Rapport de Projet

SI40 / WE4A-WE4B
Plateforme pédagogique : Moodle Simplifié
Réalisé par :

Ivann Vasic, Lucas Vigny

Table des matières

1. Introduction

1.1 Nos objectifs

Les objectifs pédagogiques de ce projet sont multiples :

- **Renforcer nos compétences en conception de bases de données** à travers l'application de la méthode Merise (MCD et MLD) et la création d'un SGBD optimisé.
- **Maîtriser les opérations SQL avancées**, notamment la gestion des droits d'accès, les jointures, les contraintes d'intégrité, et l'optimisation des requêtes.
- **Développer une application web complète en utilisant Symfony 7**, tout en respectant les bonnes pratiques de développement (sécurité, architecture MVC, séparation front-end/back-end).
- **Mettre en œuvre Bootstrap** pour améliorer l'ergonomie et l'interactivité du site sans utiliser d'autres bibliothèques non autorisées.
- **Comprendre et gérer des rôles utilisateurs complexes** (professeur, étudiant, administrateur) et sécuriser l'accès aux différentes fonctionnalités selon ces rôles.
- **Apprendre à organiser le travail en équipe**, en se répartissant les tâches de manière efficace et en assurant la cohérence entre la base de données et l'application web.
- **S'initier à la gestion d'un projet informatique complet**, de l'analyse des besoins jusqu'à la livraison d'une application fonctionnelle.

1.2 Choix stratégiques

Dans cette partie nous allons vous présenter les choix méthodologiques que nous avons fait tout au long du projet pour organiser notre travail efficacement et limiter les difficultés techniques liées à Symfony.

Tout d'abord, nous avons décidé d'utiliser **Bootstrap** afin de faciliter et uniformiser la conception visuelle de l'interface utilisateur. Ce choix nous a permis de gagner du temps sur l'intégration HTML/CSS tout en garantissant un rendu professionnel et responsive.

Conscients de la complexité qu'aurait représenté une transition tardive d'un projet classique PHP/HTML vers un projet Symfony complet, nous avons pris l'initiative de **nous former et de pratiquer Symfony 7 dès les premières semaines du semestre**.

Cela nous a permis de mieux maîtriser les outils proposés par le framework (Doctrine ORM, architecture MVC, gestion des routes, etc.).

1. Dans un premier temps, nous avons **développé l'ensemble de l'interface sans intégrer aucune logique de base de données**, en nous concentrant uniquement sur l'ergonomie, la structure HTML, et l'esthétique CSS de notre site. Cela nous a permis de disposer rapidement d'une base visuelle complète.

2. Par la suite, nous avons consacré **un temps important à la conception de la base de données**, en réalisant plusieurs itérations de notre **MCD (Modèle Conceptuel de Données)** et de notre **MLD (Modèle Logique de Données)**. Cette phase de réflexion a été appuyée par la création de **schémas de cas d'utilisation** (Use Case) et de **diagrammes de classes**, afin de garantir la cohérence de notre architecture avant toute implémentation.

3. Une fois le modèle validé, nous avons choisi de **créer entièrement notre base de données directement via Symfony**, en utilisant le terminal et les commandes Doctrine pour générer les entités, les migrations, et la synchronisation avec la base MySQL. Ce processus nous a permis d'avoir une base de données propre, bien reliée aux entités Symfony, et facile à maintenir.

4. Nous avons ensuite **progressivement relié la base de données à notre interface**, en développant de nombreux **contrôleurs** pour assurer la gestion des utilisateurs, des unités d'enseignement (UE), et des publications de contenus. Chaque fonctionnalité a été soigneusement intégrée et testée au fur et à mesure.

5. Une fois la majorité des fonctionnalités de base opérationnelles, nous avons entrepris de **rendre le site plus dynamique en implémentant du JavaScript et de l'AJAX**, notamment pour la suppression de posts et d'inscriptions, permettant ainsi des interactions fluides sans rechargement de page.

6. Enfin, selon le temps disponible, nous avons tenté d'**implémenter plusieurs fonctionnalités "Nice to Have"** proposées dans le cahier des charges, pour enrichir l'expérience utilisateur et ajouter des améliorations supplémentaires.

7. Dans la phase finale du projet, nous avons consacré **beaucoup de temps au débogage**, en cherchant à identifier et à corriger un maximum de failles ou de comportements inattendus dans notre application, afin de livrer un site web le plus stable et fonctionnel possible.

Nous avons travaillé sous la forme d'un binôme, la répartition et l'organisation des tâches ont été des éléments clés pour la réussite de ce projet qui fut très demandant en terme de temps et d'implication

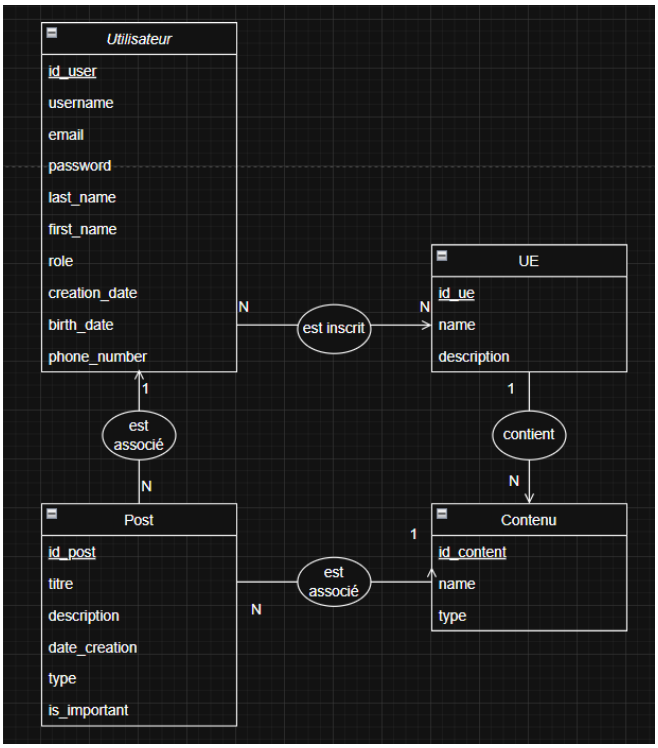
2. Analyse et Conception de la Base de Données

2.1 Méthodologie

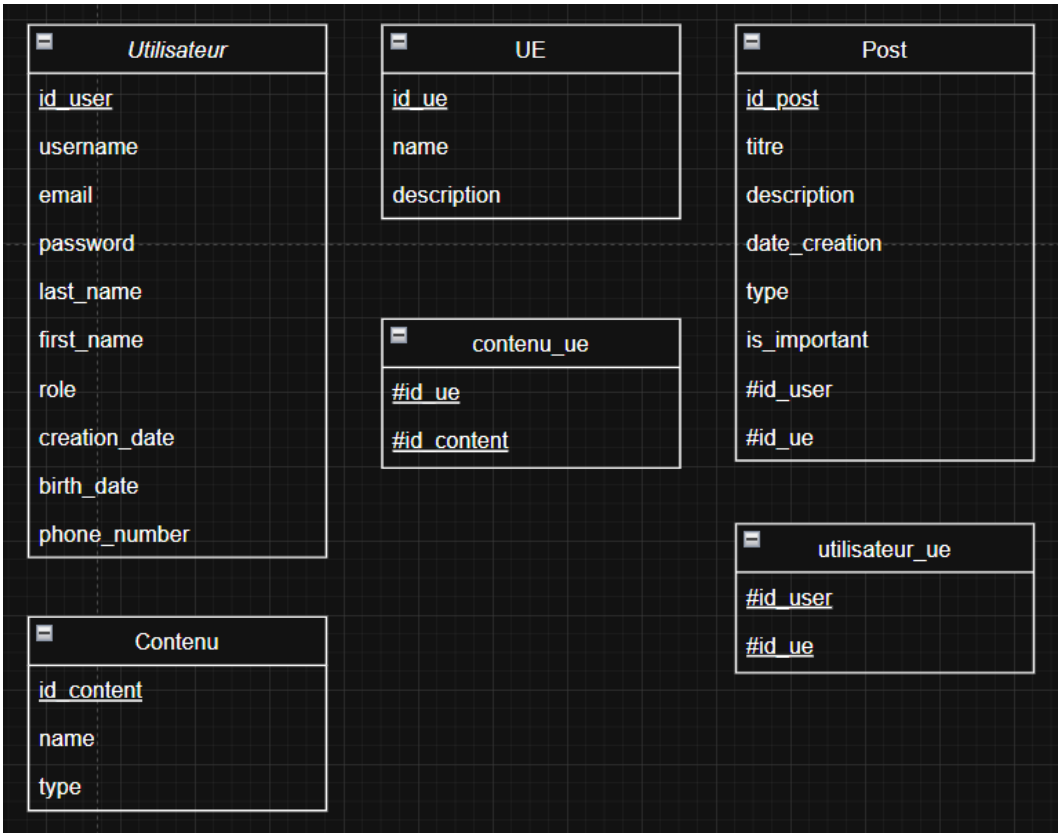
Pour concevoir notre base de données, nous avons utilisé la méthode **Merise**, qui structure la modélisation en trois niveaux :

- Le **MCD** pour représenter les entités (Utilisateur, UE, Post, etc.) et leurs relations de manière conceptuelle.
- Le **MLD**, qui transforme ces entités en tables relationnelles normalisées avec clés primaires et étrangères.
- Le **MPD**, intégré directement via Symfony et Doctrine, pour générer la base finale avec contraintes et index.

2.2 MCD



2.3 MLD



Nous avons suivi les règles suivantes pour construire notre MLD depuis le MCD :

- **Règle 1 : Chaque entité devient une table avec attributs → utilisateur, UE, post, contenu**
- **Règle 2 : Association de type (1,N) devient clé étrangère dans le fils**

Exemple : post → foreign key vers utilsateur et UE

- **Règle 3 : Association N:N devient une table**

Exemple :

- Utilisateur_UE (id_utilisateur, id_ue)
- contenu_UE (id_ue, id_contenu)

3. Implémentation du SGBD

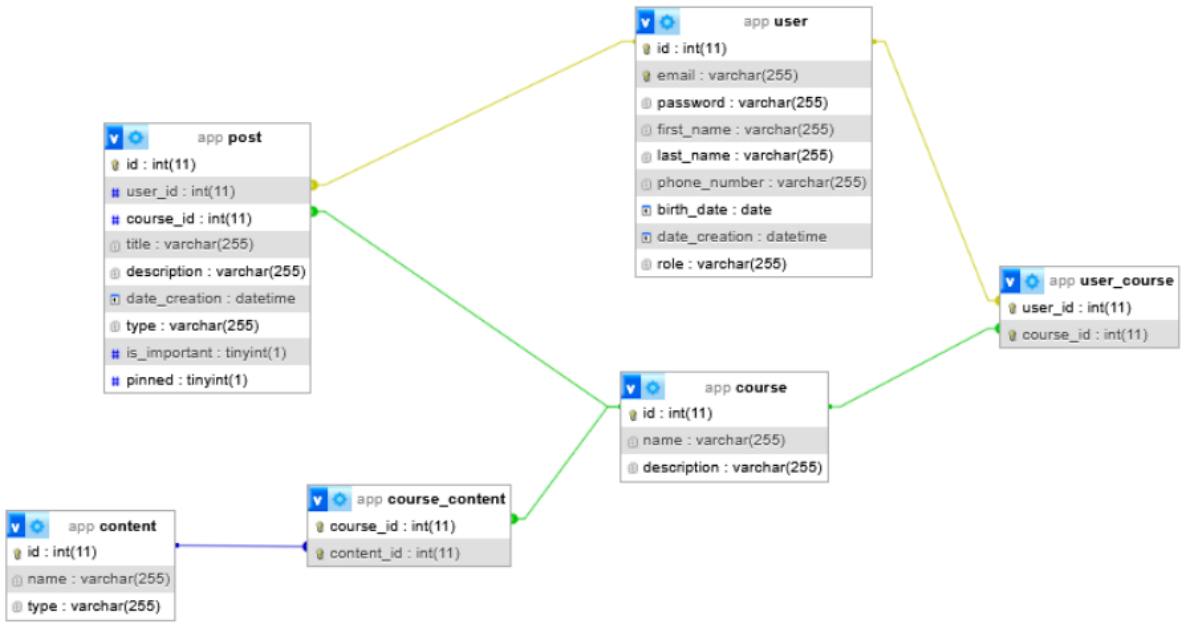
3.1 Choix de la technologie MySQL/XAMPP

Pour la mise en œuvre de notre base de données, nous avons opté pour **MySQL** via la suite **XAMPP**, conformément aux recommandations du cahier des charges. Ce choix nous a permis de bénéficier d'un environnement intégré (Apache, PHP, MySQL) compatible avec Symfony 7, facilitant le développement, le test et la maintenance de l'application. L'utilisation de **phpMyAdmin** nous a également permis de vérifier visuellement l'état de notre base et d'effectuer des manipulations rapides.

3.2 Création des tables et relations

Les tables ont été générées à partir des **entités Symfony**, en utilisant les **migrations Doctrine**. Chaque entité du modèle logique de données (Utilisateur, UE, Post, Contenu) a été traduite en table, et les associations N:N (utilisateur_ue, contenu_ue) ont été créées manuellement via Doctrine.

Le schéma de base a donc été généré automatiquement par Symfony, avec un contrôle complet via les fichiers d'entités, garantissant la cohérence entre la structure logique et le code.



3.3 Données de test et peuplement de la base

Pour tester notre application, nous avons créé un jeu de **données fictives** injectées via :

- Les **fixtures Symfony** (via ORMFixtures),

- Des **fichiers SQL exportés/importés** depuis phpMyAdmin (uniquement pour des tests)
- Ou encore via l'interface d'administration de l'application (création d'utilisateurs, UEs, posts).

Ces données comprenaient des utilisateurs de chaque rôle (étudiants, enseignants, admins), plusieurs UEs, des posts de test (messages et fichiers), et des contenus rattachés.

Cela nous a permis de tester chaque fonctionnalité de l'application de manière réaliste et d'identifier d'éventuelles incohérences fonctionnelles ou erreurs de requêtes.

4. Architecture de l'Application et Modélisation Dynamique

4.1 Architecture MVC (Model-View-Controller)

Nous avons structuré notre application Symfony 7 selon le **modèle MVC** (Modèle-Vue-Contrôleur), une architecture logicielle qui favorise la séparation des responsabilités et la maintenabilité du code.

- Modèle : Le **modèle** correspond aux **entités PHP** qui représentent les objets métiers manipulés dans la plateforme :
  - User : représente les utilisateurs (étudiants, professeurs, administrateurs).
  - Course : représente les unités d'enseignement (UE).
  - Post : représente les messages, fichiers ou annonces publiés dans les UEs.

  Ces entités sont automatiquement **mappées à la base de données** via Doctrine ORM, ce qui facilite la gestion des enregistrements SQL sans avoir à écrire manuellement les requêtes.
- Vue : La **vue** correspond aux fichiers HTML générés à l'aide du moteur de templates **Twig**, utilisé par Symfony. Les vues se contentent **d'afficher les données** préparées par le contrôleur, sans y inclure de logique métier. Cela garantit une interface propre, claire, et facilement modifiable.
- Contrôleur : Les contrôleurs Symfony reçoivent les requêtes HTTP de l'utilisateur, déclenchent les traitements nécessaires, et transmettent les résultats à la vue.
  Ils assurent :
  - La lecture ou modification des entités
  - La création de nouveaux objets (ex : post, utilisateur, cours)
  - La préparation des données avant transmission à la vue

  Par exemple, le CoursesController permet de créer un post, récupérer les utilisateurs d'une UE, ou encore afficher les contenus associés à une formation.

L'utilisation du modèle MVC nous a permis de mieux organiser notre code, de séparer clairement les responsabilités, et de faciliter la maintenance et l'évolution de l'application tout au long du projet.

4.2 Diagramme de cas d'utilisation (Use Case)

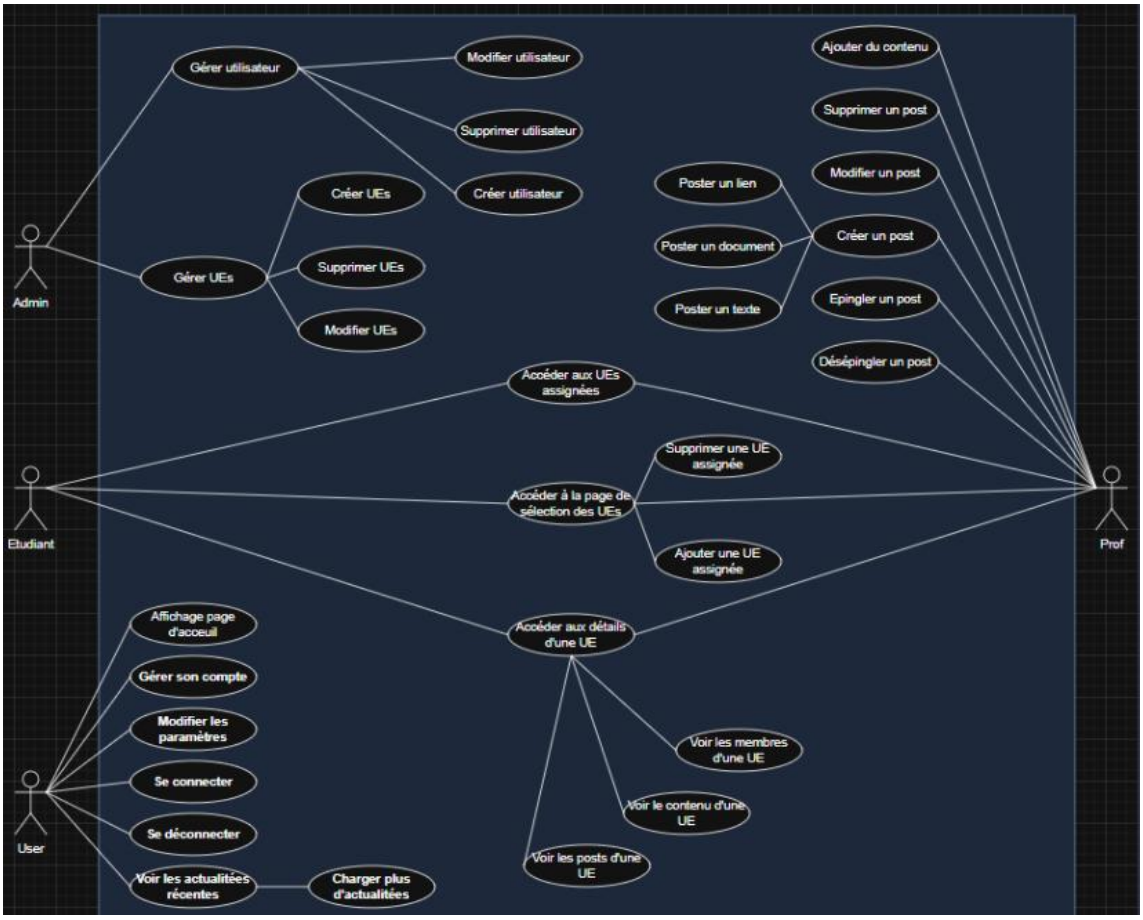
Afin de formaliser les différentes interactions possibles entre les utilisateurs et notre plateforme, nous avons réalisé un **diagramme de cas d'utilisation UML**. Celui-ci permet d'avoir une vue d'ensemble des fonctionnalités accessibles selon les **rôles** définis dans notre système.

User : un acteur générique dont héritent tous les rôles (Admin, Professeur, Étudiant). Il possède les droits communs (connexion, consultation des actualités, gestion du compte...).

Étudiant : accède à ses UEs, peut consulter les contenus et posts.

Professeur : publie et gère du contenu dans les UEs auxquelles il est affecté.

Administrateur : gère l'ensemble de la plateforme, les utilisateurs et les UEs.



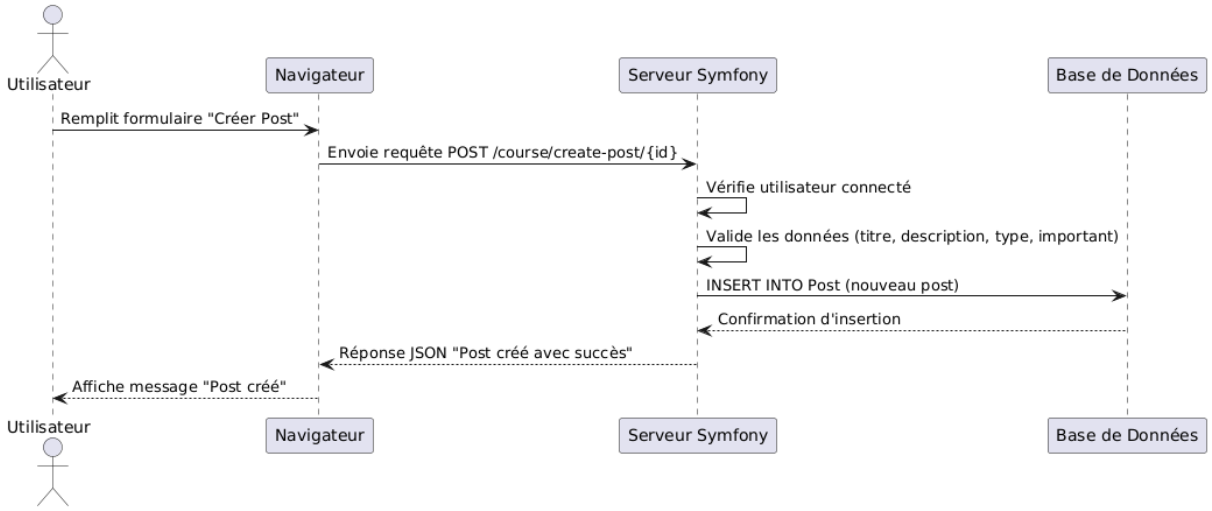
Chaque cas d'utilisation du diagramme correspond concrètement à une **route Symfony** dans notre application, associée à un contrôleur responsable du traitement et de l'affichage.

Ce diagramme nous a aidés, dès les premières étapes du projet, à **clarifier les fonctionnalités à implémenter** et à **répartir les responsabilités** selon les profils utilisateurs.

4.3 Diagramme de séquence pour la création d'un post

Le diagramme ci-dessous illustre le déroulement complet d'un scénario de **création d'un post dans une UE**, tel qu'il est implémenté dans notre application Symfony. Il nous a beaucoup aidé lors de notre travail sur l'implémentation de la création de post pour les

professeur.



Ce diagramme illustre le **flux de données complet** entre les différents composants de l'application selon le modèle MVC. Il permet de visualiser l'enchaînement des responsabilités :

- L'interface gère l'entrée utilisateur et la restitution visuelle.
- Le contrôleur centralise la logique métier.
- Le modèle (entité Post) est stocké et géré via Doctrine dans la base de données.

5. Fonctionnalités SQL Implémentées

5.1 Authentification des utilisateurs

L'authentification est gérée par Symfony avec hachage des mots de passe via l'interface UserPasswordHasherInterface. Lors de l'inscription (RegistrationController), le mot de passe est encodé avant d'être stocké dans la base. À la connexion, Symfony compare le mot de passe soumis avec celui stocké (hashé), garantissant la sécurité des identifiants.

5.2 Gestion des rôles (admin, étudiant, professeur)

Chaque utilisateur possède un rôle stocké dans la base (ROLE_ELEVE, ROLE_PROF, ROLE_ADMIN, ROLE_PROF_ADMIN). Ces rôles sont utilisés dans l'interface pour restreindre ou autoriser l'accès à certaines fonctionnalités :

- Les administrateurs peuvent gérer tous les comptes et UEs (AdminController)
- Les professeurs peuvent créer du contenu et des posts dans leurs UEs (CoursesController)
- Les étudiants peuvent consulter les contenus liés à leurs UEs (MenuController, UESelectionController)

Lorsqu'un utilisateur avec un rôle existant s'inscrit de nouveau avec un rôle différent, son rôle peut être mis à jour automatiquement (ex : passage en ROLE_PROF_ADMIN).

5.3 Gestion des unités d'enseignement (UE)

Les administrateurs peuvent créer, modifier et supprimer des UEs via des requêtes Doctrine (AdminController). Chaque UE est liée à plusieurs utilisateurs via une table de liaison, et contient des posts ainsi que des contenus pédagogiques. Une vérification empêche les doublons, et chaque professeur peut avoir un maximum de 3 UEs, les étudiants jusqu'à 5 (UESelectionController).

5.4 Création et gestion des contenus pédagogiques (posts, fichiers)

Les utilisateurs autorisés (professeurs, profs-admins) peuvent publier :

- Des **posts** (titre, description, type, important ou non)
- Des **fichiers** via des entités Content, liés aux UEs via course->addContent(...)

Les posts peuvent être créés, supprimés ou marqués comme importants. L'implémentation AJAX (dans CoursesController) permet d'interagir dynamiquement avec ces contenus sans recharger la page.

5.5 Requêtes optimisées : jointures, vues, filtres

Les requêtes Doctrine utilisent des jointures et des ordres de tri pour garantir la pertinence et la rapidité :

- Les posts sont triés par importance (isImportant) et par date de création (dateCreation)
- Le menu utilisateur ne récupère que les 3 derniers posts associés à ses UEs (MenuController)
- Des jointures explicites sont utilisées (join('p.course', 'c')) pour filtrer uniquement les posts liés aux UEs de l'utilisateur

5.6 Utilisation de Doctrine

Dans notre projet, les requêtes SQL ne sont pas écrites manuellement : elles sont entièrement **gérées par Doctrine**, l'ORM (Object-Relational Mapper) utilisé par Symfony. Doctrine se charge de **traduire les manipulations d'objets PHP (entités)** en requêtes SQL exécutées sur la base de données. Cela nous permet de bénéficier d'une gestion automatique des insertions, mises à jour, suppressions et recherches.

Par exemple, pour récupérer des posts liés à une UE, nous utilisons le **QueryBuilder de Doctrine**, qui génère dynamiquement une requête SQL basée sur nos critères (jointures, tri, filtres). Les requêtes générées sont sécurisées (protégées contre l'injection SQL) et optimisées grâce à l'utilisation de **mappings, relations entre entités**, et **annotations Symfony**.

REQUETE	SQL	DOCTRINE
SELECT	SELECT * FROM user; SELECT * FROM course; SELECT * FROM post ORDER BY pinned DESC, date_creation DESC LIMIT 3;	$em->getRepository(User::class)->findAll(); $em->getRepository(Post::class)->createQueryBuilder('p')->orderBy('p.pinned', 'DESC')->getQuery()->getResult();
INSERT	INSERT INTO course (name, description) VALUES (...); INSERT INTO post (title, description, type, is_important, user_id, course_id, date_creation) VALUES (...);	$em->persist($user); $em->persist($course); $em->persist($post); $em->flush();
UPDATE	UPDATE user SET password = '...' WHERE id = ...; UPDATE post SET pinned = 1 WHERE id = ...;	$user->setPassword(...); $post->setPinned(true); $em->flush();
DELETE	DELETE FROM user WHERE id = ...; DELETE FROM course WHERE id = ...;	$em->remove($user); $em->remove($course); $em->flush();

6. Problèmes Rencontrés et Solutions Apportées

6.1 Problèmes techniques (BDD, sécurité, framework...)

Plusieurs problèmes ont émergé avec l'utilisation de doctrine :

- Certaines requêtes complexes, comme l'affichage des derniers posts par UE en respectant les priorités (épinglé > date), ont nécessité l'utilisation directe de QueryBuilder au lieu des méthodes findBy.
- Un oubli de l'attribut #[IsGranted("ROLE_ADMIN")] sur la route /admin/load-more-posts a provoqué des erreurs 403 empêchant le chargement asynchrone des données en tant qu'admin.

Problème de cache et de persistance :

Lors du développement collaboratif, nous avons aussi rencontré des divergences entre bases de données locales. Cela a entraîné des incohérences lorsqu'un membre modifiait les entités ou ajoutait des données de test, non synchronisées automatiquement. Pour résoudre cela, nous avons systématisé l'usage de fixtures ou de commandes Doctrine (schema:update, fixtures:load) pour uniformiser les environnements.

6.2 Problèmes de logique métier (rôles, associations...)

Modélisation des rôles :

La gestion des rôles a représenté un défi majeur. Notamment lorsqu'un utilisateur existant en tant que prof s'enregistrait de nouveau en tant que admin, une logique spéciale a été implémentée, et il est transformé en ROLE_PROF_ADMIN plutôt que rejeté.

Relations entre utilisateurs et UE :

L'association entre un utilisateur et plusieurs UE est au cœur de notre logique. Un étudiant peut rejoindre jusqu'à 5 UE, un professeur jusqu'à 3. La vérification de cette contrainte a dû être ajoutée dans le contrôleur UESelectionController. Des bugs sont survenus au départ, notamment des ajouts ou suppressions de cours sans mise à jour cohérente en base. Nous avons donc centralisé les validations et les flush dans des blocs conditionnels bien délimités.

7. Pistes d’améliorations et perspectives pour la partie WE4B

Plusieurs axes d’amélioration ont été identifiés afin de rendre notre plateforme plus robuste, plus ergonomique et plus riche en fonctionnalités :

- **Développement de nouvelles fonctionnalités :**
  - Implémenter un système de messagerie privée entre étudiants et professeurs.
  - Permettre l’ajout de commentaires sous les posts, afin de favoriser l'interaction autour des contenus publiés.
  - Ajouter un module de notification pour alerter les utilisateurs en cas de nouveau post, d’inscription à une UE, ou de modification de contenu.
- **Gestion avancée des contenus :**
  - Permettre l’upload et le téléchargement de fichiers pédagogiques (PDF, ZIP, images, etc.) de manière sécurisée.
  - Implémenter un système de validation pour les contenus avant publication, pour assurer une modération par les enseignants ou administrateurs.
- **Amélioration de la BDD :**
  Création d'une table user_role
  Actuellement, le rôle d'un utilisateur est stocké directement sous forme d'un champ texte (role) dans la table user. Cette approche est simple mais présente plusieurs limites :
  - Risque d'erreurs de saisie ou de mauvaise cohérence sur les rôles.
  - Difficulté pour ajouter de nouveaux rôles ou gérer plusieurs rôles par utilisateur.

  Une amélioration importante serait d'introduire une table supplémentaire user_role, qui permettrait de lier un utilisateur à un ou plusieurs rôles (relation N:N).

8/ Conclusion

Ce projet nous a donné l'opportunité de créer une plateforme pédagogique opérationnelle, inspirée des environnements d'apprentissage tels que Moodle.

Nous avons utilisé nos compétences en développement web (Symfony 7, MySQL, Bootstrap, AJAX) et en ingénierie des bases de données (modélisation Merise, requêtes SQL via Doctrine).

Nous avons organisé notre travail autour d'une architecture MVC claire, permettant une séparation des tâches et une grande flexibilité.

La modélisation initiale avec la méthode Merise nous a permis de garantir la cohérence et l'intégrité de la base de données, tout en facilitant son intégration dans Symfony.

La mise en place de fonctionnalités complexes telles que la gestion des rôles, l'interaction AJAX ou les contrôles d'accès nous a confrontés à des défis techniques concrets, que nous avons pu surmonter par itérations.

Ce projet nous a permis de mieux appréhender les enjeux d'un développement full-stack structuré et nous a préparés pour la suite du module WE4B, où nous pourrons enrichir la plateforme de nouvelles fonctionnalités (messagerie, notifications, suivi de progression...).

Nous sommes fiers du travail accompli en binôme et heureux d'avoir livré une application fonctionnelle, claire, sécurisée et évolutive.