

Projet Solution Web

Léo Bergeaud	Ethan Gobain	Noa Rodrigues	Vianney Haag	Arnaud Martin
-----------------	-----------------	------------------	-----------------	------------------



epsi

l'école d'ingénierie
informatique



wis

l'école [tech]
de l'expertise digitale

Problématique :

Comment optimiser le rendu des devoirs entre élèves et intervenant plus fluide ?

➤ Introduction

Ce projet vise à développer un service web permettant aux utilisateurs d'envoyer et de gérer des livrables sous la forme de fichiers ZIP et PowerPoint. Ce service repose sur les technologies HTML, CSS et PHP et se veut entièrement responsive, garantissant ainsi une compatibilité optimale avec tous types d'écrans et d'appareils.

L'application est composée de deux interfaces distinctes :

- Une interface utilisateur (index.php) permettant de naviguer dans l'arborescence des dossiers disponibles et d'envoyer des fichiers dans le dossier choisi.
- Une interface d'administration (admin.php) qui offre des fonctionnalités de gestion des dossiers (CRUD : création, lecture, mise à jour et suppression), avec des mécanismes de confirmation avant toute suppression définitive.

➤ Mise en place du projet

La mise en place du projet s'est déroulée en plusieurs étapes bien définies :

1. Définition des besoins

Avant de débiter le développement, nous avons établi un cahier des charges détaillant les fonctionnalités essentielles du service :

- Gestion des livrables par catégories
- Interface simple et intuitive pour les utilisateurs
- Sécurisation des fichiers envoyés (restriction aux formats ZIP et PPTX)
- Interface administrateur permettant une gestion efficace des dossiers

2. Conception de l'architecture

Nous avons conçu l'architecture du projet en organisant les fichiers et répertoires comme suit :

- /css/ : Contient les fichiers de styles CSS.
- /uploads/ : Stocke les fichiers envoyés par les utilisateurs.
- /admin/ : Contient les fichiers nécessaires à la gestion administrative.

- index.php : Interface principale utilisateur.
- admin.php : Interface d'administration.
- config.php : Fichier de configuration du projet.

3. Développement de l'interface utilisateur

L'interface utilisateur est basée sur une page HTML5 intégrant un formulaire permettant l'affichage de l'arborescence des dossiers et la sélection d'un dossier de destination avant l'envoi du livrable. Un bouton est placé à côté de chaque dossier pour valider l'envoi.

4. Développement de l'administration

L'administration est conçue pour permettre une gestion complète des dossiers :

- Création de nouveaux dossiers en renseignant une section et une date limite de livraison.
- Visualisation de tous les dossiers et fichiers sous forme d'arborescence.
- Modification de la date limite d'un dossier existant.
- Suppression sécurisée d'un dossier avec double confirmation.

5. Mise en place du responsive design

L'application est conçue pour être totalement responsive grâce à l'utilisation de CSS et de la bibliothèque Bootstrap. L'affichage s'adapte automatiquement aux écrans de différentes tailles, garantissant une navigation fluide sur ordinateur, tablette et mobile.

6. Tests et validation

Des tests fonctionnels ont été réalisés pour s'assurer du bon fonctionnement des formulaires, de l'affichage des dossiers et des fonctionnalités d'administration. Nous avons également testé la sécurité de l'application, notamment pour empêcher l'envoi de fichiers non autorisés.

➤ Liste des outils utilisés

Nous avons utilisé plusieurs outils pour mener à bien ce projet :

1. Langages de programmation

- HTML5 : Structure des pages web.
- CSS3 : Mise en forme et design des interfaces.
- PHP : Traitement des requêtes et interactions avec le serveur.

2. Frameworks et bibliothèques

Bootstrap : Assure la compatibilité et le design responsive de l'interface.

3. Environnement de développement

- VS Code / PhpStorm : Éditeurs de code utilisés.
- XAMPP / WAMP : Serveur local pour tester le projet en environnement réel.

4. Gestion du code source

Git / GitHub : Versionnement et suivi des modifications du projet.

5. Hébergement

- Serveur local (XAMPP, WAMP) ou serveur distant : Pour le déploiement du projet.

Fonctionnalités principales :

Côté utilisateur (index.php) :

- Affichage de l'arborescence des dossiers : L'utilisateur peut naviguer dans les dossiers disponibles.
- Choix du dossier de destination : Sélectionner un dossier pour y envoyer un fichier.

- Envoi des fichiers : Seuls les fichiers ZIP et PPTX sont acceptés.
- Interface responsive : Adaptation de l’affichage selon l’appareil utilisé.

Côté administrateur (admin.php) :

- CREATE : Création de nouveaux dossiers avec section et date limite.
- READ : Affichage de l’arborescence des dossiers et fichiers stockés.
- UPDATE : Modification de la date limite d’un dossier existant.
- DELETE : Suppression sécurisée d’un dossier avec double confirmation.

Le projet a été développé en respectant la charte graphique EPSI - WIS avec l’intégration des couleurs et logos requis.

➤ Ensemble du Travail - Explication

Pour l’authentification, nous avons d’abord créé une Entity que nous avons appelé User.php. Dans cette Entity, nous avons créé la classe User puis défini les différentes variables contenues dans cette table comme ceci :

```
<?php
declare (strict_types = 1);
namespace MyApp\Entity;

class User
{
    private ?int $id = null;
    private string $nom;
    private string $prenom;
    private string $email;
    private string $password;
    private array $roles;
    public function __construct(?int $id, string $nom, string $prenom, string $email, string $password, array $roles)
    {
        $this->id = $id;
        $this->nom = $nom;
        $this->prenom = $prenom;
        $this->roles = $roles;
        $this->password = $password;
        $this->email = $email;
    }
}
```

Nous avons donc créé une déclaration de propriété typée au sein de la classe pour chaque variable, comme l’id ou le nom par exemple.

Nous avons ensuite créé une fonction publique, donc accessible en dehors de cette classe, appelée `_construct` car c'est un nom réservé pour les constructeurs en PHP. Et dans la parenthèse se trouve les paramètres de la fonction.

Ensuite, nous avons créé plusieurs fonctions qui serviront à insérer ou récupérer les différentes variables de la table User, prenons pour exemple l'id

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

public function setId(?int $id): void
{
    $this->id = $id;
}
```

:

Nous commençons d'abord par créer et appeler la fonction `getId`, qui est un getter qui nous servira à récupérer l'id qui est de type integer, cette fonction permet donc de renvoyer l'id comme l'indique l'instruction `return`. Le `public` est un modificateur de visibilité qui signifie que la méthode est accessible de partout, même en dehors de la classe. Et le `return $this->id` permet de retourner la valeur actuelle de l'id de l'élément voulu de la classe.

Nous créons ensuite la fonction `setId`, qui est un setter, et qui elle permet de créer et définir l'id. Ici, `$this` fait référence à l'instance actuelle de la classe. `$this->id` accède à la propriété `id` de cette table et on assigne la valeur passée en paramètre à cet id. Et nous refaisons la même chose pour les différentes variables de la table User comme nom ou prénom, à l'exception

de l'email et password, pour des raisons de sécurité.

```
public function getEmail(): string
{
    return $this->email;
}

public function setEmail(string $email): void
{
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        throw new InvalidArgumentException("Email invalide.");
    }
    $this->email = $email;
}

public function getPassword(): string
{
    return $this->password;
}

public function verifyPassword(string $password): bool
{
    return password_verify($password, $this->password);
}
```

Pour l'email, le getter ne change pas mais le setter est différent, on rajoute une ligne pour vérifier la validité de l'email avec `FILTER_VALIDATE_EMAIL` qui permet de vérifier si l'email est conforme, par exemple qu'il ne manque pas de @.

Pour le password, comme l'email, le getter ne change pas, par contre il n'y a pas de getter mais on vérifie si le password correspond au password haché qu'on trouvera dans la base de données.

Après avoir créé l'Entity User, nous avons créé le model UserModel.php, c'est dans ce dossier que nous ferons les différentes fonctions qui nous permettront de s'inscrire ou se connecter par exemple.

```

1  <?php
2  declare (strict_types = 1);
3  namespace MyApp\Model;
4
5  use MyApp\Entity\User;
6  use PDO;
7
8  class UserModel
9  {
10     private PDO $db;
11     public function __construct(PDO $db)
12     {
13         $this->db = $db;
14     }

```

Nous commençons en appelant l'Entity User qui nous avons précédemment créé et en indiquant que nous allons importer la classe PDO dans notre fichier.

Nous créons ensuite la classe UserModel et nous déclarons la propriété private PDO \$db

private : Rend la propriété \$db accessible uniquement au sein de la classe.

PDO : Type hint indiquant que \$db doit être une instance de la classe PDO.

\$db est la connexion à la base de données.

Nous créons ensuite la fonction constructeur _construct :

Le constructeur accepte un argument de type PDO, ce qui signifie que l'objet doit être une connexion valide à la base de données.

Cela permet de découpler la création de l'objet PDO de la classe, facilitant les tests ou le changement de connexion.

\$this->db = \$db associe l'instance de PDO fournit au constructeur à la propriété \$db de l'objet actuel.

Nous avons ensuite créé les différentes fonctions qui vont nous servir pour l'inscription et la connexion comme la fonction qui permet de créer des

utilisateurs :

```
public function createUser(User $user): bool
{
    $sql = "INSERT INTO User (nom, prenom, email, password, roles) VALUES
    (:nom, :prenom, :email, :password, :roles)";
    $stmt = $this->db->prepare($sql);
    $stmt->bindValue(':nom', $user->getNom());
    $stmt->bindValue(':prenom', $user->getPrenom());
    $stmt->bindValue(':email', $user->getEmail());
    $stmt->bindValue(':roles', json_encode($user->getRoles()));
    $stmt->bindValue(':password', $user->getPassword());
    return $stmt->execute();
}
```

Nous avons donc utilisé la requête SQL INSERT INTO afin d'insérer les différentes valeurs dans la table User tels que le nom ou le prenom.

En dessous les lignes signifient :

\$stmt = \$this->db->prepare(\$sql); (Préparation de la requête)

\$this->db est une instance de PDO (connexion à la base de données).

prepare() est une méthode qui :

- prend une requête SQL sous forme de chaîne (\$sql).
- retourne un objet de type PDOStatement que vous pouvez utiliser

pour lier des valeurs et exécuter la requête.

\$stmt->bindValue(':nom', \$user->getNom()); (Liaison de valeur)

bindValue() est une méthode qui associe une valeur spécifique à un paramètre nommé (:nom) ou positionnel (?).

:nom est un marqueur nommé qui sera remplacé par la valeur fournie.

\$user->getNom() permet de récupérer le nom

Et toutes les lignes en dessous signifient la même chose selon les différentes variables de la table.

Les prochaines fonctions permettent de récupérer l'utilisateur selon son email ou son id :

```
public function getUserByEmail(string $email): ?User
{
    $sql = "SELECT * FROM User WHERE email = :email";
    $stmt = $this->db->prepare($sql);
    $stmt->bindValue(':email', $email);
    $stmt->execute();
    $row = $stmt->fetch(PDO::FETCH_ASSOC);
    if (!$row) {
        return null;
    }
    return new User($row['id'], $row['nom'], $row['prenom'],
    $row['email'], $row['password'], json_decode($row['roles']));
}

public function getUserById(int $id): ?User
{
    $sql = "SELECT * FROM User WHERE id = :id";
    $stmt = $this->db->prepare($sql);
    $stmt->bindValue(':id', $id);
    $stmt->execute();
    $row = $stmt->fetch(PDO::FETCH_ASSOC);
    if (!$row) {
        return null;
    }
    return new User($row['id'], $row['nom'], $row['prenom'],
    $row['email'], $row['password'], json_decode($row['roles']));
}
```

Nous commençons donc par sélectionner l'ensemble des éléments dans la table User avec la requête SQL `SELECT * FROM User`, et selon la fonction, on les sélectionne avec leur email ou leur id.

Les lignes en dessous sont similaires à celles utilisées au-dessus.

Pour finir le return renvoie un nouvel utilisateur avec les données se trouvant dans la parenthèse.

Les dernières fonction permettent de modifier les informations d'un utilisateur, de le supprimer et de renvoyer tous les utilisateurs déjà présents

dans la table User :

```
public function updateUser(User $user): bool
{
    $sql = "UPDATE User SET nom = :nom, prenom = :prenom,
    email = :email, password = :password WHERE id = :id";
    $stmt = $this->db->prepare($sql);
    $stmt->bindValue(':nom', $user->getNom());
    $stmt->bindValue(':prenom', $user->getPrenom());
    $stmt->bindValue(':email', $user->getEmail());
    $stmt->bindValue(':password', $user->getPassword());
    $stmt->bindValue(':id', $user->getId());
    return $stmt->execute();
}

public function getAllUsers(): array
{
    $sql = "SELECT * FROM User";
    $stmt = $this->db->query($sql);
    $users = [];
    while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
        $users[] = new User($row['id'], $row['nom'], $row['prenom'], $row['email'], $row['password'], json_decode($row['roles']));
    }
    return $users;
}

public function deleteUser(int $id): bool
{
    $sql = "DELETE FROM User WHERE id = :id";
    $stmt = $this->db->prepare($sql);
    $stmt->bindValue(':id', $id);
    return $stmt->execute();
}
```

La première fonction permet de modifier un utilisateur grâce à la requête SQL UPDATE selon l'id de l'utilisateur.

La deuxième fonction permet de renvoyer tous les utilisateurs de la table User contrairement aux deux fonctions précédentes qui renvoient l'utilisateur selon son id ou son email.

Puis la dernière fonction du model permet de supprimer un utilisateur selon l'id de l'utilisateur, avec la requête SQL DELETE.

Par la suite, il a fallu mettre dans un dossier DefaultController les différentes fonctions qui nous serviront à nous inscrire.

```
<?php
declare (strict_types = 1);
namespace MyApp\Controller;

use MyApp\Entity\User;
use MyApp\Model\UserModel;
use MyApp\Entity\Fichier;
use MyApp\Model\FichierModel;
use MyApp\Entity\Upload;
use MyApp\Service\DependencyContainer;
use Twig\Environment;

class DefaultController
{
    private $twig;
    private $userModel;
    private $fichierModel;

    public function __construct(Environment $twig, DependencyContainer $dependencyContainer)
    {
        $this->twig = $twig;
        $this->userModel = $dependencyContainer->get('UserModel');
        $this->fichierModel = $dependencyContainer->get('FichierModel');
    }
}
```

Pour commencer, il a fallu annoncer l'Entity User et le model UserModel avec les deux lignes use MyApp\Entity\User et use MyApp\Model\UserModel.

Ensuite, pour le côté admin, il faut qu'il puisse gérer les utilisateurs et les fichiers :

```
public function deleteUser()
{
    $id = filter_input(INPUT_GET, 'id', FILTER_SANITIZE_NUMBER_INT);
    $this->userModel->deleteUser(intVal($id));
    header('Location: index.php?page=users');
}
```

Cette fonction sert, comme son nom l'indique, à supprimer un utilisateur, pour cela, on récupère l'id de l'utilisateur puis on appelle la fonction deleteUser se trouvant dans le model UserModel et on renvoie vers la page présentant les utilisateurs.

```

public function updateUser()
{
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $id = filter_input(INPUT_POST, 'id', FILTER_SANITIZE_NUMBER_INT);
        $nom = filter_input(INPUT_POST, 'nom', FILTER_SANITIZE_STRING);
        $prenom = filter_input(INPUT_POST, 'prenom', FILTER_SANITIZE_STRING);
        $email = filter_input(INPUT_POST, 'email', FILTER_SANITIZE_STRING);
        $password = filter_input(INPUT_POST, 'password', FILTER_SANITIZE_STRING);
        if (!empty($id) && !empty($nom) && !empty($prenom) && !empty($email)
            && !empty($password)) {
            $user = new User(intVal($id), $nom, $prenom, $email, $password);
            $success = $this->userModel->updateUser($user);
            if ($success) {
                header('Location: index.php?page=users');
            }
        }
    } else {
        $id = filter_input(INPUT_GET, 'id', FILTER_SANITIZE_NUMBER_INT);
    }
    $user = $this->userModel->getOneUser(intVal($id));
    echo $this->twig->render('defaultController/updateUser.html.twig', ['user' => $user]);
}

```

Cette fonction permet de modifier un utilisateur :

Les premières lignes permettent de récupérer des données envoyées par un formulaire HTML via la méthode POST, les nettoie et les valide avant de les utiliser.

Plus précisément :

Il prend les données saisies par l'utilisateur dans un formulaire, spécifiquement les champs nom, prenom, email et password.

Ensuite on vérifie que les variables présentées précédemment ne soient pas vides.

Si c'est le cas, on crée un nouvel objet User avec toutes les variables.

Ensuite on appelle une méthode updateUser() de l'objet \$this->userModel pour mettre à jour un utilisateur dans la base de données. La valeur retournée par cette méthode est stockée dans la variable \$success.

Si la variable success est validée, alors on est renvoyé sur la page des utilisateurs.

Mais si les variables sont vides, alors on récupère un id envoyé en paramètre d'URL (via la méthode GET) et le nettoie pour s'assurer qu'il s'agit d'un entier.

Puis après la condition if de base, on récupère un utilisateur spécifique dans la base de données grâce à son identifiant (\$id) et l'affiche ensuite en utilisant un template Twig (updateUser.html.twig).

```

public function addUser()
{
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $nom = filter_input(INPUT_POST, 'nom', FILTER_SANITIZE_STRING);
        $prenom = filter_input(INPUT_POST, 'prenom', FILTER_SANITIZE_STRING);
        $email = filter_input(INPUT_POST, 'email', FILTER_SANITIZE_STRING);
        $password = filter_input(INPUT_POST, 'password', FILTER_SANITIZE_STRING);
        $roles = filter_input(INPUT_POST, 'roles', FILTER_SANITIZE_STRING);
        if (!empty($_POST['nom']) && !empty($_POST['prenom']) && !empty($_POST['email']) && !empty($_POST['password']) && !empty($_POST['roles'])) {
            $user = new User(null, $nom, $prenom, $email, $password, $roles);
            $success = $this->userManager->createUser($user);
            if ($success) {
                header('Location: index.php?page=addUser');
            }
        }
    }
    echo $this->twig->render('defaultController/addUser.html.twig', []);
}

```

Ensuite, cette fonction permet d'ajouter un utilisateur.

Cette fonction est similaire à la fonction updateUser, avec comme différence la ligne \$success qui cette fois utilise la fonction createUser du model User.

Et on fait exactement la même chose pour les fichiers, en changeant juste les variables.

Puis, pour que seul les admins puissent accéder à ces fonctionnalités, il faut rajouter des données dans le router :

```

'deconnexion' => [DefaultController::class, 'deconnexion', []],
'updateUser' => [DefaultController::class, 'updateUser', ['admin']],

```

On peut le voir ici, si le mot admin est écrit, alors la page ne sera accessible que pour les admins.

```

[$controllerClass, $method, $requiredRoles] = $controllerInfo;

```

Il faut ensuite modifier cette ligne en ajoutant \$requiredRoles pour limiter l'accès aux admins uniquement.

```

        if ($this->checkUserPermissions($requiredRoles)) {
            // Vérification de l'existence de la classe et de la méthode du contrôleur à appeler
            if (class_exists($controllerClass) && method_exists($controllerClass, $method)) {
                // Instancie la classe récupérée
                $controller = new $controllerClass($twig, $this->dependencyContainer);
                // la fonction call_user_func appelle une méthode sur un objet
                call_user_func([$controller, $method]);
            } else {
                // Si la classe ou la méthode n'existe pas, utilisez le contrôleur d'erreur 500
                $this->handleError($twig, '500');
            }
        } else {
            $this->handleError($twig, '403');
        }
    }

    private function checkUserPermissions(array $requiredRoles): bool
    {
        if (!empty($requiredRoles)) {
            if (isset($_SESSION['roles'])) {
                $i = array_intersect($_SESSION['roles'], $requiredRoles);
                if (empty($i)) {
                    return false;
                } else {
                    return true;
                }
            } else {
                return false;
            }
        } else {
            return true;
        }
    }
}

```

Il faut également rajouter la fonction `checkUserPermissions` qui permet de vérifier si l'utilisateur a l'un des rôles requis dans sa session.

La fonction prend en paramètre un tableau `$requiredRoles`, qui contient les rôles requis pour l'utilisateur.

On commence par vérifier si le tableau `$requiredRoles` n'est pas vide.

Ensuite, on vérifie si la variable de session `$_SESSION['roles']` existe, qui est supposée être un tableau de rôles assignés à l'utilisateur.

On utilise `array_intersect()` pour vérifier s'il existe des rôles communs entre `$_SESSION['roles']` et `$requiredRoles`. Si aucun rôle commun n'est trouvé (`empty($i)`), elle renvoie `false`, ce qui signifie que l'utilisateur n'a pas les rôles nécessaires, sinon on renvoie `true` ce qui signifie que l'utilisateur possède les rôles nécessaires.

Si `$_SESSION['roles']` n'existe pas, la fonction retourne false, indiquant que les rôles ne sont pas définis pour l'utilisateur.

Et si l'utilisateur ne possède pas les permissions requises, alors si il essaye d'accéder à une page réservée aux admins, on renvoie une erreur 403.

```
public function inscription()
{
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $nom = filter_input(INPUT_POST, 'nom', FILTER_SANITIZE_STRING);
        $prenom = filter_input(INPUT_POST, 'prenom', FILTER_SANITIZE_STRING);
        $email = filter_input(INPUT_POST, 'email', FILTER_VALIDATE_EMAIL);
        $password = $_POST['password'];

        $passwordLength = strlen($password);
        $containsDigit = preg_match('/\d/', $password);
        $containsUpper = preg_match('/[A-Z]/', $password);
        $containsLower = preg_match('/[a-z]/', $password);
        $containsSpecial = preg_match('/[^a-zA-Z\d]/', $password);

        if (!$nom || !$prenom || !$email || !$password) {
            $_SESSION['message'] = 'Erreur : données invalides';
        } elseif ($passwordLength < 12 || !$containsDigit || !$containsUpper || !$containsLower || !$containsSpecial) {
            $_SESSION['message'] = 'Erreur : mot de passe non conforme';
        } else {
            // Hachage du mot de passe
            $hashedPassword = password_hash($password, PASSWORD_DEFAULT);
            $user = new User(null, $nom, $prenom, $email, $hashedPassword, ['user']);
            // Enregistrez les données de l'utilisateur dans la base de données
            $result = $this->userModel->createUser($user);
            if ($result) {
                $_SESSION['message'] = 'Votre inscription est terminée';
                header('Location: index.php?page=connexion');
                exit;
            } else {
                $_SESSION['message'] = 'Erreur lors de l\'inscription';
            }
        }
        header('Location: index.php?page=inscription');
        exit;
    }

    echo $this->twig->render('defaultController/inscription.html.twig', []);
}
```

Voici la fonction inscription, qui permet aux utilisateurs de s'inscrire et leurs données se retrouvent directement dans la base de données.

Les premières lignes, comme pour les fonctions précédentes, permettent de récupérer des données envoyées par un formulaire HTML via la méthode POST, les nettoie et les valide avant de les utiliser.

Plus précisément :

Il prend les données saisies par l'utilisateur dans un formulaire, spécifiquement les champs nom, prenom, dateNaissance et email.

Ensuite, il applique des filtres pour nettoyer ces données. Le nettoyage consiste à supprimer les caractères indésirables comme les balises HTML ou

autres éléments qui pourraient être potentiellement dangereux ou inappropriés.

La ligne du `$password` permet de récupérer la valeur du champ password envoyé via la méthode POST d'un formulaire HTML.

Ensuite, les 5 lignes suivantes permettent plusieurs vérifications de sécurité sur un mot de passe afin de s'assurer qu'il respecte certaines règles de complexité :

La ligne `$passwordLength = strlen($password);` calcule la longueur du mot de passe et l'assigne à la variable

La ligne `$containsDigit = preg_match('/\d/', $password);` permet de savoir si le mot de passe contient au moins un chiffre

La ligne `$containsUpper = preg_match('/[A-Z]/', $password)` permet de savoir si le mot de passe contient au moins une majuscule*

La ligne `$containsLower = preg_match('/[a-z]/', $password);` permet de savoir si le mot de passe contient des minuscules

Et enfin la ligne `$containsSpecial = preg_match('/[^a-zA-Z\d]/', $password);` permet de savoir si le mot de passe contient au moins un caractère spécial

```
if (!$nom || !$prenom || !$email || !$password ) {
```

```
    $_SESSION['message'] = 'Erreur : données invalides';
```

Cette ligne permet de renvoyer le message 'Erreur : données invalides' si les variables nom, prenom, dateNaissance, email et password ne correspondent pas aux filtres présentés plus tôt.

Après, si jamais ce n'est pas le cas, alors le code passe à la ligne suivante

```
elseif ($passwordLength < 12 || !$containsDigit || !$containsUpper ||  
!$containsLower || !$containsSpecial) {
```

```
$_SESSION['message'] = 'Erreur : mot de passe non conforme';
```

qui nous indique que si le mot de passe contient moins de 12 caractères et ne contient aucunes de conditions présentées au dessus, alors le message 'mot de passe non conforme' s'affiche.

```
$hashedPassword = password_hash($password, PASSWORD_DEFAULT);
```

Cette ligne permet de hacher le mot de passe afin de renforcer la sécurité pour que le mot de passe de l'utilisateur ne puisse pas être visible dans la base de données directement.

```
$user = new User(null, $nom, $prenom, $email, $hashedPassword, ['user']);
```

Cette ligne nous permet donc de créer un nouvel utilisateur si les conditions précédentes ont été remplies.

```
$result = $this->userModel->createUser($user);
```

```
if ($result) {
```

```
    $_SESSION['message'] = 'Votre inscription est terminée';
```

```
    header('Location: index.php?page=connexion');
```

Ces lignes nous indiquent que si le result, donc ici le fait de créer un nouvel utilisateur, est vérifié alors le message 'Votre inscription est terminée' s'affiche et nous sommes redirigé vers la page connexion.

```
else {
```

```
    $_SESSION['message'] = 'Erreur lors de l\'inscription';
```

Et si les conditions ne sont pas vérifiées alors le message ci-dessus s'affiche et l'utilisateur reste sur la page d'inscription.

Maintenant après l'inscription vient la connexion :

```
public function connexion()
{
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $email = filter_input(INPUT_POST, 'email', FILTER_VALIDATE_EMAIL);
        $password = $_POST['password'];
        $user = $this->userModel->getUserByEmail($email);
        if (!$user) {
            $_SESSION['message'] = 'Utilisateur ou mot de passe erroné';
            header('Location: index.php?page=connexion');
        } else {
            if ($user->verifyPassword($password)) {
                $_SESSION['connexion'] = $user->getEmail();
                $_SESSION['roles'] = $user->getRoles();
                header('Location: index.php');
                exit;
            } else {
                $_SESSION['message'] = 'Utilisateur ou mot de passe erroné';
                header('Location: index.php?page=connexion');
                exit;
            }
        }
    }
    echo $this->twig->render('defaultController/connexion.html.twig', []);
}
```

Les deuxième et troisième lignes sont exactement les mêmes que celles de l'inscription.

Ensuite si la condition user n'est pas remplie, alors le message 'Utilisateur ou mot de passe erroné' et fait rester l'utilisateur sur la page de connexion.

Sinon, le mot de passe de l'utilisateur est vérifié pour voir s'il remplit les conditions établies plus tôt.

Si c'est le cas, alors l'utilisateur est retrouvé dans la base de données grâce à son email et son rôle et il est ensuite redirigé vers la page d'accueil.

Et si le mot de passe n'est pas vérifié, alors le message 'Utilisateur ou mot de passe erroné' et il reste sur la page de connexion.

```

<div class="collapse navbar-collapse" id="navbarColor02">
  <ul class="navbar-nav me-auto">
    <li class="nav-item">
      <a class="nav-link" href="index.php?page=home">Accueil</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="index.php?page=mentionsLegales">Mentions légales</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="index.php?page=users">Listes des utilisateurs</a>
    </li>
    {%if session.connexion is defined %}
    {%if 'admin' in session.roles %}
      <li class="nav-item dropdown">
        <a class="nav-link dropdown-
toggle" data-bs-toggle="dropdown" href="#" role="button" aria-haspopup="true" aria-expanded="false">Utilisateur</a>
        <div class="dropdown-menu">
          <a class="dropdown-item" href="index.php?page=updateUser">Modifier des utilisateurs</a>
          <a class="dropdown-item" href="index.php?page=addUser">Ajouter des utilisateurs</a>
          <a class="dropdown-item" href="index.php?page=deleteUser">Supprimer des utilisateurs</a>
        </div>
      </li>
      <li class="nav-item dropdown">
        <a class="nav-link dropdown-
toggle" data-bs-toggle="dropdown" href="#" role="button" aria-haspopup="true" aria-expanded="false">Fichiers</a>
        <div class="dropdown-menu">
          <a class="dropdown-item" href="index.php?page=updateFichier">Modifier des fichiers</a>
          <a class="dropdown-item" href="index.php?page=addFichier">Ajouter des fichiers</a>
          <a class="dropdown-item" href="index.php?page=deleteFichier">Supprimer des fichiers</a>
        </div>
      </li>
    {% endif %}
  {% endif %}

```

Nous pouvons voir ici le fichier base.html.twig, qui servira pour la création de tous les fichiers twig.

Dans ce fichier, nous pouvons retrouver la navbar, la barre de navigation, qui permettra d'accéder aux différentes pages sur le site.

```
{%if session.connexion is defined %}
```

A partir de cette ligne, on retrouve les conditions de sécurité selon le rôle, si le rôle de l'utilisateur est admin, alors il pourra gérer les utilisateurs ou les fichiers.

Alors que si le rôle de l'utilisateur est client, alors il pourra juste accéder aux circuits et se déconnecter.

```
{% extends "base.html.twig" %}
{% block title %}
    {{ parent() }}
    Se connecter
{% endblock %}
{% block content %}
    <div class="container-fluid">
        <div class="row justify-content-center">
            <div class="col-12 col-md-10 text-primary">
                <h1 class="text-center text-black">Se connecter</h1>
                <form method="post" action="">
                    <div class="mb-3">
                        <label for="userEmail" class="form-label">Email</label>
                        <input type="email" class="form-control" id="userEmail" name="email" required>
                    </div>
                    <div class="mb-3">
                        <label for="userPassword" class="form-label">Mot de Passe</label>
                        <input type="password" class="form-control" id="userPassword" name="password" required>
                    </div>
                    <div class="text-center">
                        <button type="submit" class="btn btn-primary">Se connecter</button>
                    </div>
                </form>
            </div>
        </div>
    </div>
{% endblock %}
```

Ce fichier Twig génère une page de connexion en héritant de base.html.twig. Il définit un titre personnalisé ("Se connecter") et utilise Bootstrap pour la mise en page.

Structure principale :

1. Héritage du template principal :
 - Il réutilise base.html.twig pour garder une structure cohérente avec le reste du site.
2. Définition du titre :
 - Ajoute "Se connecter" au titre de la page tout en conservant le titre défini dans le fichier parent.
3. Mise en page avec Bootstrap :
 - Utilise une container-fluid et une row justify-content-center pour centrer le formulaire.

- Le formulaire est contenu dans une col-12 col-md-10 text-primary, qui ajuste sa largeur selon l'écran.
-
- 4. Formulaire de connexion :
 - Champ Email :
 - `<input type="email" name="email" required>` avec un label associé.
 - Champ Mot de passe :
 - `<input type="password" name="password" required>`, également avec un label.
 - Bouton de soumission :
 - `<button type="submit">Se connecter</button>` avec un style Bootstrap (btn btn-primary).

Ce code permet d'afficher la page de connexion

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

{% block title %}
    {{ parent() }}
    Inscription
{% endblock %}
{% block content %}
    <div class="container-fluid">
        <div class="row justify-content-center">
            <div class="col-12 col-md-10 text-primary">
                <h1 class="text-center text-black">Inscrivez-vous</h1>
                <form method="post" action="">
                    <div class="mb-3">
                        <label for="userName" class="form-label">Nom</label>
                        <input type="text" class="form-control" id="userNom" name="nom" required>
                    </div>
                    <div class="mb-3">
                        <label for="userName" class="form-label">Prenom</label>
                        <input type="text" class="form-control" id="userPrenom" name="prenom" required>
                    </div>
                    <div class="mb-3">
                        <label for="userPassword" class="form-label">Mot de Passe</label>
                        <input type="password" class="form-control" id="userPassword" name="password" required>
                    </div>
                    <div class="mb-3">
                        <label for="userEmail" class="form-label">Email</label>
                        <input type="email" class="form-control" id="userEmail" name="email" required>
                    </div>
                    <button type="submit" class="btn btnprimary text-black">S'inscrire</button>
                </div>
            </div>
        </div>
    </div>
{% endblock %}
```

Ce fichier Twig génère une page d'inscription en héritant de base.html.twig. Il utilise Bootstrap pour la mise en page et contient un formulaire permettant aux utilisateurs de s'inscrire en fournissant plusieurs informations.

Explication détaillée :

Héritage du template principal

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

1.

- Ce fichier réutilise la structure de base.html.twig.

Définition du titre de la page

```
{% block title %}
```

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

```
    Inscription
```

```
{% endblock %}
```

2.

- Ajoute "Inscription" au titre défini dans le fichier parent.

{% block content %}

3.

- Tout ce qui suit est affiché dans la section principale de la page.

Mise en page avec Bootstrap

html

```
<div class="container-fluid">  
  <div class="row justify-content-center">  
    <div class="col-12 col-md-10 text-primary">
```

4.

- Utilisation de container-fluid et row justify-content-center pour centrer le formulaire.
- col-12 col-md-10 ajuste la largeur du formulaire selon l'écran.

5. Formulaire d'inscription

Champ Nom

html

```
<input type="text" class="form-control" id="userNom" name="nom" required>
```

- Permet de saisir le nom de l'utilisateur.

Champ Prénom

html

```
<input type="text" class="form-control" id="userPrenom" name="prenom"  
required>
```

- Permet de saisir le prénom.

.

Champ Mot de passe

html

```
<input type="password" class="form-control" id="userPassword"  
name="password" required>
```

- Champ sécurisé pour saisir un mot de passe.

Champ Email

html

```
<input type="email" class="form-control" id="userEmail" name="email" required>
```

- Permet de saisir une adresse e-mail.

Bouton d'inscription

html

```
<button type="submit" class="btn btnprimary text-black">S'inscrire</button>
```

- Bouton pour soumettre le formulaire.

Ce code affiche donc une page d'inscription avec des champs obligatoires, tout en assurant une mise en page propre et responsive grâce à Bootstrap.

Mais tout cela ne concerne que la table User, mais pour les fichiers et l'upload le code change légèrement

Pour les fichiers, nous avons également créé une Entity et un Model comme pour la table User, mais pour l'upload, nous avons créé une Entity différente :

```
<?php
declare (strict_types = 1);
namespace MyApp\Entity;

class Upload
{
    private array $extensions;
    private string $path;
    private int $size;
    public function __construct(array $extensions, string $path, int $size)
    {
        $this->extensions = $extensions;
        $this->path = $path;
        $this->size = $size;
    }
}
```

Le début reste similaire aux autres tables, avec l'annonce des variables et la fonction constructeur, mais contrairement aux autres avec des getter et

setter, ici il n'y a qu'une fonction

```
public function save(string $data): array
{
    $file = ['name' => null, 'error' => null];

    if (!isset($_FILES[$data]) || empty($_FILES[$data]['name'])) {
        return $file;
    }

    $fileExtension = strtolower(pathinfo($_FILES[$data]['name'], PATHINFO_EXTENSION));
    if (!in_array($fileExtension, $this->extensions)) {
        $file['error'] = 'Type de fichier non autorisé';
        return $file;
    }

    if ($_FILES[$data]['size'] > $this->size) {
        $file['error'] = 'Fichier trop volumineux';
        return $file;
    }

    $uniqueName = uniqid().'.'.$fileExtension;
    $destination = $this->path.'/'.$uniqueName;

    if (!move_uploaded_file($_FILES[$data]['tmp_name'], $destination)) {
        $file['error'] = 'Erreur lors de l\'upload';
        return $file;
    }

    $file['name'] = $uniqueName;
    return $file;
}
```

On commence par initialiser un tableau pour stocker les informations sur le fichier uploadé, tableau dans lequel les valeurs 'name' et 'error' sont définis comme null.

Ensuite si l'input file \$data n'existe pas dans \$_FILES ou si aucun fichier n'a été sélectionné (name est vide), la fonction retourne directement \$file.

Puis pour le \$fileExtension, on récupère l'extension du fichier en minuscule, puis on vérifie si l'extension est dans la liste d'extensions autorisées avec \$this->extensions.

Si ce n'est pas le cas, une erreur est ajoutée au tableau \$file et la fonction retourne ce dernier.

On va ensuite vérifier la taille du fichier (`$_FILES[$data]['size']`) avec la taille limite autorisée (`$this->size`). Retourne une erreur si le fichier est trop volumineux.

Les deux prochaines lignes permettent de :

- créer un nom unique pour le fichier en utilisant `uniqid()` (pour éviter les conflits de nom).
- créer ensuite le chemin complet de destination (`$this->path` est le répertoire de destination).

Le prochain if permet de déplacer le fichier depuis son emplacement temporaire (`tmp_name`) vers le dossier de destination.

Si l'opération échoue, une erreur est ajoutée au tableau `$file`.

Puis enfin, si l'upload réussit, le nom du fichier est enregistré dans `$file['name']` et retourné.

Si une erreur est survenue avant cette étape, `$file['error']` contiendra un message d'erreur correspondant.

Il a ensuite fallu modifier la fonction `addFichier` dans le `DefaultController` avec ces lignes ci :

```
if (!empty($_POST['nomFichier']) && !empty($_POST['dates'])) {  
  
    $upload = new Upload(['png', 'gif', 'jpeg', 'jpg', 'zip', 'txt', 'pdf'], 'images',  
500000);  
  
    $image = $upload->save('images');  
  
    if ($image['error'] !== null) {  
  
        echo 'Erreur lors de l\'upload : ' . $image['error'];  
  
        return;  
  
    }  
  
    $imageName = !empty($image['name']) ? $image['name'] : 'default.png';
```

```
$fichier = new Fichier(null, $nomFichier, $dates, $imageName);
```

la ligne “if (!empty(\$_POST['nomFichier']) && !empty(\$_POST['dates'])) {” permet de vérifier si les données envoyées via la méthode POST contiennent nomFichier et dates et que ceux-ci ne sont pas vides.

Si l'un des deux est manquant, ce bloc de code sera ignoré.

La ligne “\$upload = new Upload(['png', 'gif', 'jpeg', 'jpg', 'zip', 'txt', 'pdf'], 'images', 500000);” permet de créer une instance de la classe Upload.

Et les paramètres fournis au constructeur sont :

- ['png', 'gif', 'jpeg', 'jpg', 'zip', 'txt', 'pdf'] : Extensions autorisées.
- 'images' : Répertoire de destination pour l'upload.
- 500000 : Taille maximale du fichier en octets (environ 500 Ko).

La ligne \$image = \$upload->save('images'); permet d'appeler la méthode save() de la classe Upload (que vous avez montrée plus tôt) et tente d'enregistrer le fichier envoyé via l'input nommé 'images' et cela renvoie un tableau associatif contenant :

- 'name': Nom unique du fichier enregistré.
- 'error': Message d'erreur s'il y en a eu.

Puis les trois lignes if (\$image['error'] !== null) {

```
    echo 'Erreur lors de l\'upload : ' . $image['error'];
```

```
    return;
```

```
}
```

permettent de renvoyer un message d'erreur si 'error' n'est pas null et le return; met fin au script si une erreur survient.

Avec la ligne \$imageName = !empty(\$image['name']) ? \$image['name'] : 'default.png'; si un nom de fichier valide est retourné (\$image['name'] existe), il est assigné à \$imageName sinon, \$imageName prend la valeur par défaut 'default.png'.

Enfin la dernière ligne permet de créer une nouvelle instance de la classe Fichier.

Mais il faut également ajouter quelques lignes dans le fichier addFichier.html.twig :

Tout d'abord il faut ajouter `enctype="multipart/form-data"` dans le `<form action="" method="post">`, car c'est obligatoire pour envoyer des fichiers et cela permet l'envoi de données binaires en plus des données textuelles habituelles.

Puis il faut aussi ajouter `<input type="file" name="image" id="image" class="form-control"/>`

afin de permettre à l'utilisateur de sélectionner un fichier depuis son appareil.

`name="image"` correspond au nom qui sera utilisé pour récupérer le fichier dans `$_FILES['image']` côté serveur.

`id="image"` permet de relier cet `<input>` à l'élément `<label>`.

`class="form-control"` permet d'appliquer un style Bootstrap ou personnalisé pour avoir un champ stylisé.