Dossier Technique du projet « Accès campus » - Partie gestion des données (PSW)

Table des matières

I - Situation dans le projet 2
1.1 - Synoptique de la réalisation 2
1.2 - Description de la partie personnelle 2
II – Incrément 1 2
2.1 – Modèle conceptuel de données 2
2.2 – Objectif 3
2.3 – Planification 4
2.4 – Mise en place PSW 4
2.5 – Fonctionnement de l'API 5
2.6 – Routes à implémenter 8
2.7 – Réalisation 8
2.7.1 – Base de l'API 8
2.7.2 – Visualisation 10
2.7.3 – Implémentation et tests 13

GitHub : github.com/RobinMartineau/Acc-s-campus-

I - Situation dans le projet

1.1 - Synoptique de la réalisation

Le projet "Accès Campus" a été réalisé dans le cadre du BTS CIEL. Il a pour objectif de gérer et sécuriser l'accès à un établissement scolaire en utilisant des lecteurs RFID (PEA, BAE), une base de données centralisée et plusieurs services (PGS, PSW, etc.). Le système repose sur une architecture distribuée avec différents équipements interconnectés et une API REST pour la communication entre les services. Chaque membre du groupe a une responsabilité sur une partie de l'infrastructure ou du développement logiciel.

1.2 - Description de la partie personnelle

Dans ce projet, ma mission principale concerne la gestion des données (PSW), incluant la mise en place du serveur, de la base de données PostgreSQL, la conception de l'API en Python avec FastAPI, la création des schémas Pydantic, des modèles SQLAlchemy et l'implémentation des différentes routes nécessaires au fonctionnement du système. J'ai aussi réalisé les tests unitaires et les documents techniques associés. Mon travail s'inscrit dans un ensemble collaboratif avec des échanges réguliers avec les autres membres du projet, notamment pour l'intégration finale.

II – Incrément 1

2.1 – Modèle conceptuel de données

Les données sont divisées en 13 tables (*voir Figure 1*). On a une première série de table qui représente quelque chose de physique : Utilisateur, Badge, Salle, Classe, Salle et Equipement (soi BAE, soi PEA). Ensuite on va avoir les différents emplois du temps (EDTClasse, EDTUtilisateur, EDTSalle), qui ne vont en réalité pas représenter un emploi du temps complet mais plutôt un créneau ou un cours. En raccord avec EDTUtilisateur, on va avoir les retards et absences liés au cours. Enfin on a Autorisation pour les différentes autorisations d'accès qu'un utilisateur peut avoir et Log pour enregistrer tous les mouvements des utilisateurs à chaque fois qu'il badge une BAE ou une PEA.

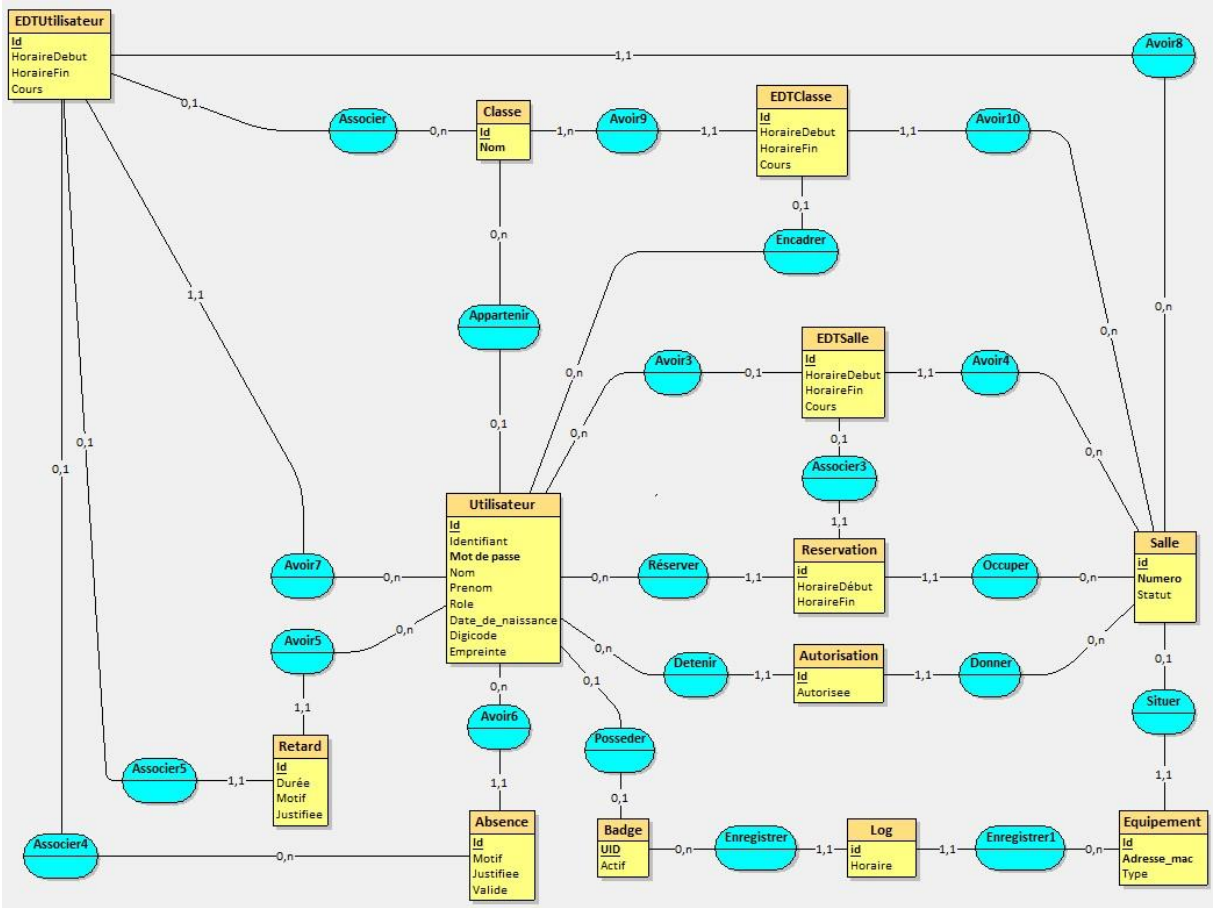


Figure 1 Modèle conceptuel de données

Il m'a fallut plusieurs essaie pour atteindre ce modèle. Les autres versions du MCD sont disponibles sur notre GitHub.

2.2 – Objectif

L'objectif de ce premier incrément sera le même qu'au deuxième et au troisième : permettre l'accès aux données. La différence c'est qu'à chaque incrément mes collègues devront répondre à de nouveaux cas d'utilisation.

Pour ce premier incrément je vais devoir répondre à ces cas d'utilisations :

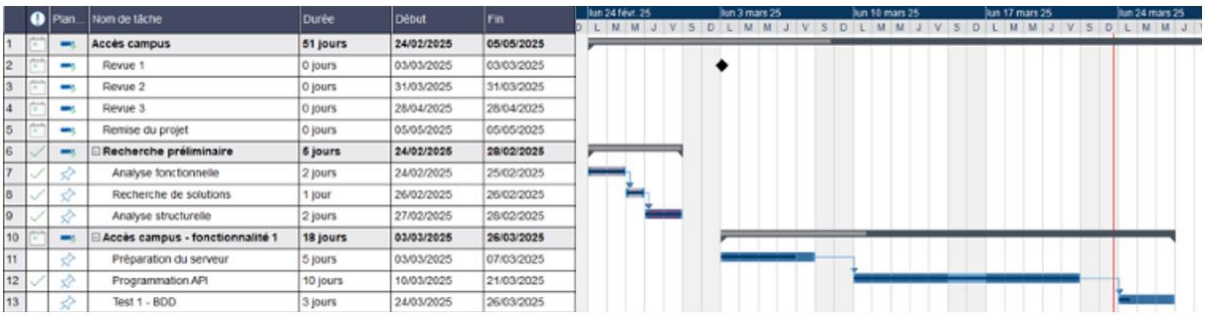
- PEA : Permettre l'accès à l'aide d'un badge RFID et vérifier l'autorisation d'accès.
- BAE : Permettre de signaler l'entrée dans une salle, afficher les informations liées à l'étudiant et mettre à jour les présences et les absences.
- PSW : Permettre de consulter l'historique des absences et des retards.
- PGS : Gérer les badges (Authentifier, créer, supprimer et modifier).

Les diagrammes de cas d'utilisations complets sont disponibles sur notre GitHub.

2.3 – Planification

Pour ce premier incrément, une durée de 4 semaine a été fixé pour toute l'équipe afin de réaliser une intégration au bout de cette durée, juste avant la revue 2.

Pour ma partie, je me suis fixé une semaine pour mettre en place le PSW, puis 2 pour mettre en place l'API. La dernière semaine a été désigné pour réaliser les test unitaires (*voir Figure 2*).



	Plan...	Nom de tâche	Durée	Début	Fin
1		**Accès campus**	**51 jours**	**24/02/2025**	**05/05/2025**
2		Revue 1	0 jours	03/03/2025	03/03/2025
3		Revue 2	0 jours	31/03/2025	31/03/2025
4		Revue 3	0 jours	28/04/2025	28/04/2025
5		Remise du projet	0 jours	05/05/2025	05/05/2025
6		**Recherche préliminaire**	**5 jours**	**24/02/2025**	**28/02/2025**
7		Analyse fonctionnelle	2 jours	24/02/2025	25/02/2025
8		Recherche de solutions	1 jour	26/02/2025	26/02/2025
9		Analyse structurelle	2 jours	27/02/2025	28/02/2025
10		**Accès campus - fonctionnalité 1**	**18 jours**	**03/03/2025**	**26/03/2025**
11		Préparation du serveur	5 jours	03/03/2025	07/03/2025
12		Programmation API	10 jours	10/03/2025	21/03/2025
13		Test 1 - BDD	3 jours	24/03/2025	26/03/2025

Figure 2 Planification Incrément 1

2.4 – Mise en place PSW

Pour cette partie du projet, je ne travaille pas seul : je suis en collaboration avec Thomas Gasche, qui est chargé de réaliser le site web. Nous avons choisi d'utiliser Debian 12 comme système d'exploitation pour sa stabilité, ce qui en fait une solution plus adaptée qu'Ubuntu pour un fonctionnement en continu (24/7). Dès le début, nous avons installé le service SSH afin de pouvoir administrer le serveur à distance. Nous avons également mis en place le pare-feu UFW pour renforcer la sécurité du système.

J'ai ensuite procédé à l'installation de PostgreSQL, puis à la création de la base de données nommée « campus_db ». Celle-ci repose sur deux types ENUM (role et type) et contient 13 tables (voir Figure 3).

```
CREATE TYPE type AS ENUM ('BAE', 'PEA');

CREATE TABLE Equipement(
   Id SMALLSERIAL,
   Adresse_mac CHAR(17) NOT NULL,
   Type type NOT NULL,
   Id_Salle SMALLINT NOT NULL,
   PRIMARY KEY(id),
   UNIQUE(Adresse_mac),
   FOREIGN KEY(Id_Salle) REFERENCES Salle(Id)
);
```

Figure 3 Exemple de création d'un type ENUM et d'une table en SQL

Le script SQL complet permettant de générer l'ensemble de la base de données est disponible sur notre GitHub.

Afin de préparer l'environnement de développement, j'ai installé venv pour créer un environnement virtuel, indispensable sous Debian 12 où pip n'est pas disponible par défaut. J'ai

ensuite installé toutes les bibliothèques Python nécessaires au bon fonctionnement de l'API, ainsi que l'éditeur de texte micro.

Enfin, pour l'environnement de production, je n'utilise pas directement le PSW mais une machine virtuelle sur laquelle j'ai reproduit exactement la même installation.

Un guide d'installation et d'utilisation du PSW est disponible sur notre GitHub.

2.5 – Fonctionnement de l'API

L'API utilise les bibliothèques spécifiques suivantes (au-delà des basiques) :

- FastAPI, pour le fonctionnement général et la création des différentes routes.
- SQLAlchemy, pour l'enregistrement des données dans la base de données.
- Pydantic, pour la vérification de la réception des bons types de données lors de l'appel d'une requête http.

Pour l'enregistrement des données avec SQLAlchemy, je créer une classe par table dans ma base de données. Le diagramme de classe associer est celui de la *Figure 4*. C'est une version simplifiée par soucis de clarté, mais toute les classes hérite de Base, une classe de la bibliothèque SQLAlchemy. Elles sont aussi composées de la classe Column (toujours de SQLAlchemy) qui permet de créer un attribut intégrable dans la base de données. Une version du diagramme plus détaillée, avec notamment la classe Base et la classe Column, est disponible sur notre GitHub.

Enfin pour la vérification des données avec Pydantic, le nombre de classe varie en fonction du nombre de route disponible sur mon API, avec 0 à 1 classe par route. Le diagramme de classe associer est celui de la *Figure 5*. Chaque classe contient les attributs que l'on reçoit dans la requête. Une version complète de ce diagramme de classe est disponible sur notre GitHub.

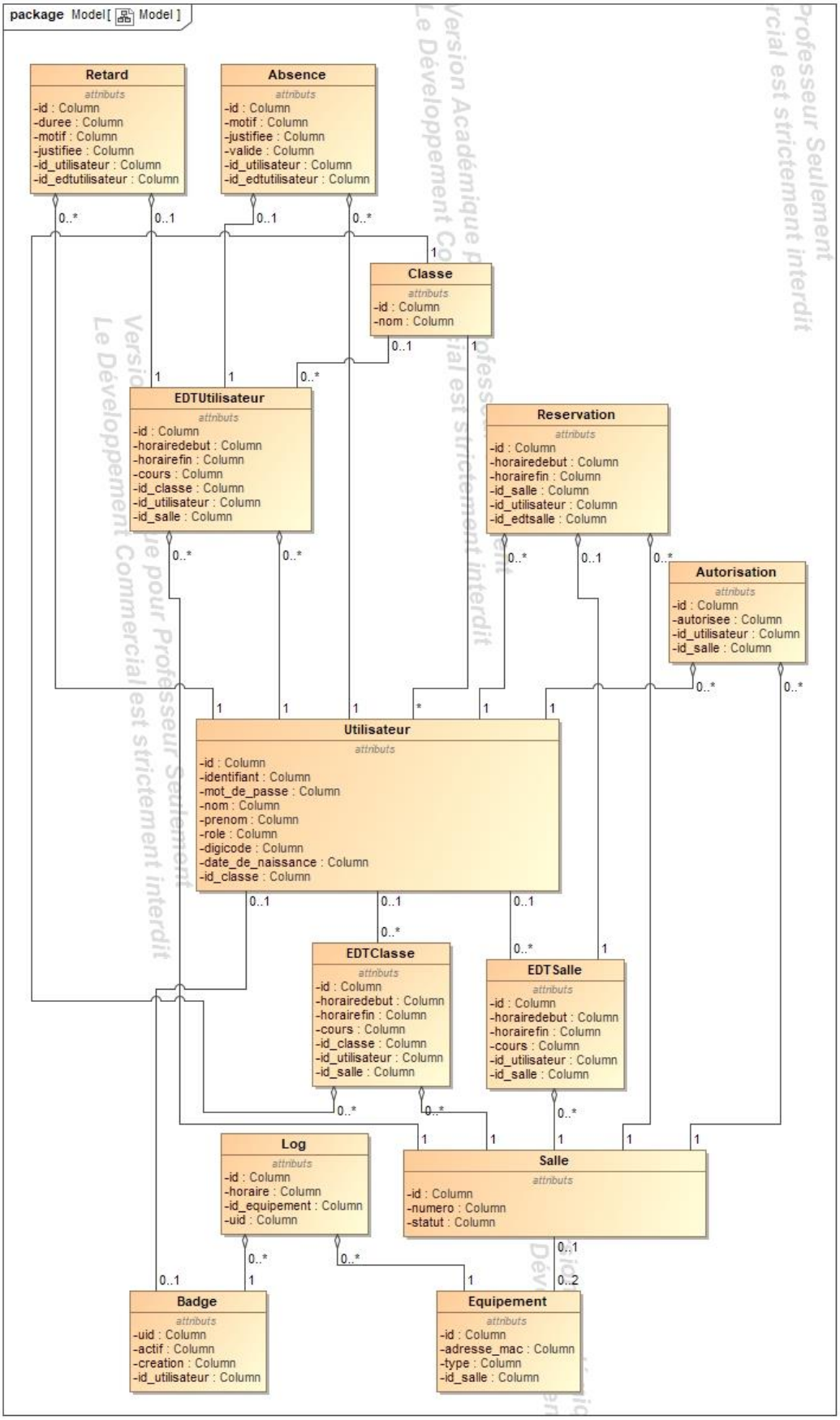


Figure 4 Diagramme de classe (simplifié) relatif à SQLAlchemy

GitHub : github.com/RobinMartineau/Acc-s-campus-

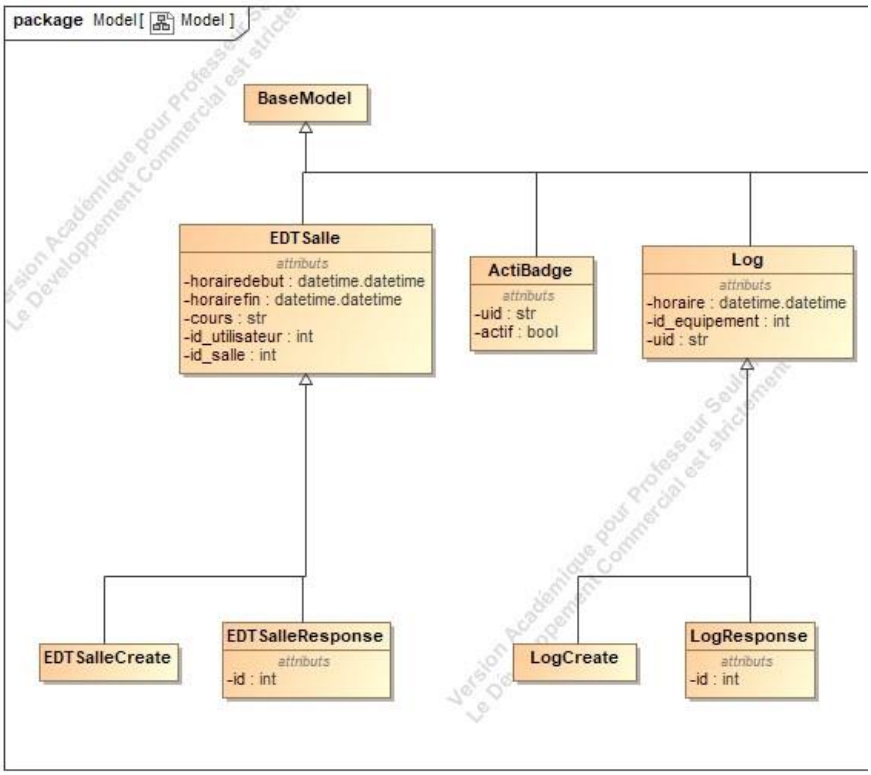


Figure 5 Diagramme de classe (partielle) relatif à Pydantic

L'organisation des fichiers est la suivante :

/API
- main.py
- database.py
- schemas.py
- models.py

/routes
- pea.py
- bae.py
- pgs.py
- psw.py
- retard.py
- etc...

main.py : récupère toutes les routes et lance l'API, c'est le centre de notre API.

database.py : initie la connexion avec la base de données.

schemas.py : contient la déclaration de toutes les classes Pydantic.

models.py : contient la déclaration de toutes les classes SQLAlchemy.

pea.py, bae.py, ... : contient les routes.

2.6 – Routes à implémenter

Voici toutes les routes que je dois implémenter pour répondre aux attentes de l'incrément 1 :

Pour la PEA :

- Récupérer l'UID du badge, l'adresse mac de la PEA et vérifier si l'utilisateur est autorisé à rentrer. S'il n'est pas autorisé je renvoie un message d'erreur, sinon je renvoie son nom, son prénom, son rôle et son autorisation → Requête POST.

Pour la BAE :

- Récupérer l'UID du badge, l'adresse mac de la BAE et enregistre sa présence au cours, ainsi que rajouter un retard si besoins. Je renvoie son nom, son prénom et sa classe. S'il n'est pas dans la bonne classe je lui renvoie la classe ou il doit être → Requête POST.

Pour le PSW (site) :

- Récupérer l'identifiant et le mot de passe d'un utilisateur, vérifier si ça correspond bien et renvoyer s'il est autorisé à se connecter au site web, ainsi que son nom, son prénom et son id → Requête POST.
- Récupérer l'id de l'utilisateur et renvoyer tous ses retards et absences s'il y en a → Requête GET.
- Envoyer tous les élèves de la base de données → Requête GET.

Pour le PGS :

- Récupérer l'UID d'un badge et créer un nouveau badge dans la base de données → Requête POST.
- Récupérer les modifications sur les attributs d'un badge et les appliquer dans la base de données → Requête PUT.
- Envoyer tous les utilisateurs présents dans la base de données → Requête GET.
- Récupérer un UID et un id utilisateur, modifier le badge pour l'associer à l'utilisateur dans la base de données → Requête PUT.
- Recevoir l'UID d'un badge et le supprimer de la base de données → Requête DELETE.

2.7 – Réalisation

2.7.1 – Base de l'API

On commence par le fichier database.py (*voir Figure 6*). Comme dis précédemment il sert de création d'une session local pour communiqué avec la base de données.

```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# URL de connexion PostgreSQL
DATABASE_URL = "postgresql://postgres:BTS2425@localhost/campus_db"

# Création de l'engine SQLAlchemy
engine = create_engine(DATABASE_URL)

# Création d'une session locale
SessionLocal = sessionmaker(bind=engine, autoflush=False, autocommit=False)

# Base pour les modèles SQLAlchemy
Base = declarative_base()
```

Figure 6 Contenu du fichier database.py

Ensuite on créé les fichiers models.py (*voir Figure 7*) et schemas.py (*voir Figure 8*). Le contenue intégrale de ces fichiers est disponible sur notre GitHub.

Pour models.py, j'implémente chaque classe lier aux tables de ma base de données. « relationship() » sert à lier certaine classe entre elle avec les ForeignKey. Les paramètre de Column tel que « nullable » ou bien encore « unique » servent à soi savoir si le paramètre peut être vide, ou bien alors à savoir si deux entrées de la même table peuvent avoir la même valeur. On peut aussi décider d'une valeur par défaut pour un attribut avec « attribut ».

Pour schemas.py je créé des classes pour vérifier que les données que l'API reçoit correspondent. En l'occurrence pour les requêtes spécifiques je créé une classe, tel que pour l'accès à une salle via une PEA.

```python
from sqlalchemy import Column, Integer, String, Boolean, SmallInteger, Date, DateTime, Enum, ForeignKey
from sqlalchemy.orm import relationship
from database import Base
import datetime

class Classe(Base):
    __tablename__ = "classe"

    id = Column(SmallInteger, primary_key = True, autoincrement = True, index = True)
    nom = Column(String(20), unique = True, nullable = False)

class Utilisateur(Base):
    __tablename__ = "utilisateur"

    id = Column(Integer, primary_key = True, autoincrement = True, index = True)
    identifiant = Column(String(61), nullable = False, unique = True)
    mot_de_passe = Column(String(150), nullable = True, unique = True)
    nom = Column(String(30), nullable = True)
    prenom = Column(String(30), nullable = False)
    role = Column(Enum('Invite', 'Personnel', 'Eleve', 'Prof', 'Admin', name = "role_enum"), nullable = False)
    digicode = Column(String(4), nullable = True)
    date_de_naissance = Column(Date, nullable = True)
    id_classe = Column(SmallInteger, ForeignKey("classe.id"), nullable = True)

    classe = relationship("Classe")

class Badge(Base):
    __tablename__ = "badge"

    uid = Column(String(8), primary_key = True, index = True)
    actif = Column(Boolean, nullable = False, default = False)
    creation = Column(Date, nullable = False, default = datetime.date.today)
    id_utilisateur = Column(Integer, ForeignKey("utilisateur.id"), unique = True, nullable = True)

    utilisateur = relationship("Utilisateur")
```

Figure 7 1Contenue partiel du fichier models.py

```python
from pydantic import BaseModel
from typing import Optional
from enum import Enum
import datetime
import chiffrement

#Création des types ENUM
class RoleEnum(str, Enum):
    Invite = "Invite"
    Personnel = "Personnel"
    Eleve = "Eleve"
    Prof = "Prof"
    Admin = "Admin"

class TypeEquipementEnum(str, Enum):
    BAE = "BAE"
    PEA = "PEA"

#Modèle pour la PEA
class AccesRequest(BaseModel):
    uid: str
    adresse_mac: str


#Modèle pour la BAE
class AppelRequest(BaseModel):
    uid: str
    adresse_mac: str


#Modèle pour le PSW
class LoginRequest(BaseModel):
    identifiant: str
    mot_de_passe: str

```

Figure 8 Contenue partiel du fichier schemas.py

2.7.2 – Visualisation

Pour la création d'une route, on va se pencher sur une en particulière : celle de la PEA. Le processus reste le même pour les autres routes et les fichiers et diagrammes liés aux autres routes sont disponible sur notre GitHub.

Tout d'abord pour comprendre comment va fonctionner cette route je m'appuie sur un digramme de séquence fais au préalable (*voir Figure 9*).

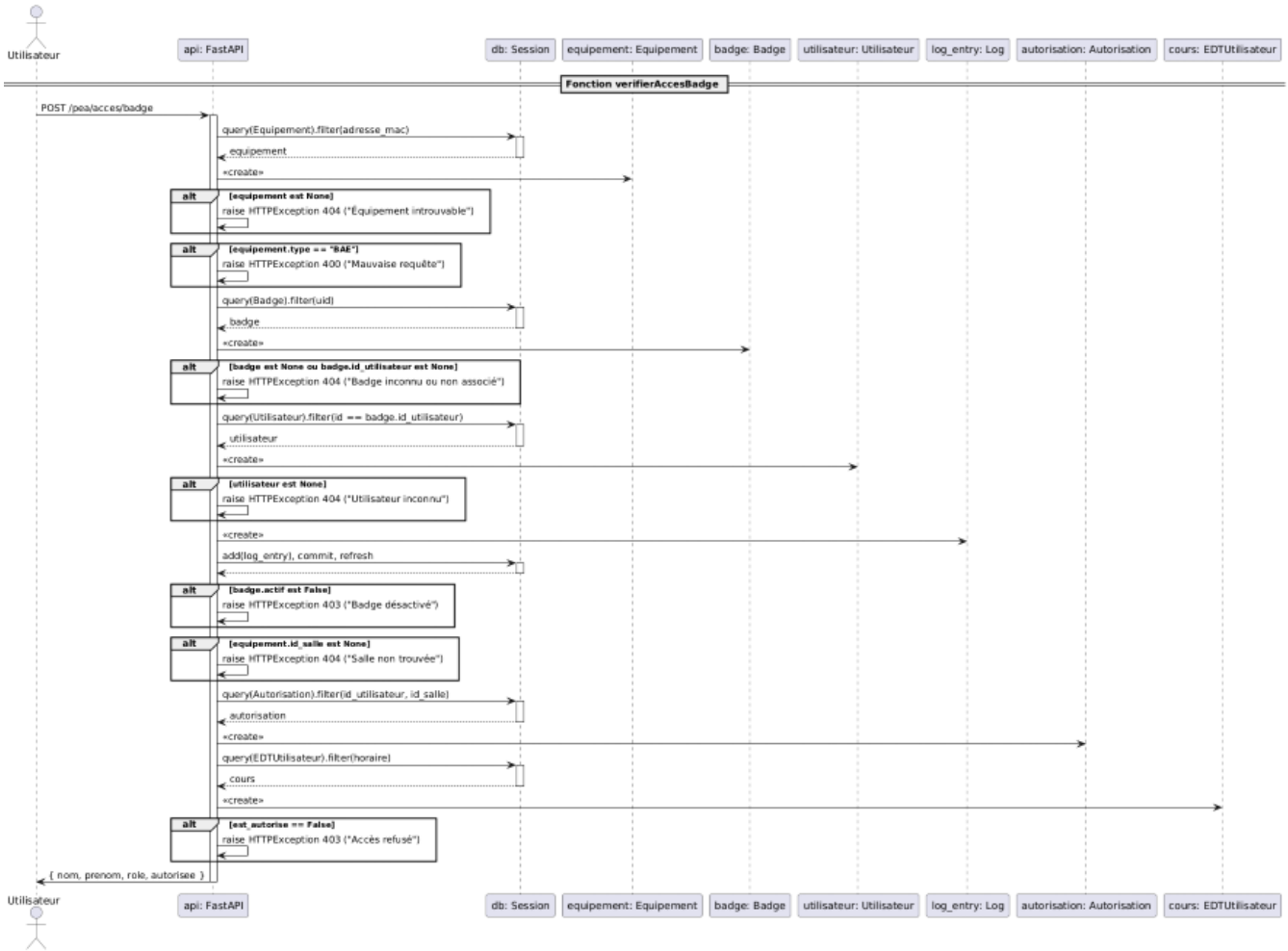


Figure 92 Diagramme de séquence pour l'accès à une salle

L'API va donc recevoir l'UID d'un badge ainsi que l'adresse mac de la PEA, ensuite elle va interroger la BDD (base de données) pour récupérer la PEA correspondant à l'adresse mac, vérifier que c'est bien une PEA et pas une BAE, récupérer le badge puis l'utilisateur lié au badge. Ensuite elle va vérifier si le badge est bien actif sur le campus et récupérer la salle où est installé la PEA. Pour vérifier l'autorisation l'API récupère l'autorisation correspondant à l'utilisateur et à la salle (s'il y en a une), ainsi que le possible cours de l'utilisateur à cette heure-ci. Ainsi l'API détermine si l'utilisateur est autorisé à rentré ou non et renvoie le nom, prénom, rôle et l'autorisation de l'utilisateur si oui.

Précision, à chaque fois que l'api récupère une entrée dans la base de données, une vérification est faite pour vérifier qu'une donnée est bien trouvée, sinon une erreur se lève.

Ensuite à partir de ce diagramme on réalise le plan de test :

Plan de test – Fonction verifierAccesBadge

1 - Identification du test

Nom : Test d'accès via badge RFID
Numéro : T1

2 - Référence du module testé

routes/pea.py – Fonction verifierAccesBadge

3 - Objectif du test

Valider le comportement de la fonction verifierAcces dans tous les cas possibles : accès autorisé, refusé ou erreurs, selon les règles métiers.

4 - Procédure du test

- **Initialisation** : Préparer la BDD de test selon le scénario
- **Lancement** : Envoyer une requête POST à la route /pea/acces/badge avec un corps JSON comme :

```
{
  "uid": "04A3BC1D",
  "adresse_mac": "AA:BB:CC:DD:EE:FF"
}
```

- **Observation** : Comparaison du code de réponse HTTP et du contenu avec les attentes

5 - Résultats attendus

N° Test	Condition	BDD Préparée	Résultat attendu	Statut attendu
1	MAC inconnue	Aucun équipement avec cette MAC	404 Équipement introuvable	Erreur
2	Equipement = BAE	MAC valide, type = 'BAE'	400 Mauvaise requête	Erreur
3	Badge inconnu	Aucun badge avec l'UID fourni	404 Badge inconnu ou non associé	Erreur
4	Badge non lié à utilisateur	Badge trouvé, id_utilisateur = NULL	404 Badge inconnu ou non associé	Erreur
5	Utilisateur inexistant	Badge avec id_utilisateur invalide	404 Utilisateur inconnu	Erreur
6	Badge désactivé	Badge trouvé, actif = False	403 Badge désactivé	Erreur

N° Test	Condition	BDD Préparée	Résultat attendu	Statut attendu
7	Equipement sans salle	Equipement trouvé, id_salle = NULL	404 Salle non trouvée	Erreur
8	Pas d'autorisation ni de cours	Aucun enregistrement dans Autorisation ou EDT	403 Accès refusé	Erreur
9	Autorisation non autorisée	Autorisation trouvée, autorisee = False	403 Accès refusé	Erreur
10	Autorisation autorisée	Autorisation trouvée, autorisee = True	Retour infos utilisateur + autorisee=True	Succès
11	Pas d'autorisation mais cours actif	Aucun enregistrement Autorisation mais cours dans EDT	Retour infos utilisateur + autorisee=True	Succès

6 - Moyens à mettre en œuvre

- **Logiciels** : FastAPI.
- **Matériel** : Poste de développement, base de données locale.
- **Préconditions** : Données de test insérées dans les tables : Equipement, Badge, Utilisateur, Autorisation, EDTUtilisateur.

2.7.3 – Implémentation et tests

Enfin on va pouvoir commencer à code, pour ce on suit à la lettre le diagramme de séquence. Pour les erreurs j'utilise « raise HTTPException » qui stop l'exécution de la fonction et renvoie une erreur http (*voir Figure 10*).

```python
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from database import SessionLocal
import schemas
import models
import datetime

#Instanciation d'un router FastAPI
router = APIRouter()

#Fonction pour récupérer la session de la BDD
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

#Route POST pour vérifier l'accès d'un utilisateur
@router.post("/pea/acces/")
def verifierAcces(request: schemas.AccesRequest, db: Session = Depends(get_db)):
    uid = request.uid
    adresse_mac = request.adresse_mac

    equipement = db.query(models.Equipement).filter(models.Equipement.adresse_mac == adresse_mac).first()
    heure_actuelle = datetime.datetime.now()

    #Vérifier que l'equipement existe
    if not equipement:
        raise HTTPException(status_code = 404, detail = "Équipement introuvable")

    #Vérifier que l'adresse mac correspond bien à une PEA
    if equipement.type == "BAE":
        raise HTTPException(status_code = 400, detail = "Mauvaise requête: contacter un administrateur réseau")

    #Trouver l'utilisateur lié au badge
    badge = db.query(models.Badge).filter(models.Badge.uid == uid).first()

    if not badge or not badge.id_utilisateur:
        raise HTTPException(status_code = 404, detail = "Badge inconnu ou non associé")

    #Vérifier que l'utilisateur existe bel et bien
    utilisateur = db.query(models.Utilisateur).filter(models.Utilisateur.id == badge.id_utilisateur).first()

    if not utilisateur:
        raise HTTPException(status_code = 404, detail = "Utilisateur inconnu")

    #Vérifier si le badge est désactivé
    if not badge.actif:
        raise HTTPException(status_code = 403, detail = "Accès refusé : Veuillez rapporter le badge à un membre de la vie scolaire.")

    #Trouver la salle correspondant à la PEA
    if not equipement or not equipement.id_salle:
        raise HTTPException(status_code = 404, detail = "Salle non trouvée")

    id_salle = equipement.id_salle

    #Vérifier si une autorisation existe pour cet utilisateur dans cette salle
    autorisation = db.query(models.Autorisation).filter(
        models.Autorisation.id_utilisateur == utilisateur.id,
        models.Autorisation.id_salle == id_salle
    ).first()

    #Vérifier s'il a un cours en ce moment dans EDTUtilisateur
    cours = db.query(models.EDTUtilisateur).filter(
        models.EDTUtilisateur.id_utilisateur == utilisateur.id,
        models.EDTUtilisateur.id_salle == id_salle,
        models.EDTUtilisateur.horairedebut <= heure_actuelle,
        models.EDTUtilisateur.horairefin >= heure_actuelle
    ).first()

    #Déterminer si l'utilisateur est autorisé
    if autorisation:
        est_autorise = autorisation.autorisee
    else:
        est_autorise = bool(cours)

    #Refus de l'accès si l'utilisateur n'est pas autorisé
    if not est_autorise:
        raise HTTPException(status_code = 403, detail = "Accès refusé")

    return {
        "nom": utilisateur.nom,
        "prenom": utilisateur.prenom,
        "role": utilisateur.role,
        "autorisee": est_autorise
    }
```

Figure 10 Contenue partiel du fichier pea.py

Il ne me reste plus qu'a réaliser mes tests unitaires sur ce que je viens de coder à l'aide du plan de test réalisé au préalable et d'effectuer des modifications si nécessaire. La *Figure 11* présente le procès-verbal du plan de test sur la fonction verifierAccesBadge.

verifierAccesBadge		
Date du test:	**12/03/2025**	
Type du test:	**Fonctionnel**	
Objectif du test:	**Vérifier le bon fonctionnement de la requête d'ouverture d'une porte.**	
Condition du test		
Etat initial	**Environnement du test**	
API en marche	**Le test est réalisé sur une machine Debian 12 à l'aide du fichier pytest verifierAccesBadge.py.**	
Procédure du test		
Opération	**Résultats attendus**	**Résultats obtenus**
Test n°1 - Adresse MAC inexistante	**404 Équipement introuvable**	**404 Équipement introuvable**
Test n°2 - Adresse MAC d'une BAE	**400 Mauvaise requête**	**400 Mauvaise requête**
Test n°3 - UID badge inconnu	**404 Badge inconnu ou non associé**	**404 Badge inconnu ou non associé**
Test n°4 - Badge non lié à utilisateur	**404 Équipement introuvable**	**404 Badge inconnu ou non associé**
Test n°5 - Utilisateur inexistant	**404 Utilisateur inconnu**	**404 Utilisateur inconnu**
Test n°6 - Badge désactivé	**403 Badge désactivé**	**403 Badge désactivé**
Test n°7 - Equipement sans salle	**404 Salle non trouvée**	**404 Salle non trouvée**
Test n°8 - Pas d'autorisation ni de cours	**403 Accès refusé**	**403 Accès refusé**
Test n°9 - Autorisation non autorisée	**403 Accès refusé**	**403 Accès refusé**
Test n°10 - Autorisation autorisée	**Accès autorisé + infos utilisateur**	**Accès autorisé + infos utilisateur**
Test n°11 - Pas d'autorisation mais cours actif	**Accès autorisé + infos utilisateur**	**Accès autorisé + infos utilisateur**
Nature des modifications apportées au fichier source du module testé.		
Aucune		

Figure 11 Procès-verbal du test de la fonction verifierAccesBadge

/ ! \ RAPPEL : Le même résonnement et fonctionnement est appliqué pour les autres routes, par soucis de place je ne peux pas toutes les affichées ici mais elles restent disponibles sur notre GitHub. / ! \