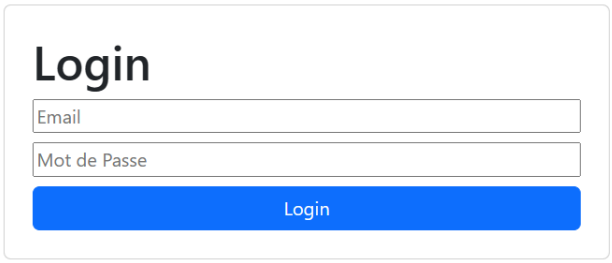
1	**id**	int(11)		Non	*Aucun(e)*	AUTO_INCREMENT
2	**email**	varchar(255)	utf8mb4_general_ci	Non	*Aucun(e)*	
3	**nom**	varchar(255)	utf8mb4_general_ci	Non	*Aucun(e)*	
4	**prenom**	varchar(255)	utf8mb4_general_ci	Non	*Aucun(e)*	
5	**idRole**	int(11)		Non	*Aucun(e)*	

On peut voir que le lien s'est bien fait :



On peut ensuite passer à la création des pages :





Ensuite on récupère les informations dans le Controller :

```php
if (isset($_POST['submitRegister'])) {
    if (
        isset($_POST['firstName'])
        && isset($_POST['lastName'])
        && isset($_POST['email'])
        && isset($_POST['password'])
        && isset($_POST['passwordVerif'])
    ) {
        $password = $_POST['password'];
        $passwordVerif = $_POST['passwordVerif'];

        if ($password == $passwordVerif) {
            $firstname = $_POST['firstName'];
            $lastname = $_POST['lastName'];
            $email = $_POST['email'];
        }
    }
}
```

En même temps, on vérifie si tous les champs sont bien remplis, et si les mots de passe concordent.

De même que dans la page Login :

```php
if (isset($_POST['submitLogin'])) {
    if (
        isset($_POST['email'])
        && isset($_POST['password'])
    ) {
        $password = $_POST['password'];
        $email = $_POST['email'];
    }
}
```

Afin de gérer la création de users, il nous faut la fonction saveUsers(), elle permet d'insérer des utilisateurs dans la base de données :

```php
function saveUser($db, $email, $nom, $prenom, $idrole)
{
    $query = $db->prepare("INSERT INTO users (email, nom, prenom, idRole)
                           VALUES (:email, :nom, :prenom, :idRole)");
    return $query->execute([
        'email' => $email,
        'nom' => $nom,
        'prenom' => $prenom,
        'idRole' => $idrole
    ]);
}
```

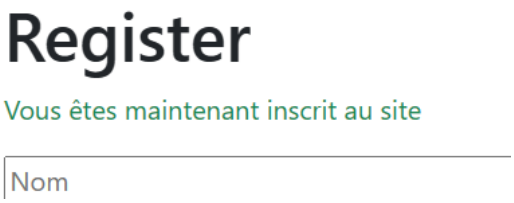
Nous avons également besoin de savoir si un utilisateur existe déjà :

```php
function getOneUserCredentials($db, $email)
{
    $query = $db->prepare("SELECT id, email, `password` FROM users WHERE email=:email");
    $query->execute([
        'email' => $email
    ]);

    $product = $query->fetch();

    return $product;
}
```

Cette fonction permet à la fois de savoir si un utilisateur existe, et s'il existe de récupérer les informations le concernant.

En aillant créé auparavant les rôles « Client » et « Administrateur », on réussit à créer un utilisateur grâce au formulaire :

id	email	nom	prenom	idRole	password
2	clement.parisot@epsi.fr	Parisot	Clément	1	$2y$10$.3mitX3LiPvkzfVANd/klumLFCQxJl/1NpqUMGG03px...



On peut également constater que le mot de passe est haché et est donc sécurisé.

Afin de vérifier si tous les champs sont bien remplis et que le mot de passe fait au moins 3 caractères, on peut ajouter toutes les conditions :

```php
if (isset($_POST['submitRegister'])) {
    if (
        isset($_POST['firstname'])
        && isset($_POST['lastname'])
        && isset($_POST['email'])
        && isset($_POST['password'])
        && isset($_POST['passwordVerif'])

        && $_POST['firstname'] != ""
        && $_POST['lastname'] != ""
        && $_POST['email'] != ""
        && $_POST['password'] != ""
    ) {
        if (strlen($_POST['password']) >= 3) {
            $email = $_POST['email'];
```

On vérifie si toutes les variables sont bien définies

On vérifie qu'il y a quelque chose d'inséré dans les variables

Puis on vérifie que le mot de passe fait au moins 3 caractères.

On fera de même pour la page login :

```php
if (isset($_POST['submitLogin'])) {
    if (
        isset($_POST['email']) &&
        isset($_POST['password']) &&
        $_POST['email'] != "" &&
        $_POST['password'] != ""
    ) {
        $email = $_POST['email'];
        $password = $_POST['password'];
        $user = getOneUserCredentials($db, $email);
```

On voit bien qu'on est connecté parce que l'on a les identifiants dans la variable SESSION :

```
array(1) { ["auth"]=> array(2) { ["login"]=> string(23) "clement.parisot@epsi.fr" ["role"]=> int(1) } }
```

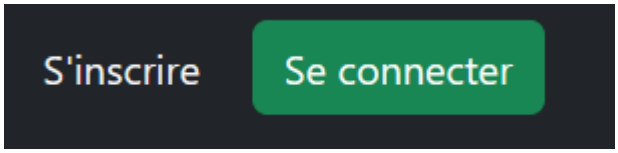
On va rendre l'acquisition de la variable SESSION dans le Twig plus simple en récupérant la variable SESSION directement en tant que variable globale :

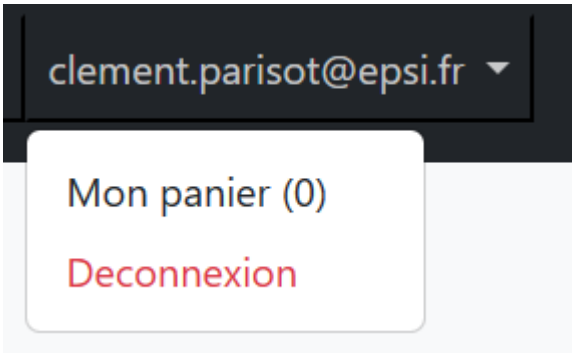
```php
function initTwig($path)
{
    $loader = new \Twig\Loader\FilesystemLoader($path);
    $twig = new \Twig\Environment($loader, []);
    $twig->addGlobal('session', $_SESSION);
    return $twig;
}
```

On peut donc récupérer le contenu de la variable SESSION dans Twig, et donc afficher soit des options de connexion, soit la possibilité de se déconnecter :

```
    {% if session.auth is defined %}
    <li class="nav-item">
        <div class="dropdown">
            <button class="dropdown-toggle nav-link bg-dark border-dark" type="button"
                data-bs-toggle="dropdown" aria-expanded="false">
                {{ session.auth.login }}
            </button>
            <ul class="dropdown-menu">
                <li><a class="dropdown-item" href="">Mon panier (0)</a></li>
                <li><a class="dropdown-item text-danger" href="">Deconnexion</a></li>
            </ul>
        </div>
    </li>
    {% else %}
    <li class="nav-item"><a href="?page=register" class="btn ms-2 text-light">S'inscrire</a></li>
    <li class="nav-item"><a href="?page=login" class="btn btn-success ms-2">Se connecter</a></li>
    {% endif %}
/ul>
```

Ce qui donne ceci :





Puis, pour que l'utilisateur ne puisse voir le menu d'administration que lorsqu'il est administrateur, on n'a qu'à rajouter une condition :

```
{% if session.auth is defined %}
{% if session.auth.role == 2 %}
<li class="nav-item">
    <div class="dropdown">
        <button class="dropdown-toggle nav-link bg-dark border-dark" type="button"
            data-bs-toggle="dropdown" aria-expanded="false">
            Administration
        </button>
        <ul class="dropdown-menu">
            <li><a class="dropdown-item" href="?page=adminProducts">Gérer les Produ
            <li><a class="dropdown-item" href="?page=adminProductCategories">Gérer
                Catégories</a></li>
        </ul>
    </div>
</li>
{% endif %}
```

Pour pouvoir se déconnecter, il faut un nouveau Controller, accessible par le bouton précédemment fait :

```php
function logoutController($twig, $db)
{
    $_SESSION = [];
    session_destroy();
    echo $twig->render('home.html.twig', []);
}
```

Une fois ce Controller utilisé, on est renvoyé vers la page Home, où on voit bien que l'on est déconnecté.

Nous allons passer à la réception d'email.

Nous allons commencer par installer PostFix (ici je l'avais déjà car travaillé en SLAM) :

```
root@HTTPS-8060:/var/www/html/Projet-E-Commerce# apt install postfix
Reading package lists... Done
Building dependency tree
Reading state information... Done
postfix is already the newest version (3.4.23-0+deb10u1).
0 upgraded, 0 newly installed, 0 to remove and 123 not upgraded.
root@HTTPS-8060:/var/www/html/Projet-E-Commerce#
```

Nous pouvons ensuite installer phpmailer :

```
login8060@HTTPS-8060:~/Projet-E-Commerce$ composer require phpmailer/phpmailer
Warning: This development build of Composer is over 60 days old. It is recommended
on.
Info from https://repo.packagist.org: #StandWithUkraine
Using version ^6.7 for phpmailer/phpmailer
./composer.json has been updated
Running composer update phpmailer/phpmailer
Loading composer repositories with package information
Info from https://repo.packagist.org: #StandWithUkraine
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
  - Locking phpmailer/phpmailer (v6.7.1)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
```

Nous allons par la suite créer un nouveau fichier dans template/mail :

Projet-E-Commerce > template > mail > egister_message.html.twig > p

```twig
<p>Bonjour {{ email }},</p>
<p>Merci d'avoir rejoint notre site.</p>
<p>
    Cordialement,<br />
    L'équipe Shop
</p>
```

Ce fichier servira de référence à l'envoi de mail au nouveaux-venus dans le site.

Ensuite, dans le RegisterController, on ajoute ce code au moment d'ajouter l'utilisateur dans la base de données :

```php
include_once '../config/confMail.php';

$mail = new PHPMailer();
$mail->isSMTP(); // Paramétrer le Mailer pour utiliser SMTP
$mail->Host = 'smtp.office365.com'; // Spécifier le serveur SMTP
$mail->SMTPAuth = true; // Activer authentication SMTP
$mail->Username = $conf['email'];
$mail->Password = $conf['mdp']; // Le mot de passe de cette adresse email
$mail->SMTPSecure = 'tls'; // Accepter SSL
$mail->Port = 587;
$mail->setFrom($conf['email'], 'Site E-Commerce'); // Personnaliser l'envoyeur
$mail->addAddress($email, $firstname . ' ' . $lastname);
$mail->isHTML(true); // Paramétrer le format des emails en HTML ou non
$mail->Subject = 'Inscription à Shop';
$mail->Body = $twig->render('mail/register_message.html.twig', [
    'email' => $email
]);

if (!$mail->send()) {
    $form['message'] = "Message non-envoyé !";
}
```

On crée un nouvel objet PHPMailer

On le configure avec notre adresse mail et le mot de passe

On configure l'envoyeur

On configure le contenu du message

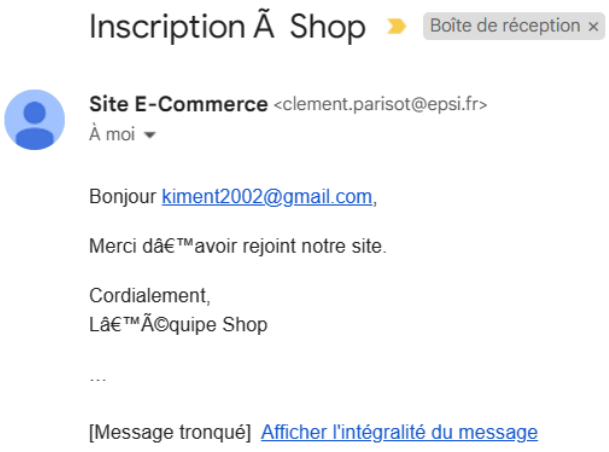
Et on envoie et vérifie s'il s'est bien envoyé.

Ici, j'ai configuré l'email d'envoi ainsi que le mot de passe dans un fichier autre, qui est spécifié dans le .gitignore afin de ne pas diffuser le mot de passe sur GitHub :

Projet-E-Commerce > config > confMail.php > ...

```php
<?php

$conf = [
    'email' => 'clement.parisot@epsi.fr',
    'mdp' => ''
];
```

Lors du test, on reçoit bien :



Chapitre 18 : Sécurisation des routes

Nous allons maintenant sécuriser toutes les routes afin que n'importe qui n'accède pas aux pages d'administration.

On va commencer par changer le code du router :

```php
$routeParameters = explode(':', $route);
$controller = ucfirst($routeParameters[0]);
$access = $routeParameters[1] ?? 0;
if ($access != 0) {
    if (!isset($_SESSION['auth']) || $_SESSION['auth']['role'] < $access) {
        $controller = "HomeController";
    }
}

require_once 'controller/' . $controller . '.php';
return $controller;
```

On va également transformer les routes pour spécifier à chaque fois le niveau d'accès demandé pour accéder à la page :

```php
$routes = [
    'home' => 'homeController:0',
    'about' => 'aboutController:0',
    'contact' => 'contactController:0',
    'mentionsLegales' => 'mentionsLegalesController:0',
    'notFound' => 'notFoundController:0',
    'dbError' => 'dbErrorController:0',

    'addProduct' => 'addProductController:2',
    'updateProduct' => 'updateProductController:2',
    'deleteProduct' => 'deleteProductController:2',
    'showProduct' => 'showProductController:1',
    'adminProducts' => 'adminProductsController:2',

    'adminProductCategories' => 'adminProductCategoriesController:2',
    'addCategory' => 'addCategoryController:2',

    'login' => 'loginController:0',
    'register' => 'registerController:0',
    'logout' => 'logoutController:0',
```

-1 : Utilisateur
-2 : Administrateur
-0 : Visiteur

Comme attendu, un utilisateur n'a pas accès à une page administrateur

Dernier problème, en étant connecté, on a accès aux pages de connexion et d'inscription. Pour remédier à cela, il suffit de faire cela :

```php
function loginController($twig, $db)
{
    include_once '../src/model/UserModel.php';

    $form = [];

    if (isset($_SESSION['auth'])) {
        echo $twig->render('home.html.twig', [
            'form' => $form
        ]);

        return 0;
    }

    if (isset($_POST['submitLogin'])) {
        if (
            isset($_POST['email']) &&
            isset($_POST['password']) &&
            $_POST['email'] != "" &&
```

Mettre une condition : si la variable SESSION a une connexion d'utilisateur, on renvoie la page home et on stop la fonction. On pourrait utiliser un header mais il empêche le débogage et cela n'est pas à faire dans la période de développement.

Nous ferons le même système dans la page d'inscription.

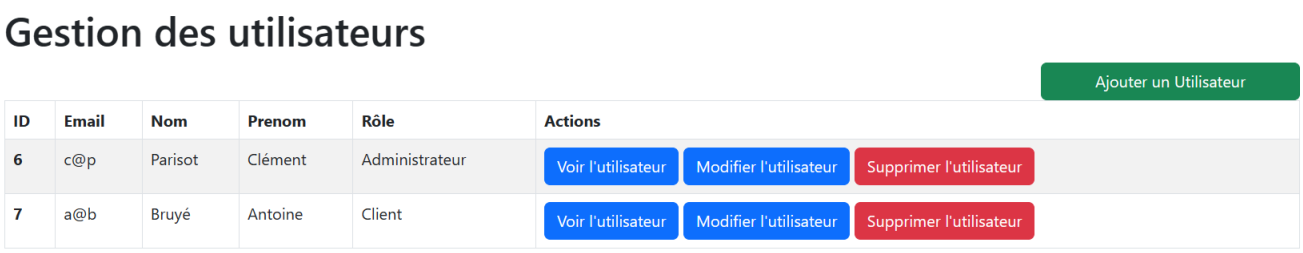
En tant qu'administrateur, on doit pouvoir ajouter, supprimer, consulter et modifier un utilisateur, il faut donc une page pour cela.

Pour pouvoir afficher tous les utilisateurs, il faut d'abord des fonctions permettant de faire cela :

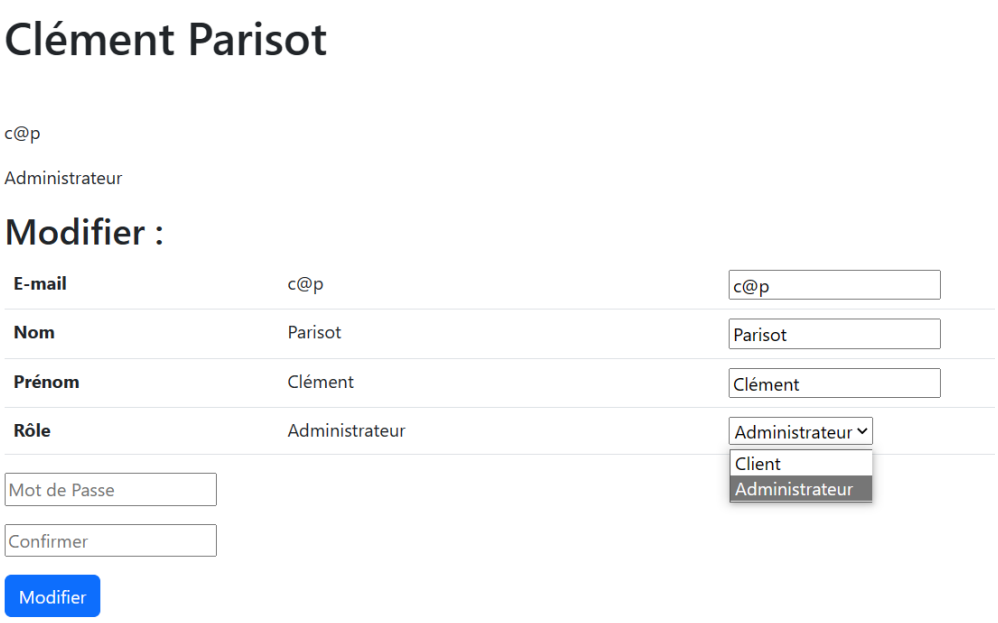
```php
function getAllUsers($db)
{
    $query = $db->prepare("SELECT id, nom, prenom, idRole, email FROM users");
    $query->execute([]);

    $users = $query->fetchAll();

    return $users;
}

function getAllRoles($db)
{
    $query = $db->prepare("SELECT id, label FROM role");
    $query->execute([]);

    $roles = $query->fetchAll();

    return $roles;
}
```

Une qui permet de récupérer tous les utilisateurs et une autre qui permet de récupérer tous les rôles.

En récupérant ce qui a été fait pour les produits, on se retrouve avec cela :



La page voir l'utilisateur et la page modifier seront la même :



```php
$user = null;

if (isset($_POST['modSubmit'])) {

    if ($_POST['modPass'] == $_POST['modPassVerif']) {
        $password = $_POST['modPass'];
        $user = getOneUserCredentials($db, $_SESSION['auth']['login']);

        if (password_verify($password, $user['password'])) {

            updateOneUser(
                $db,
                $_GET['user'],
                $_POST['modNom'],
                $_POST['modPrenom'],
                $_POST['modEmail'],
                $_POST['modRole']
            );
        }
    }
}

if (isset($_GET['user'])) {
    $user = getOneUser($db, $_GET['user']);
}

echo $twig->render('showUser.html.twig', [
    'user' => $user,
    'roles' => getAllRoles($db)
]);
```

Dans le Controller, on doit d'abord vérifier si le bouton a été pressé,

Puis si les mots de passe concordent,

Puis si le mot des passe administrateur est correct,

Puis on peut update l'utilisateur en fonction des données renseignées, pré-complétées pour ne pas avoir à re-renseigner des informations.

Le bouton de suppression d'utilisateur quant à lui fonctionne exactement comme la suppression de catégorie ou de produit.

L'ajout de personne n'est qu'une reprise à 90% du formulaire d'inscription :