UE-L204 MINI-PROJET

Rapport final

Sylvain Chambon,
Jade Faroux,
Jeanne Salvadori,
Zoé Van De Moortele

14 décembre 2025

Table des matières

Tables des requêtes SQL

Tables des scripts PHP

Première partie

Le projet

1 Méthodes de travail et outils utilisés

- **Teams** pour la communication et le partage de fichiers ;
- **GitHub** pour la mise en commun du code sur un dépôt et pouvoir mieux gérer les modifications via le système de branche ;
- Forum de groupe de l'UE : communication et retour sur l'avancement du projet ;
- **Word** pour une rédaction commune et LaTeXpour la finalisation des rapports ;
- un système d'intelligence artificielle générationnelle (**Gemini**) pour générer les entrées dans les tables.
- **Diagram.net** pour créer un modèle de notre future BDD et visualiser les tables et leurs connexions
- **PhpMyAdmin** pour la création des tables et la génération des entrées dans ces table
- **VS Code** (ou autre éditeurs) : rédaction du code
- **XAMPP** sous Windows ou un stack **LAMP** sous Linux.

2 Organisation du groupe

2.1 Échanges synchrones et asynchrones

- Réunion synchrone par semaine (soit 2 en tout) pour tout le groupe, et autres réunions synchrones pour faire des points d'avancement sur certaines parties du projet, expliquer des changements, etc.
- Communication continue tout au long du projet, points d'étapes réguliers entre nous.

2.2 Répartition des tâches

Afin d'assurer une progression efficace et structurée du projet, les tâches ont été réparties au sein du groupe en fonction des différents domaines à couvrir. Chaque membre a pris en charge une ou plusieurs parties spécifiques du projet, tout en participant activement aux réflexions communes et aux décisions globales. Cette organisation a permis de travailler efficacement tout en garantissant la cohérence et l'évolution continue du projet.

Tâches	**Assignation**
Création du Github et de la BDD	Sylvain Chambon
Mise à jour des idées et de l'avancement du projet	Sylvain Chambon
Mise en forme des rapports sur LaTeX	Sylvain Chambon
Scénario du projet	Zoe Van De Moortele
Gestion de l'admin et des sessions	Jeanne Salvadori
Gestion des enseignants	Sylvain Chambon
Gestion des étudiants	Zoe Van De Moortele
Mise en forme CSS	Jade Faroux
Création du support de présentation	Jade Faroux
Création du rapport final	Jeanne Salvadori

Tâches communes
Création, reflexion sur le projet et ses fonctionnalités
Amélioration continue, apport d'idées
Communication continue

3 Difficultés rencontrées

- Prise en main de la BDD avec PDO : configuration PDO différente selon Windows / Linux, activation des extensions PDO MySQL ou du serveur Apache.
  ☑ **Résolue**
- Problème : les mots de passe fournis dans la BDD ne correspondaient pas au hash réel de "123456". Nous avons fait des tests avec `password_hash()` et `password_verify()` ⟶ découverte que le hash était invalide ⟶ remplacement des mots de passe hachés directement en BDD.
  ☑ **Résolue**
- Gestion des chemins relatifs entre les pages (`index.php`, `pages/accueil.php`, `pages/deconnexion` erreurs 404 ou redirections incorrectes à cause de chemins mal construits
  ☑ **Résolue**
- Hashages réalisés par l'IA pour les utilisateurs fictifs se sont avérés faux. NOus avons dû les refaire à l'aide de commande PHP.
  ☑ **Résolue**
- Manque de temps personnel pour cette semaine pour la plupart d'entre nous à causes de raisons personnelles. À adapter en deuxième semaine.

Deuxième partie

Code, techniques et choix de développement

Depuis le début du projet, nous avons progressivement fait évoluer l'architecture et le code afin de gagner en lisibilité, en sécurité et en maintenabilité. Le projet a d'abord été développé avec des fonctionnalités essentielles, puis celles-ci ont été progressivement améliorées et complétées au fil du temps.

4 Étude préliminaire

4.1 Scénario

Créer un site universitaire avec différents niveaux d'accès : étudiant et professeur, éventuellement un administrateur.

Les professeurs peuvent :

- Créer, modifier, supprimer des cours.
- Voir la liste d'étudiant inscrit.
- Gérer les inscriptions (accepter/refuser les étudiants dans leurs cours).
- Modifier leur profil (informations personnelles).

Les étudiants peuvent :

- Consulter l'ensemble des cours disponibles.
- Rechercher des cours (nom, professeur, domaine).
- S'inscrire à des cours.
- Modifier leur profil (informations personnelles).

Si assez de temps, mettre en place un système de note (prof donne des notes et les étudiants peuvent y accéder), et d'emploi du temps.

4.2 Objectifs adaptés à notre scénario

4.2.1 Page de connexion

- Formulaire d'entrée où l'utilisateur entre son login et mot de passe.
- Système vérifie si cet utilisateur existe dans la BDD.
- Le mot de passe doit être crypté.
- Si correct, accès au portail étudiant ou professeur.
- Si incorrect, message d'erreur.
- Changement de mot de passe lors de la première connexion (sécurité).

4.2.2 Page de recherche

- Champ de recherche de cours (par nom, professeur ou domaine).
- Doit interroger la BDD avec les critères de recherche.
- Afficher les résultats sous forme de tableau, ou message d'erreur si aucun résultat.

4.2.3 Page d'ajout de contenu

- Les professeurs ont une page dédiée pour ajouter, modifier ou supprimer leur cours.
- Les professeurs et étudiants peuvent modifier leurs informations personnelles (mail, tel, mot de passe...).
- Toutes ces modifications doivent être enregistrées dans la BDD.

4.2.4 Vérification des données

Avant d'enregistrer quoi que ce soit dans la BDD, tests de sécurité à effectuer systématiquement :

- Avant d'enregistrer quoi que ce soit dans la BDD, tests de sécurité à effectuer systématiquement :
- Vérifier que les champs obligatoires soient remplis
- Les formats des entrées doivent être corrects
- Pas de caractères dangereux (balise html)
- Messages d'erreur si une de ces vérifications est non conforme

5 Finalisation

5.1 Arborescence

Choix d'une arborescence claire :

1. Le dossier `assets` regroupe tout ce qui concerne la mise en forme et les ressources statiques (CSS, images, polices).
2. Le dossier `classes` contient les classes PHP, notamment celles dédiées à l'accès à la base de données et à la centralisation des requêtes SQL.
3. Le dossier `docs` est utilisé pour stocker des documents de travail, notamment des fichiers Markdown servant à noter les idées, les pistes d'amélioration et l'avancement du projet.
4. Le dossier `pages` est structuré en sous-dossiers afin de séparer la logique applicative (logic) de l'affichage (views). Cette séparation, faite en deuxième semaine, permet de rendre le code plus lisible et plus facile à maintenir.
5. Au premier niveau du projet, on retrouve les fichiers `index.php`, `config.php`, `functions.php` ainsi que les scripts SQL de la base de données, qui centralisent la configuration, l'initialisation et les fonctions communes utilisées par l'ensemble des pages.

5.2 Fonctionnalités

Au fil du développement, plusieurs fonctionnalités majeures ont été implémentées :

- Un système **d'authentification** permettant de différencier les utilisateurs **selon leur rôle** (administrateur, enseignant, étudiant). En fonction de ce rôle, l'utilisateur est redirigé vers des pages spécifiques (`admin.php`, `teacher.php` et `student.php`) et n'a accès qu'aux fonctionnalités qui le concernent.
- L'**espace administrateur** permet de gérer les utilisateurs : lister les enseignants et les étudiants, modifier leurs informations, les supprimer, ou encore en ajouter via des formulaires dédiés.
- L'**espace enseignant** propose quant à lui des fonctionnalités liées aux cours, comme la consultation du catalogue, la création de nouveaux cours avec gestion des prérequis, et des actions préparatoires pour l'enseignement ou la suppression de cours.
- L'**espace étudiant** propose des fonctionnalités qui permettent de lister tous les cours de l'université, de pouvoir s'y inscrire et ensuite de pouvoir visualiser ses propres cours. Par contre, certains cours nécessitent des prérequis à l'inscription. Exemple : si on veut s'inscrire à `INFO-L204`, il faudra avoir suivi le cours `INFO-L104`.
- Un fichier de log `database-errors.log` est également utilisé afin de **tracer les erreurs** liées à la base de données, ce qui facilite le débogage et l'analyse des problèmes côté serveur.

6 Base de données

Nous avons modélisé notre base de données dans diagrams.net. Nous avons pensé à définir deux utilisateurs pour notre base de données : l'enseignant et l'étudiant. En structurant notre base, nous nous sommes aperçu que ces deux entités partageaient beaucoup d'attributs donc nous avons préférer définir l'entité plus générale d'`utilisateur` et définir les entités `enseignant` et `etudiant` comme des héritages de l'entité `utilisateur` avec certains attributs spécifiques.

Dans cette base de données, nous aurons évidemment aussi une entité `cours`.

Nous aurons enfin des tables de liaison :

- une table permettant d'associer des enseignants avec des cours (entité `enseigne`) ;
- un autre permettant d'associer des étudiants avec des cours (entité `etudiant`) ;
- une dernière permettant de définir des contraintes sur les inscriptions dans certains cours (entité `prerequis`).

6.1 Entités

6.1.1 Entité *utilisateur*

L'entité `utilisateur` aura les attributs (voir requête n° 1) :

- `login` : identifiant de connexion ;
- `mot_de_passe` : mot de passe pour la connexion : il sera encrypté par l'application (ce n'est pas de la responsabilité de la BBD) ;
- `mot_de_passe_provisoire` : booléen servant de drapeau pour savoir si le mot de passe initial a été changé ou pas ;
- `nom`, `prenom`, `email` : pour renseigner des élément lié à la personne et son contact ;
- `role` : trois rôles sont défini ici pour pouvoir tester les droits relatifs à chaque utilisateur de la BDD ;
- `date_creation` pour stocker la date à laquelle l'utilisateur a été saisi dans la BDD ;
- `actif` drapeau pour connaître si la personne est encore en activité dans la faculté.

Requête SQL n° 1 : Création de la table `utilisateur`

```sql
CREATE TABLE utilisateur (
id INT PRIMARY KEY AUTO_INCREMENT,
login VARCHAR(255) UNIQUE NOT NULL,
mot_de_passe VARCHAR(255) NOT NULL,
mot_de_passe_provisoire BOOLEAN DEFAULT TRUE,
nom VARCHAR(100) NOT NULL,
prenom VARCHAR(100) NOT NULL,
email VARCHAR(255) UNIQUE NOT NULL,
```

```sql
    role ENUM('enseignant', 'etudiant', 'admin') NOT NULL,
    date_creation TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    actif BOOLEAN DEFAULT TRUE
    );
```

On se rend compte que tous ces attributs serons partagés à la fois par les enseignants et par les étudiants.

Un ajout d'un utilisateur dans la BDD peut donc se déclarer ainsi (voir exemple de requête nº 2)

Requête SQL nº 2 : Création d'un utilisateur dans la table

```sql
    (1, 'turing', '<mot-de-passe>', 'Turing', 'Alan', 'alan.turing@univ.fr', '
        enseignant'),
```

6.1.2 Entité *enseignant*

L'entité enseignant hérite de de l'entité utilisateur avec quelques spécificités :

- bureau : localisation de la salle de travail de l'enseignant ;
- telephone : numéro ;
- specialite : domaine d'expertise ;
- statut : sous quel titre l'enseignant a-t-il été embauché.

On constate la présence d'une clé étrangère afin de lier l'entité enseignant avec une entité utilisateur existante (héritage).

Requête SQL nº 3 : Création de la table enseignant

```sql
    CREATE TABLE enseignant (
    id INT PRIMARY KEY AUTO_INCREMENT,
    utilisateur_id INT UNIQUE NOT NULL,
    bureau VARCHAR(50),
    telephone VARCHAR(20),
    specialite VARCHAR(255),
    statut ENUM('titulaire', 'vacataire', 'contractuel') DEFAULT 'titulaire',
    FOREIGN KEY (utilisateur_id) REFERENCES utilisateur(id) ON DELETE CASCADE
    );
```

On ajoutera un enseignant ainsi dans la base (voir exemple de requête nº 4)

Requête SQL nº 4 : Création d'un enseignant dans la table

```sql
    INSERT INTO enseignant (utilisateur_id, bureau, telephone, specialite, statut)
        VALUES
    (1, 'B101', '0102030401', 'Intelligence Artificielle', 'titulaire');
```

6.1.3 Entité *etudiant*

L'entité `etudiant` hérite de de l'entité `utilisateur` avec quelques spécificités :

- le classique `numero_etudiant`, comme référence nationale ;
- `niveau` qui référence si l'étudiant est en licence ou master et en quelle année ;
- `date_inscription` qui peut-être de la `date_creation` si l'étudiant n'a pas validé ses frais de scolarité par exemple.

On constate ici aussi la présence d'une clé étrangère afin de lier l'entité `étudiant` avec une entité `utilisateur` existante (héritage, comme pour l'entité `enseignant`).

Requête SQL n° 5 : Création de la table `etudiant`

```sql
CREATE TABLE etudiant (
id INT PRIMARY KEY AUTO_INCREMENT,
utilisateur_id INT UNIQUE NOT NULL,
numero_etudiant VARCHAR(20) UNIQUE NOT NULL,
niveau ENUM('L1', 'L2', 'L3', 'M1', 'M2') NOT NULL,
date_inscription DATE NOT NULL,
FOREIGN KEY (utilisateur_id) REFERENCES utilisateur(id) ON DELETE CASCADE
);
```

Pour ajouter un étudiant dans la base, on pourra procéder ainsi (voir exemple de requête n° 6)

Requête SQL n° 6 : Création d'un étudiant dans la table

```sql
INSERT INTO etudiant (utilisateur_id, numero_etudiant, niveau, date_inscription
    ) VALUES
(12, '20250001', 'L3', '2024-09-01');
```

6.1.4 Entité *cours*

L'entité `cours` a les attributs suivants :

- `code` joue le rôle d'identifiant visuel et sera pratique pour les recherches ;
- `nom` intitulé du cours ;
- `credits` pour garder le nombre de crédits ECTS ;
- `description` pour donner le détail sdu contenu du cours ou un syllabus ;
- `capacite_max` pour gérer le nombre d'étudiants qui peuvent s'inscrire ;
- `annee_universitaire` indique quand le cours est proposé ;
- `actif` en un drapeau booléen permettant de savoir si le cours est proposé en enseignement ou pas.

Requête SQL n° 7 : Création de la table `cours`

```sql
CREATE TABLE cours (
id INT PRIMARY KEY AUTO_INCREMENT,
```

```
3   code VARCHAR(20) UNIQUE NOT NULL,
4   nom VARCHAR(255) NOT NULL,
5   credits INT NOT NULL CHECK (credits > 0),
6   description TEXT,
7   capacite_max INT DEFAULT 30,
8   annee_universitaire VARCHAR(9) NOT NULL, -- "2025-2026"
9   actif BOOLEAN DEFAULT TRUE
10  );
```

La création d'un cours peut s'effectuer à l'aide de la requête suivante :

Requête SQL nº 8 : Création d'un cours dans la table

```
1   INSERT INTO cours (code, nom, credits, description, annee_universitaire) VALUES
2   ('INFO-L101', 'Introduction à l''Algorithmique', 6, 'Logique, pseudo-code,
        variables et boucles', '2025-2026');
```

6.2 Tables de liaison

À ce stade, nous n'avons généré dans la base qu'une seule table de liaison : celles concernant les prérequis.

Pour les autres tables de liaison (`enseigne` et `inscription`), nous pensons qu'il faudra plutôt les faire depuis l'interface.

Un prérequis pour un cours fonctionne ainsi : on réunit deux ID de cours, la première référençant un Cours A, la deuxièmes référençant un Cours B nécessaire pour suivre le Cours A. Ainsi, cette relation est bien une relation *many-to-many* :

- le Cours A ayant besoin de plusieurs cours pour être suivi ;
- le Cours A pouvant être nécessaire à d'autres cours.

Voici comment nous avons donc défini cette table :

Requête SQL nº 9 : Table de liaison `prerequis`

```
1   CREATE TABLE prerequis (
2   cours_id INT NOT NULL,
3   prerequis_cours_id INT NOT NULL,
4   PRIMARY KEY (cours_id, prerequis_cours_id),
5   FOREIGN KEY (cours_id) REFERENCES cours(id) ON DELETE CASCADE,
6   FOREIGN KEY (prerequis_cours_id) REFERENCES cours(id) ON DELETE CASCADE,
7   CHECK (cours_id ≠ prerequis_cours_id)
8   );
```

La dernière ligne nº 7 permet d'interdire un auto-référencement.

Pour déclarer une telle relation, nous pouvons faire (voir exemple de requête ci-dessous nº 10) :

Requête SQL nº 10 : Création d'une relation de prérequis

```sql
INSERT INTO prerequis (cours_id, prerequis_cours_id) VALUES
((SELECT id FROM cours WHERE code='INFO-L201'), (SELECT id FROM cours WHERE
    code='INFO-L101'));
```

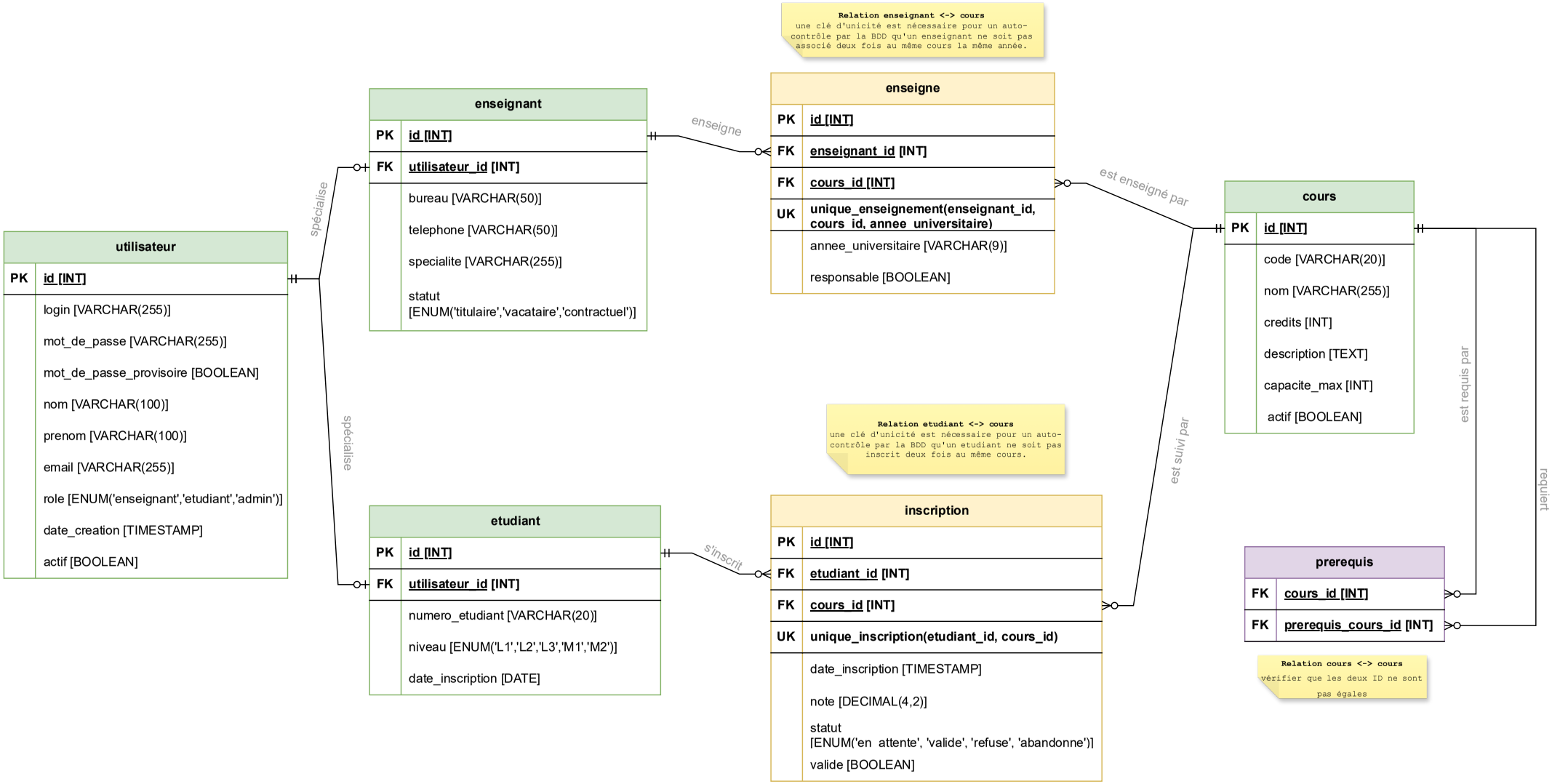
On remarque que plutôt que d'utiliser directement les ID des cours, on utilise l'attribut `code` sur une condition dans une clause qui va nous permettre de retrouver l'ID du cours. Cette méthode est plus robuste car si l'ID d'un cours venait à changer alors on ne perdrait pas la relation de prérequis.

6.3 Cohérences des données

La base de données a été pensée de manière relationnelle, avec une table centrale utilisateur et des tables spécifiques pour les enseignants, les étudiants et les cours avec des systèmes de vérifications propres à une base de données (identifiants unique, repérage des doublons, etc.).

Les opérations sensibles (ajout d'un enseignant ou d'un étudiant) sont réalisées à l'aide de transactions afin de garantir la cohérence des données. C'est la base de données seule qui est responsable de cette cohérence et non le code PHP qui ne fait qu'interpréter les erreurs renvoyées et agir en conséquence. En cas d'erreur, les modifications sont annulées et l'erreur est enregistrée dans les logs.

L'utilisation de clés étrangères permet également de gérer automatiquement certaines suppressions grâce aux mécanismes de cascade.


Relation enseignant <-> cours
une clé d'unicité est nécessaire pour un auto-contrôle par la BDD qu'un enseignant ne soit pas associé deux fois au même cours la même année.
enseignant
PK id [INT]
FK utilisateur_id [INT]
bureau [VARCHAR(50)]
telephone [VARCHAR(50)]
specialite [VARCHAR(255)]
statut [ENUM('titulaire','vacataire','contractuel')]
enseigne
PK id [INT]
FK enseignant_id [INT]
FK cours_id [INT]
UK unique_enseignement(enseignant_id, cours_id, annee_universitaire)
annee_universitaire [VARCHAR(9)]
responsable [BOOLEAN]
est enseigné par
cours
PK id [INT]
code [VARCHAR(20)]
nom [VARCHAR(255)]
credits [INT]
description [TEXT]
capacite_max [INT]
actif [BOOLEAN]
est requis par
requiert
utilisateur
PK id [INT]
login [VARCHAR(255)]
mot_de_passe [VARCHAR(255)]
mot_de_passe_provisoire [BOOLEAN]
nom [VARCHAR(100)]
prenom [VARCHAR(100)]
email [VARCHAR(255)]
role [ENUM('enseignant','etudiant','admin')]
date_creation [TIMESTAMP]
actif [BOOLEAN]
spécialise
spécialise
Relation etudiant <-> cours
une clé d'unicité est nécessaire pour un auto-contrôle par la BDD qu'un etudiant ne soit pas inscrit deux fois au même cours.
etudiant
PK id [INT]
FK utilisateur_id [INT]
numero_etudiant [VARCHAR(20)]
niveau [ENUM('L1','L2','L3','M1','M2')]
date_inscription [DATE]
s'inscrit
inscription
PK id [INT]
FK etudiant_id [INT]
FK cours_id [INT]
UK unique_inscription(etudiant_id, cours_id)
date_inscription [TIMESTAMP]
note [DECIMAL(4,2)]
statut [ENUM('en attente', 'valide', 'refuse', 'abandonne')]
valide [BOOLEAN]
est suivi par
prerequis
FK cours_id [INT]
FK prerequis_cours_id [INT]
Relation cours <-> cours
vérifier que les deux ID ne sont pas égales

7 Codes

7.1 Page unique

La page `accueil.php` charge les éléments communs comme le bandeau et reconnaît le rôle de l'utilisateur connecté.

En fonction de ce rôle, la partie propre à chaque utilisateur est injectée (voir script nº 1) permettant de limiter les répétitions dans le code et de préserver la maintenance.

Script PHP nº 1 : Page `accueil.php` (extrait)

```php
<main style="margin: 2rem;">

  <p>Connecté en tant que : <strong><?= htmlspecialchars($login, ENT_QUOTES, '
      UTF-8') ?></strong></p>

  <h2>Accueil <?= $elements['accueil'] ?></h2>
  <p>
    <?= $elements['message'] ?>
  </p>
  <?php
  switch ($role) {
    case 'admin':
      include('views/admin.php');
      break;
    case 'enseignant':
      include('views/teacher.php');
      break;
    case 'etudiant':
      include('views/student.php');
      break;

    default:
      # code...
      break;
  } ?>
</main>
```

7.2 Chargement des dépendances et structure commune

Les pages reposent sur l'utilisation systématique de `require_once` pour charger les fichiers nécessaires (configuration, fonctions communes, logique métier). Cela permet d'éviter les redéfinitions multiples et garantit que toutes les dépendances sont disponibles avant l'exécution du code.

Script PHP nº 2 : Dépendances

```php
require_once __DIR__ . '/../../config.php';
require_once __DIR__ . '/../../functions.php';
require_once __DIR__ . '/../../classes/universite-db.class.php';
```

7.3 Gestion des sessions et contrôle d'accès

La gestion des sessions est centralisée dans functions.php. Chaque page sensible vérifie que l'utilisateur est connecté et qu'il possède le rôle adéquat avant de continuer.

Script PHP nº 3 : Vérification de la connexion

```php
startSession();

if (!isConnecte()) {
    header('Location: ../index.php');
    exit;
}

if (!isAdmin()) {
    header('Location: ../accueil.php');
    exit;
}
```

Ce mécanisme empêche l'accès direct aux pages via l'URL et renforce la sécurité côté serveur.

7.4 Centralisation de l'accès à la base de données

Toutes les requêtes SQL sont regroupées dans une classe dédiée (`UniversiteDB`), héritant d'une classe générique `Database` permettant la gestion de la connection à la base de données de l'université.

Script PHP nº 4 : Classe `Database`

```php
class DataBase
{
  private $host = DB_HOST;
  private $dbname = DB_NAME;
  private $user = DB_USER;
  private $pwd = DB_PWD;

  private $pdo = null;

  /**
   * Établit et retourne une connexion PDO à la BDD de l'université.
```

```php
     * @return PDO La connexion à la BDD
     * @throws Exception si la connexion échoue.
     */
    protected function connect()
    {
        if ($this->pdo !== null) {
            // la connection à la base de données est déjà établie
            return $this->pdo;
        }

        $dsn = 'mysql:host=' . $this->host . ';dbname=' . $this->dbname . ';charset=utf8';

        try {
            $this->pdo = new PDO($dsn, $this->user, $this->pwd, [
                PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
                PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            ]);
        } catch (PDOException $exception) {
            error_log('[' . date(DATE_RFC2822) . '] Database connection error : ' . $exception->getMessage() . PHP_EOL, 3, ERROR_LOG_PATH);
            throw new Exception("Connection to database failed");
        }
        return $this->pdo;
    }

    /**
     * Ferme la connexion à la BDD de l'université
     */
    protected function disconnect() {
        if ($this->pdo !== null) {
            $this->pdo = null;
        }
    }
}
```

Script PHP nº 5 : Classe UniversiteDB

```php
class UniversiteDB extends DataBase
{
    /**
     * Récupère un utilisateur par son login
     * @param string $login Le login de l'utilisateur
     * @return array|null L'utilisateur trouvé ou null si non
     */
    public function getUserByLogin(string $login): ?array
    {
        $sql = "SELECT id, login, mot_de_passe, role
                FROM utilisateur
                WHERE login = :login";

        $stmt = $this->connect()->prepare($sql);
        $stmt->execute([':login' => $login]);
        $user = $stmt->fetch();
```

```php

    return $user ?: null;
  }

  // Méthodes suivent
}
```

Les pages appellent uniquement des méthodes métier, sans manipuler directement PDO.

Script PHP nº 6 : Exemple de fonction de la classe `UniversiteDB`

```php
public function getAllEnseignants(): array {
    $stmt = $this->connect()->query("SELECT * FROM enseignant_view");
    return $stmt->fetchAll(PDO::FETCH_ASSOC);
}
```

Script PHP nº 7 : Exemple d'utilisation de la classe `UniversiteDB`

```php
$db = new UniversiteDB();
$enseignants = $db->getAllEnseignants();
```

7.5 Utilisation des transactions et gestion des erreurs

Les opérations critiques utilisent des transactions afin de garantir la cohérence des données. En cas d'erreur, la transaction est annulée et l'erreur est enregistrée dans un fichier de log.

Script PHP nº 8 : Exemple de gestion de transaction

```php
$pdo->beginTransaction();

try {
    // Insertion utilisateur
    // Insertion enseignant
    $pdo->commit();
    return true;
} catch (PDOException $e) {
    $pdo->rollBack();
    error_log($e->getMessage(), 3, ERROR_LOG_PATH);
    return false;
}
```

7.6 Validation des formulaires côté serveur

Les formulaires sont validés côté serveur à l'aide de fonctions dédiées. Cela garantit l'intégrité des données, même si les contrôles HTML sont contournés.

Script PHP nº 9 : Exemple de fonction de validation

```php
function validate_email_required(array &$errors, string $field, string $label):
    ?string {
    if (!isset($_POST[$field]) || trim($_POST[$field]) === '') {
        $errors[] = "$label obligatoire.";
        return null;
    }

    if (!filter_var($_POST[$field], FILTER_VALIDATE_EMAIL)) {
        $errors[] = "$label invalide.";
        return null;
    }

    return htmlspecialchars($_POST[$field], ENT_QUOTES, 'UTF-8');
}
```

Script PHP nº 10 : Exemple de vérifications

```php
  $email = validate_email_required($errors, 'email', 'Email');
$nom = validate_required_text($errors, 'nom', 'Nom', 1, 32);
```

7.7 Réutilisabilité grâce aux fonctions de validation

Les fonctions de validation sont génériques et réutilisables dans l'ensemble du projet. Elles sont utilisées aussi bien lors de la création que de la modification des utilisateurs.

Script PHP nº 11 : Exemple de fonction de validation

```php
  function validate_required_text(
    array &$errors,
    string $key,
    string $label,
    int $minLen,
    int $maxLen
): ?string {
    $v = trim($_POST[$key] ?? '');
    if ($v === '') {
        $errors[$key] = "$label obligatoire.";
        return null;
    }
    if (mb_strlen($v) < $minLen || mb_strlen($v) > $maxLen) {
        $errors[$key] = "$label doit contenir entre $minLen et $maxLen caractères
            .";
        return null;
    }
    return htmlspecialchars($v, ENT_QUOTES, 'UTF-8');
}
```

7.8 Système d'actions pour la gestion des pages

La page `accueil.php` utilise un paramètre `action` pour déterminer le traitement à effectuer. Cela permet de centraliser la logique dans un seul fichier par espace utilisateur.

Script PHP nº 12 : Exemple de gestion d'une action utilisateur

```php
$action = $_GET['action'] ?? null;

if ($action === 'liste_enseignants') {
    $enseignants = $db->getAllEnseignants();
}

if ($action === 'edit_enseignant') {
    $enseignantCourant = $db->getEnseignantById((int)$_GET['id']);
}
```

7.9 Séparation logique / affichage

La logique applicative est séparée de l'affichage HTML :

- Les fichiers `*.logic.php` contiennent les traitements PHP sont regroupés dans un dossier `pages/logic` ;
- tandis que les éléments de codes ajoutés à la page `accueil.php` et regroupés dans `pages/views` se concentrent sur l'interface.

Script PHP nº 13 : Exemple de séparation logique/affichage

```php
  // admin.logic.php
$enseignants = $db->getAllEnseignants();

<!-- admin.php -->
<?php foreach ($enseignants as $e): ?>
<tr>
    <td><?= htmlspecialchars($e['nom']) ?></td>
    <td><?= htmlspecialchars($e['prenom']) ?></td>
</tr>
<?php endforeach; ?>
```

7.10 Mécanisme de feedback utilisateur

Un système de feedback basé sur la session permet d'informer l'utilisateur du résultat des actions effectuées.

Script PHP nº 14 : Exemple de feedback

```php
$_SESSION['feedback'] = [
    'message' => 'Utilisateur ajouté avec succès',
    'success' => true
];
```

Script PHP nº 15 : Utilisation du feedback à l'affichage

```php
<?php if (hasFeedbackInSession()): ?>
    <p class="<?= $_SESSION['feedback']['success'] ? 'success' : 'warning' ?>">
        <?= htmlspecialchars($_SESSION['feedback']['message']) ?>
    </p>
<?php unset($_SESSION['feedback']); endif; ?>
```

7.11 Authentification sécurisée côté serveur

Mise en place d'une authentification robuste côté serveur. D'une part, la récupération de l'utilisateur s'appuie sur une requête préparée PDO avec paramètre nommé (`:login`), ce qui empêche les injections SQL en évitant toute concaténation directe de données utilisateur dans la requête. D'autre part, la vérification du mot de passe repose sur `password_verify()`, qui compare un mot de passe saisi à un hash stocké en base, sans jamais manipuler ou stocker le mot de passe en clair.

Script PHP nº 16 : Méthodes de la classe `UniversiteDB` pour l'authentiofication

```php
public function getUserByLogin(string $login): ?array
{
  $sql = "SELECT id, login, mot_de_passe, role
          FROM utilisateur
          WHERE login = :login";

  $stmt = $this->connect()->prepare($sql);
  $stmt->execute([':login' => $login]);
  $user = $stmt->fetch();

  return $user ?: null;
}

public function goodLoginPasswordPair(string $login, string $password): bool
{
  $user = $this->getUserByLogin($login);
  if (!$user) return false;

  if (!password_verify($password, $user['mot_de_passe'])) {
    return false;
  }
  return true;
}
```

7.12 Ajout d'un cours : transaction + gestion des prérequis

Choix important pour garantir l'intégrité de la base de données : l'utilisation de transactions lors d'opérations complexes. L'ajout d'un cours peut entraîner plusieurs insertions (création du cours + insertion des prérequis). En encapsulant l'ensemble dans une transaction (`beginTransaction`, `commit`, `rollBack` – voir exemple nº 17), le système garantit un comportement « tout ou rien » : si un prérequis est invalide (ex. code inexistant), la transaction est annulée et la base reste dans un état cohérent, sans cours partiellement créé.

Script PHP nº 17 : Extraits de la méthode `addCourse` de la classe `UniversiteDB`

```php
$pdo->beginTransaction();

$stmtCours = $pdo->prepare($sqlCours);
$stmtCours->execute([
  ':code' => $code,
  ':nom' => $nom,
  ':credits' => $credits,
  ':desc' => $description,
  ':cap' => $capaciteMax,
  ':annee' => $annee
]);

// Ajout des prérequis (si présents)...
foreach ($prerequisCodes as $prerequisCode) {
  $stmtGetId->execute([':code_prerequis' => $prerequisCode]);
  $prerequisId = $stmtGetId->fetchColumn();
  if ($prerequisId == false) { $succes = false; break; }

  $stmtInsertPrerequis->execute([
    ':cours_id' => $nouveauCoursId,
    ':prerequis_cours_id' => $prerequisId
  ]);
}

if ($succes) {
  $pdo->commit();
} else {
  $pdo->rollBack();
}
```

7.13 Inscription d'un étudiant : contrôle métier "prérequis" + exception explicite

Avant l'inscription, le code vérifie les prérequis via une requête dédiée (`getMissingPrerequisites`). Si des prérequis ne sont pas validés, le système bloque l'opération et déclenche une exception explicite, ce qui permet de remonter un message clair côté interface (feedback).

Cette séparation entre “contrôle métier” (prérequis) et “action technique” (INSERT) améliore la fiabilité et évite que des inscriptions incohérentes soient enregistrées en base.

Script PHP nº 18 : Extrait de la méthode `addEnrollment` de la classe `UniversiteDB`

```php
  $missing = $this->getMissingPrerequisites($etudiantId, $coursId);
if (!empty($missing)) {
  $missingCodes = array_column($missing, 'code');
  $missingCodesStr = implode(', ', $missingCodes);

  throw new Exception('Inscription impossible : manquent la validation de ' .
      $missingCodes);
}

$sql = "INSERT INTO inscription (etudiant_id, cours_id) VALUES (:etudiant_id, :
    cours_id)";
$stmt = $this->connect()->prepare($sql);
$stmt->execute([
  ':etudiant_id' => $etudiantId,
  ':cours_id' => $coursId
]);
```

Troisième partie

Prolongements

Il existe bien évidemment une foules de possibilités à partir d'une base de données comme celle-ci.

Dans l'immédiat, nous aurions voulu avoir le temps d'implémenter ces fonctionnalités :

- Mécanisme de gestion de durée de connexion.
- Vérification sur les champs des formulaires (à perfectionner).
- Gestion des mots de passe provisoires (un attribut dans l'entité `utilisateur` a été prévu à cet égard).
- Les demandes d'inscription (des élèves) et de création de cours (des enseignants) devraient rester en attente de validation de l'admin.
- Gestion des notes.
- Possibilité pour les utilisateurs de changer des données personnelles.
- Système de contrôle des spécialités dans la création d'un cours (nécessiterait une table particulière et la table de liaison afférente).
- Gestion de la responsabilité d'un cours (délégation de la validation de l'inscription d'un étudiant d'un cours à l'enseignant responsable – flag prévu dans l'entité `cours`).
- Possibilité de lister les enseignants animant un cours.

Quatrième partie

Annexe : génération de la base de données

La base utilisée par l'application porte le nom de `universite1`. Afin d'initialiser cette base de données pour cette application, de créer les entités nécessaires et de les alimenter de quelques entrées, il convient d'exécuter les scripts suivants (présents dans l'archive du projet) **en respectant l'ordre** !

1. `universiteDB_access.sql`
2. `universiteDB_tables.sql`
3. `universiteDB_populate.sql`

L'utilisateur administrateur de l'application est `admin`. Il pourra lister les autres utilisateurs pour faire éventuellement des tests avec différents rôles.

Tous les mots de passe sont 123456 par défaut, y compris lors de création de nouvel utilisateur dans la base.