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

Table des matières

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

I – Présentation

1.1 - Contexte et objectifs

Le projet Campus Accès vise à améliorer la gestion et la sécurité des accès aux salles du Campus Saint Aubin La Salle. Grâce à un système automatisé basé sur des badges RFID, les étudiants et le personnel peuvent accéder aux salles de manière sécurisée tout en enregistrant leurs entrées et sorties. L'objectif est d'assurer un contrôle centralisé des accès, optimisé grâce au Poste Serveur Web (PSW).

Figure 1 Campus Saint Aubin La Salle

Ma tâche personnelle dans ce projet de grande envergure est de centralisé les données et de permettre l'accès à ces données (*voir Figure 2*).

Étudiant 5 EC ☐ IR ☒	*Liste des fonctions assurées par l'étudiant* Poste Serveur Web : • Gérer le stockage de données • Permettre l'accès aux données • Permettre de consulter l'historique des absences et des retards (mode personnel)	Installation : • Le lecteur NFC ACR122U Mise en œuvre : • Le système d'exploitation du PSW, la base de données, l'environnement de développement Configuration : • L'infrastructure réseau externe Réalisation : • Réalisation des fonctionnalités en charge. Documentation : • Le dossier technique et les documents relatifs au module. Un guide de mise en route et d'utilisation du module

Figure 2 Mission étudiant 5

1.2 - Description du système

Le Poste Serveur Web (PSW) est au cœur du système, assurant la gestion centralisée des accès. Il héberge la base de données PostgreSQL, qui stocke les informations des utilisateurs, des badges et des historiques d'accès. L'API, permet d'interagir avec la base. Les Poignées Électroniques Autonomes (PEA) communiquent avec le PSW pour vérifier les accès, les Bornes d'Appel Etudiant automatise l'appel dans un cours, tandis que le Poste de Gestion des Salles (PGS) permet d'administrer les réservations et les droits d'accès.

La communication avec l'API se fait grâce à différentes requêtes http soi GET, POST, PUT ou DELETE. L'API, elle, interroge la base de données pour ensuite renvoyer des réponses au format JSON (*voir Figure 3*).

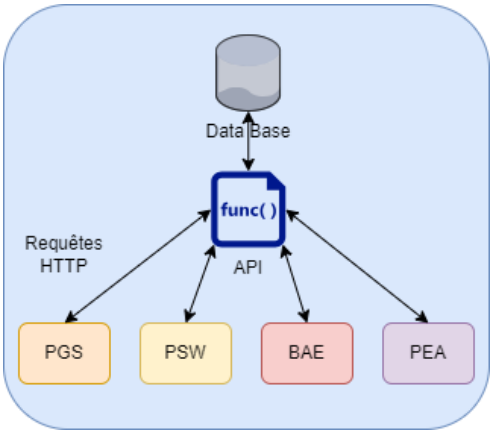


Figure 3 Communication entre les différents acteurs

1.3 - Infrastructure réseau

Notre infrastructure réseau (*voir figure 4*) est un seul et unique grand réseau d'adresse 192.168.248.0/21, ce que nous offre la possibilité de configurer 2046 hôtes différends. Le réseau est divisé par 3 VLANs :

- Publique (VLAN 10), pour tous les PCs du campus accessible pour tout type d'utilisateur. Un serveur DHCP est dédié pour ce VLAN qui peut attribuer 1583 adresses différentes.
- Badge (VLAN 20), pour toutes les BAEs et PEAs. Le serveur DHCP peut attribuer jusqu'à 666 adresses.
- Administration (VLAN 30), pour tous les serveurs et poste d'administration. Les adresses IP sont fixe pour les différends serveurs ou bien pour le poste de gestion des salles. Les autres adresses sont attribuées par le PSW.

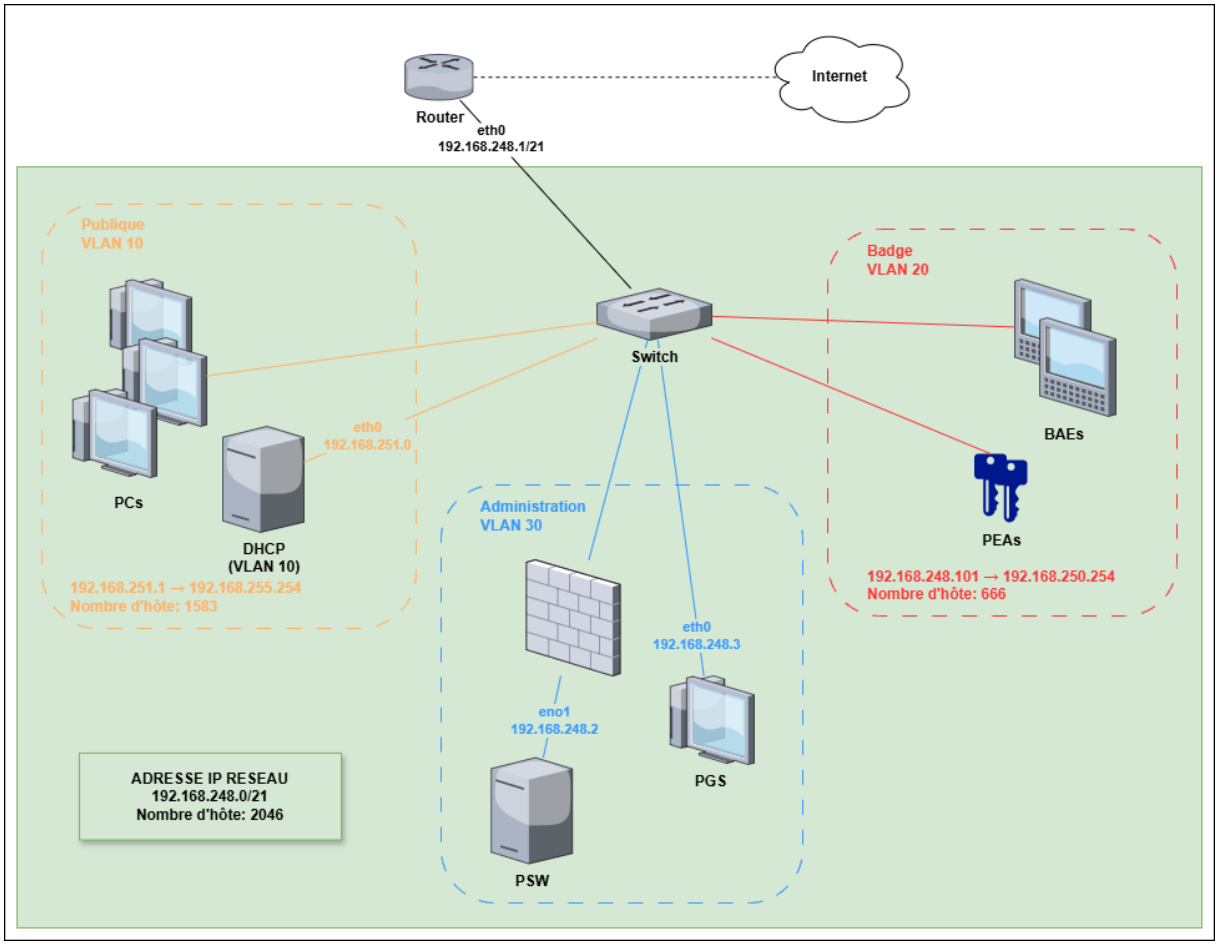


Figure 4 Infrastructure réseau

Bien sur un pare-feu est présent sur le PSW, il bloque tous les ports et n'ouvre que ceux nécessaire. Les suivants donc : 22 (SSH), 53 (DNS), 80 (HTTP), 443 (HTTPS), 3000 (Node.js), 5432 (pgAdmin), 8000 (uvicorn).

Des manuels d'installation sont dispo sur notre GitHub.

1.4 – Modèle conceptuel de données

Les données sont divisées en 13 tables (*voir Figure 5*). 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.

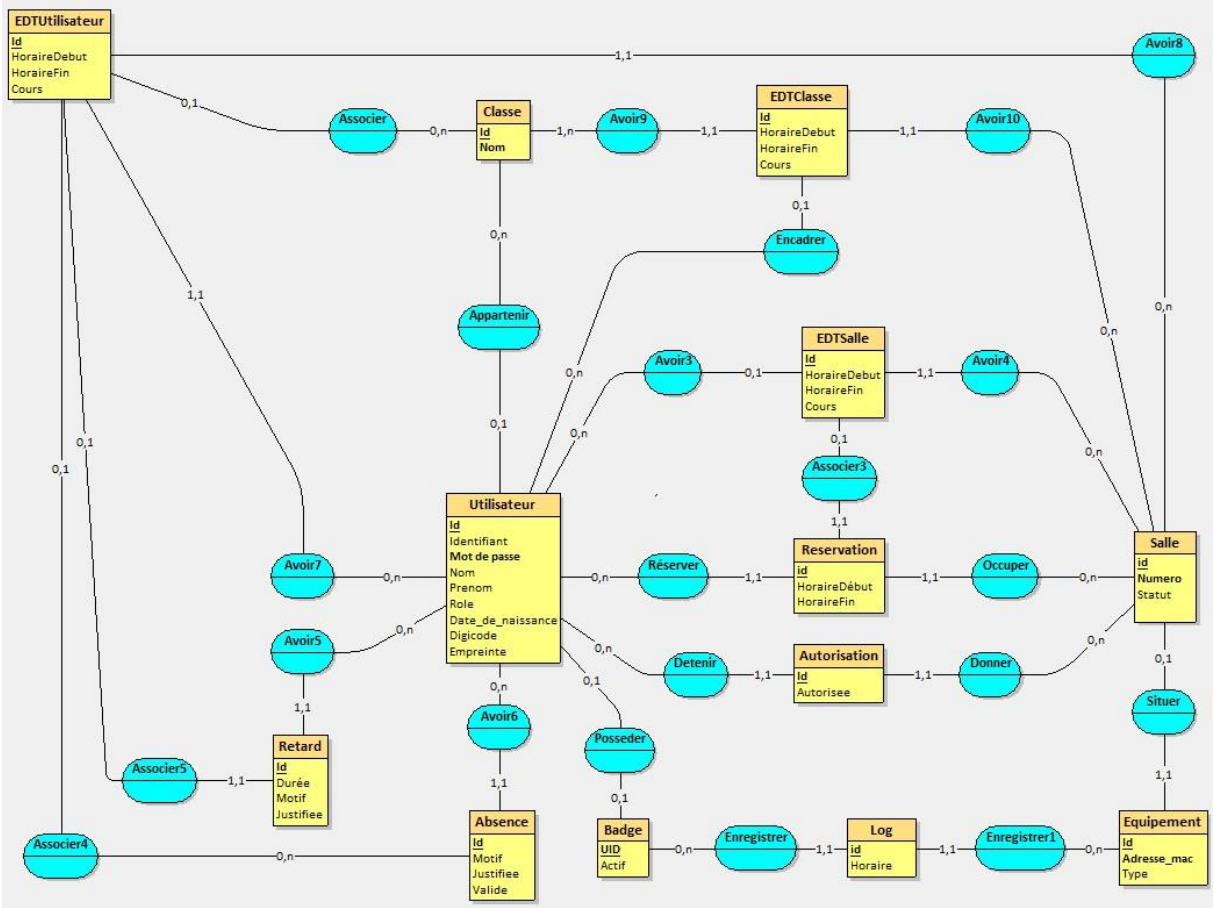


Figure 5 Modèle conceptuel de données

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

II – Incrément 1

2.1 – 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 :

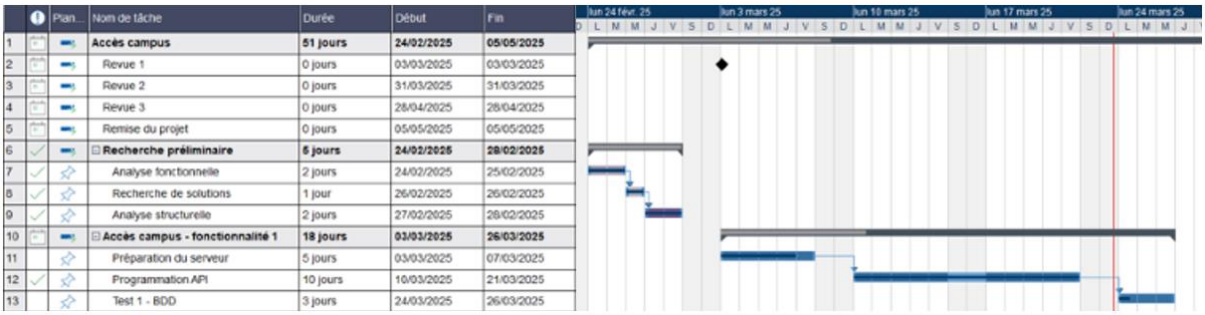
- PEA : Permettre l'accès à l'aide d'un badge RFID et vérifier l'autorisation d'accès.
- BAE : Permettre de signaler l'entrée dans une salle, afficher les informations liées à l'étudiant et mettre à jour les présences et les absences.
- PSW : Permettre de consulter l'historique des absences et des retards.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

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

2.2 – Planification

Pour ce premier incrément, une durée de 4 semaine à 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 6*).



	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 6 Planification Incrément 1

2.3 – Mise en place PSW

Pour cette partie je ne suis pas seul à travailler dessus, je travaille avec Thomas Gasche qui lui doit réaliser un site web.

On a choisi de partir sur Debian 12 comme OS, pour sa stabilité. C'est plus adéquat que Ubuntu pour de 24/7. On installe ssh dès le début pour ensuite pouvoir travailler à distance dessus. Ensuite on met en place le pare-feu ufw.

J'installe PostgreSQL puis je créé ma base de données « campus_db ». J'ai créé deux type ENUM (role, et type) et mes 13 tables (*exemple Figure 7*).

```
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 7 Exemple de création d'un type ENUM et d'une table en SQL

Le script SQL pour créer toute la base de données est disponible sur notre GitHub.

Ensuite j'ai installé venv pour me créer un environnement virtuel car sinon pip n'est pas disponible sur Debian 12. Je finis par installer toute les bibliothèques pythons dont j'ai besoins et micro, un éditeur de texte.

Pour la production je ne vais pas travailler sur le PSW directement mais plutôt sur une machine virtuelle. Je fais la même installation dessus.

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

2.4 – 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 9*. 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 8*. 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.

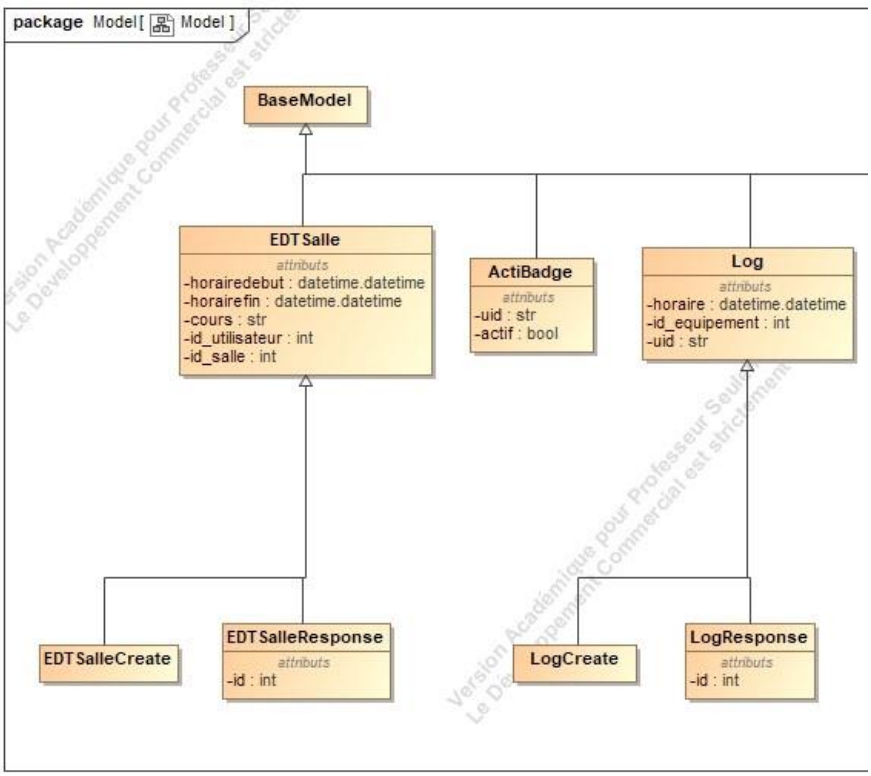


Figure 8 Diagramme de classe (partielle) relatif à Pydantic

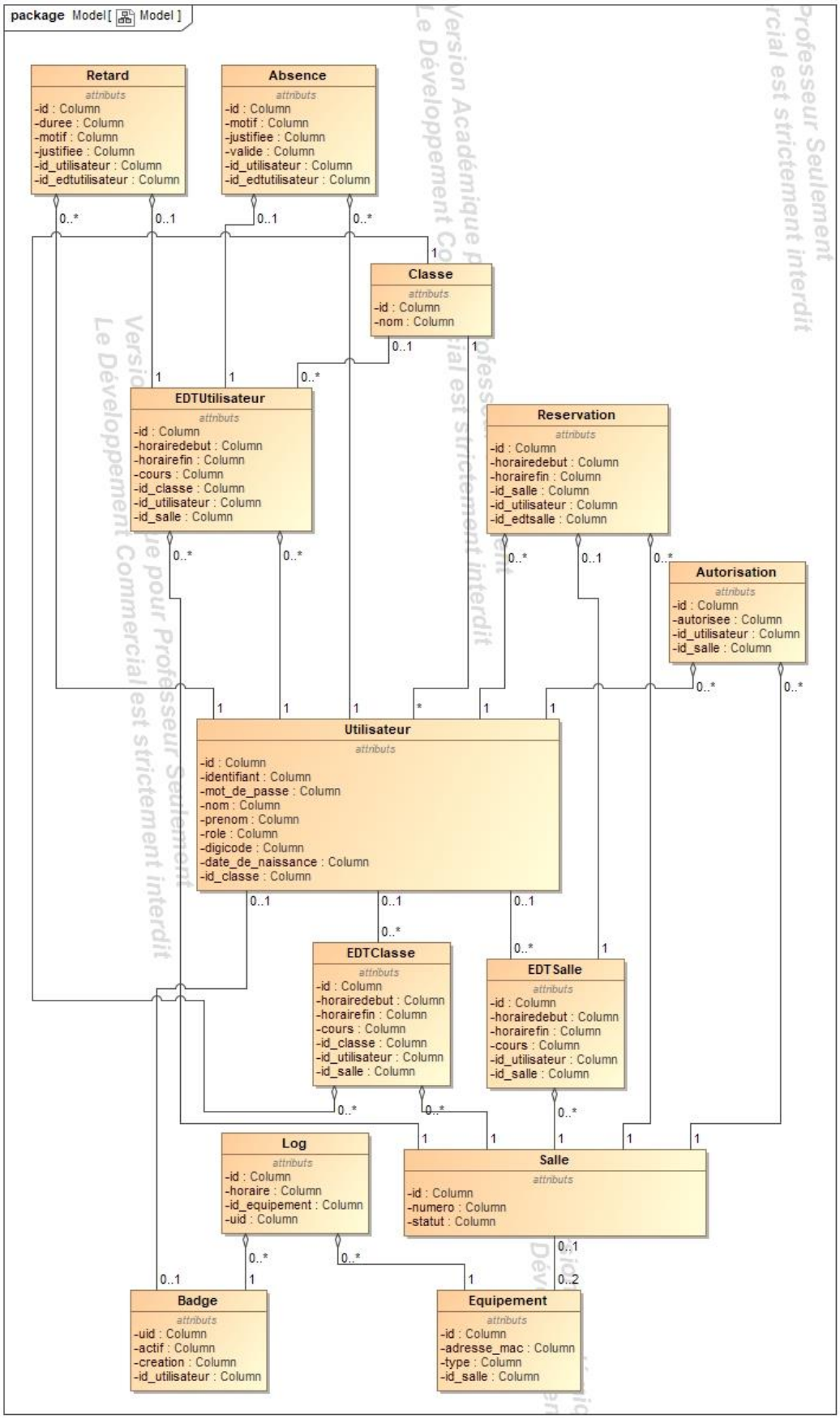


Figure 9 Diagramme de classe (simplifié) relatif à SQLAlchemy

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

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.

2.5 – 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.

2.6 – Réalisation

On commence par le fichier database.py (*voir Figure 10*). 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 10 Contenu du fichier database.py

Ensuite on créé les fichiers models.py (*voir Figure 11*) et schemas.py (*voir Figure 12*). 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 11Contenue 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 12 Contenue partiel du fichier schemas.py

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 13*).

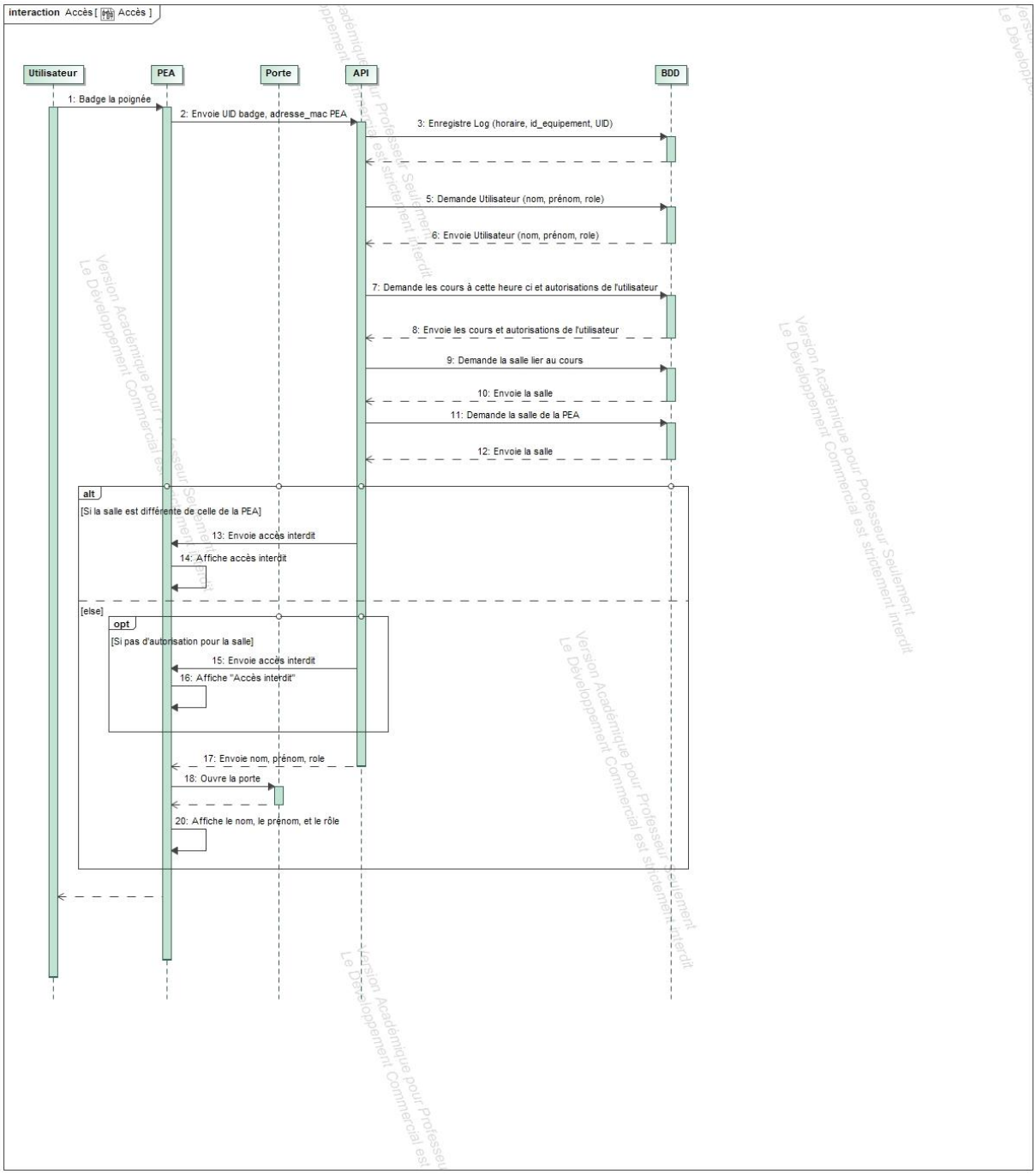


Figure 13 Diagramme de séquence pour l'accès à une salle

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 (*voir Figure 14*).

PEA		
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 de commandes curl.	
Procédure du test		
Opération	Résultats attendus	Résultats obtenus
Test n°1 - Badge acitf, autorisation	Accès autorisé	-
Test n°2 - Badge actif, cours en cours	Accès autorisé	-
Test n°3 - Adresse mac d'une BAE	Accès interdit	-
Test n°4 - Adresse mac inéxistente	Accès interdit	-
Test n°5 - UID badge inconnu	Accès interdit	-
Test n°6 - Badge désactivé	Accès interdit	-
Test n°7 - Pas d'autorisation, ni de cours	Accès interdit	-
Nature des modifications apportées au fichier source du module testé.		
-		

Figure 14 Plan de test de la fonction verifierAcces() (route pour la PEA)

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 15*).

```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 15 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.

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