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

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

Table des matières

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 – Infrastructure réseau

Pour cette partie, les guides d'installations sont disponibles sur notre GitHub sois dans */main/PartieCommune/Deploiement/Installation/PSW* sois */main/PartieCommune/Reseau*.

2.1 – Présentation

Comme dis dans notre dossier commun, notre infrastructure réseau est un seul et unique grand réseau d'adresse 192.168.248.0/21 (*voir Figure 1*), qui est divisé en 3 sous réseau géré par 3 VLANs différents:

- Pedagogique (VLAN 10, adresse : 192.168.0.0/22), pour tous les PCs du campus accessible pour tout type d'utilisateur.

- Acces (VLAN 20, adresse : 192.168.4.0/22), pour toutes les BAEs et PEAs.

- Administration (VLAN 30, adresse : 192.168.30.0/22), pour tous les serveurs et postes d'administrations.

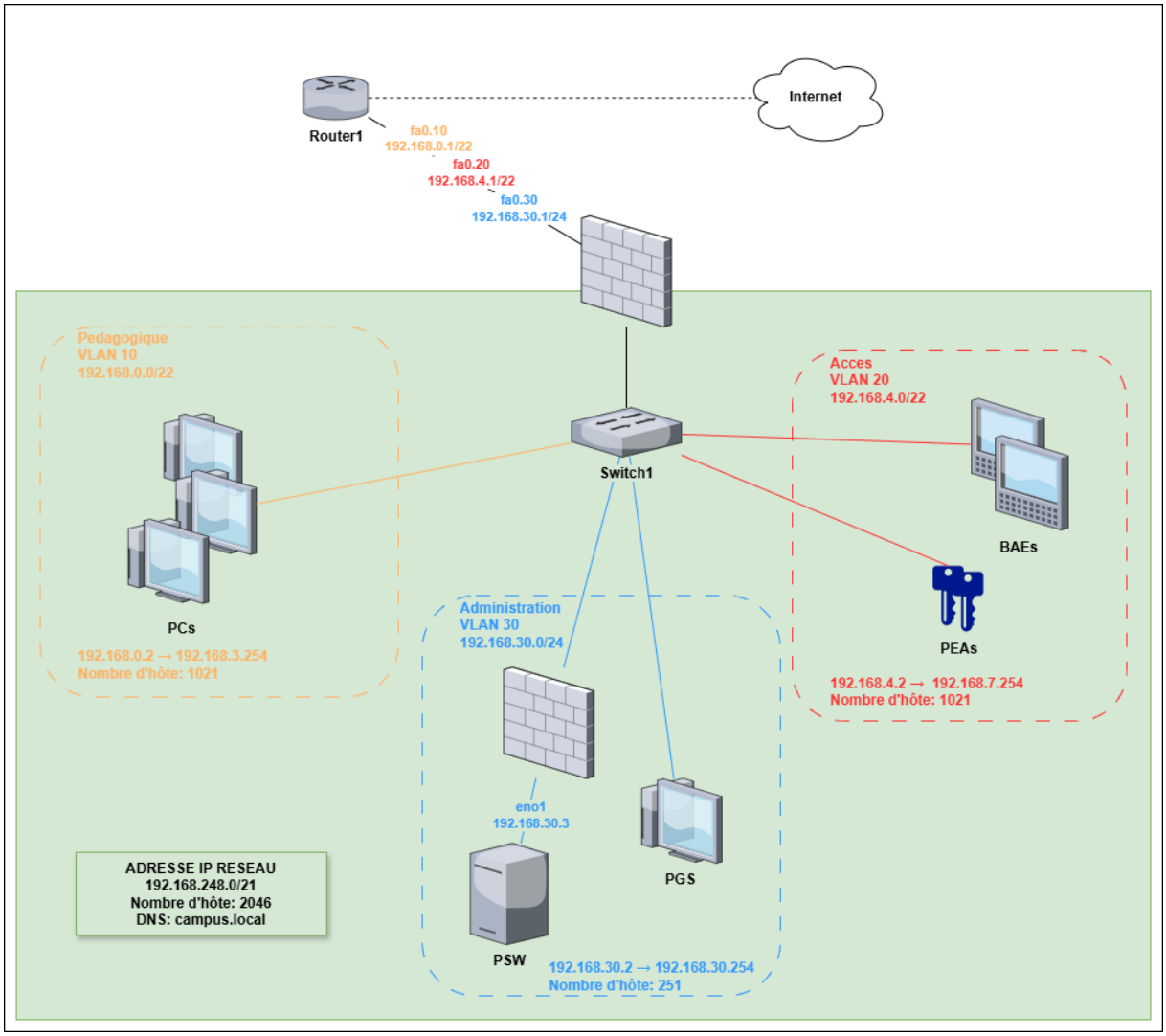

Figure 1 Infrastructure réseau

2.2 – Mise en place Switch

Les interfaces du switch sont toutes réparties entre les 3 VLANs (*voir Figure 2*). L'interface G0/1 est destiné au mode trunk et est lié au router pour faire du routage inter VLAN. Le VLAN 99 a donc comme adresse 192.168.248.2.

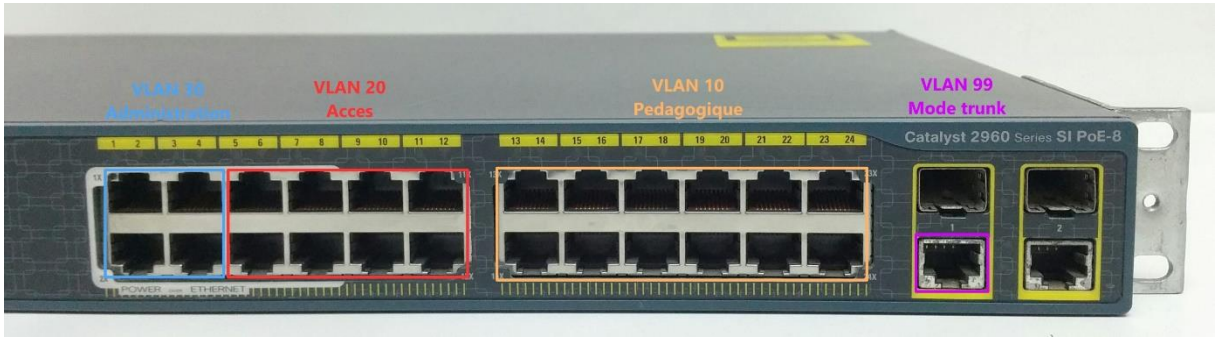

Figure 2 Répartition des interfaces du switch

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

2.3 – Mise en place Router

Le router est relié au switch depuis l'interface fa0. Sur le router VLAN 99 a comme adresse 192.168.248.1.

L'interface fa0 est divisé en trois sous interfaces :

- fa0.10 d'adresse 192.168.0.1/22
- fa0.20 d'adresse 192.168.4.1/22
- fa0.30 d'adresse 192.168.30.1/24

Enfin l'encapsulation dot1Q et le DHCP relay sont mis en place sur ces sous interfaces.

Une ACL est déployée afin de filtrer les requêtes. Le VLAN 30 lui peut communiquer avec les deux autres VLANs comme il veut, mais le VLAN 10 et VLAN 20 sont limiter vers le VLAN 30 aux ports 443 pour HTTPS, 68 pour le DHCP et 53 pour le DNS. Le VLAN 10 et le VLAN 20 ne peuvent pas du tout communiquer entre eux.

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. Les ports suivants sont ouverts :

- 22 : ssh
- 53 : DNS
- 80 : http
- 443 : https
- 5432 : pgAdmin
- 67 : DHCP

2.4.1 – Mise en place DHCP

Le PSW héberge donc aussi le DHCP. Trois configurations différentes sont faites afin de pouvoir distribuer correctement les bonnes adresses IP dans le bon VLAN grâce au DHCP relay:

- Pour le VLAN 10, le DHCP distribue les adresses allant de 192.168.0.2/22 à 192.168.3.254/22.
- Pour le VLAN 20, le DHCP distribue les adresses allant de 192.168.4.2/22 à 192.168.7.254/22.

- Pour le VLAN 30, le DHCP distribue les adresses allant de 192.168.30.10/24 à 192.168.30.254/24.

2.4.2 – Mise en place DNS

Un DNS est mis en place afin de mettre en place un serveur Proxy et de passer en HTTPS à l'aide de bind9. Le nom de domaine de notre réseau est campus.local. Trois noms d'hôte son mis en place :

- psw d'adresse 192.168.30.3.
- web d'adresse 192.168.30.3, pour accéder au site web.
- api d'adresse 192.168.30.2, pour accéder à l'API.

2.4.3 – Mise en place Proxy

Le proxy est mis en place à l'aide de nginx. A l'aide de openssl j'ai créé un certificat SSL auto-signé. Dans les fichiers de conf nginx le PSW écoute sur le port 443 afin de rediriger les requêtes HTTPS vers le bon service, sois l'api sois le site web. Il écoute aussi sur le port 80 afin de rediriger les requêtes HTTP vers du HTTPS.

III – Base de données et API

3.1 – Modèle conceptuel de données

Les données sont divisées en 13 tables (*voir Figure 3*). 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.

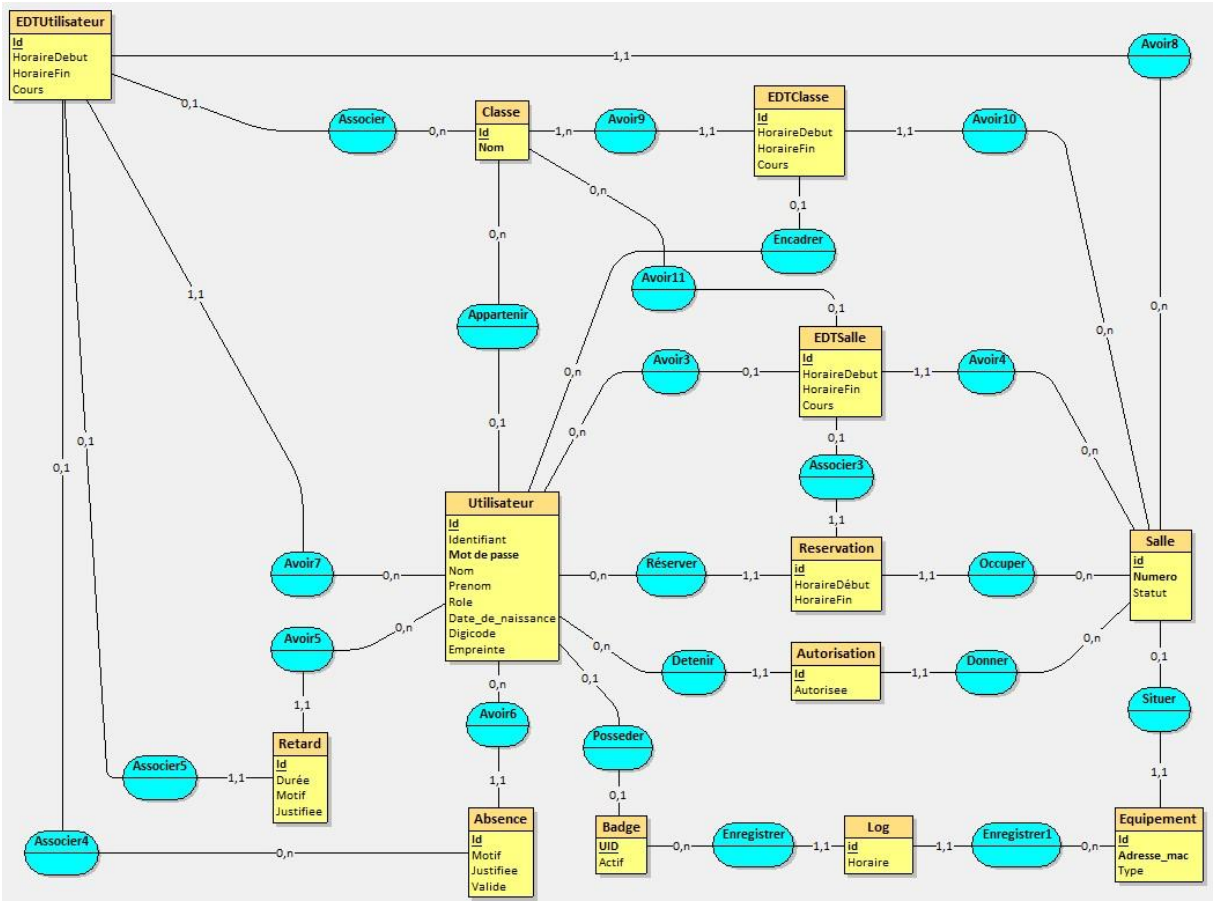

Figure 3 Modèle conceptuel de données

Il m'a fallut plusieurs essaie pour atteindre ce modèle. Les autres versions du MCD sont disponibles sur notre GitHub dans */main/BDD/Données/MCD*.

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 4*).

```
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 4 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 sous */main/BDD/Données*.

3.2 – Incréments et méthode de travail

Comme indiquer dans le dossier commun, notre groupe a décidé de diviser le projet en 3 groupes d'incrément avec chacun des dates butoirs.

Pour le premier groupe d'incrément :

- 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).

Pour le deuxième groupe d'incrément :

- PEA : Permettre l'accès à l'aide d'un digicode et vérifier l'autorisation d'accès, afficher les informations liées à l'utilisateur.
- BAE : Afficher les détails de la séance.
- PSW : Visualiser les disponibilités des salles et rechercher une salle libre pour une période données.
- PGS : Réservé une salle et visualiser l'occupation des salles.

Pour le troisième groupe d'incrément :

- PGS : Personnaliser les droits d'accès, visualiser l'historiques des badges créés.

Les diagrammes de cas d'utilisations complets sont disponibles sur notre GitHub dans */PartieCommune/UseCase*.

Pour la phase de réalisation, j'ai adopté une approche itérative et incrémentale, structurée autour d'un modèle en V. Chaque fonctionnalité suit un cycle précis : conception ciblée à l'aide d'un diagramme de séquence, élaboration d'un plan de test associé, développement de la fonctionnalité, phase de test avec Pytest, puis ajustements éventuels avant de passer à la fonctionnalité suivante (voir *Figure 5*).

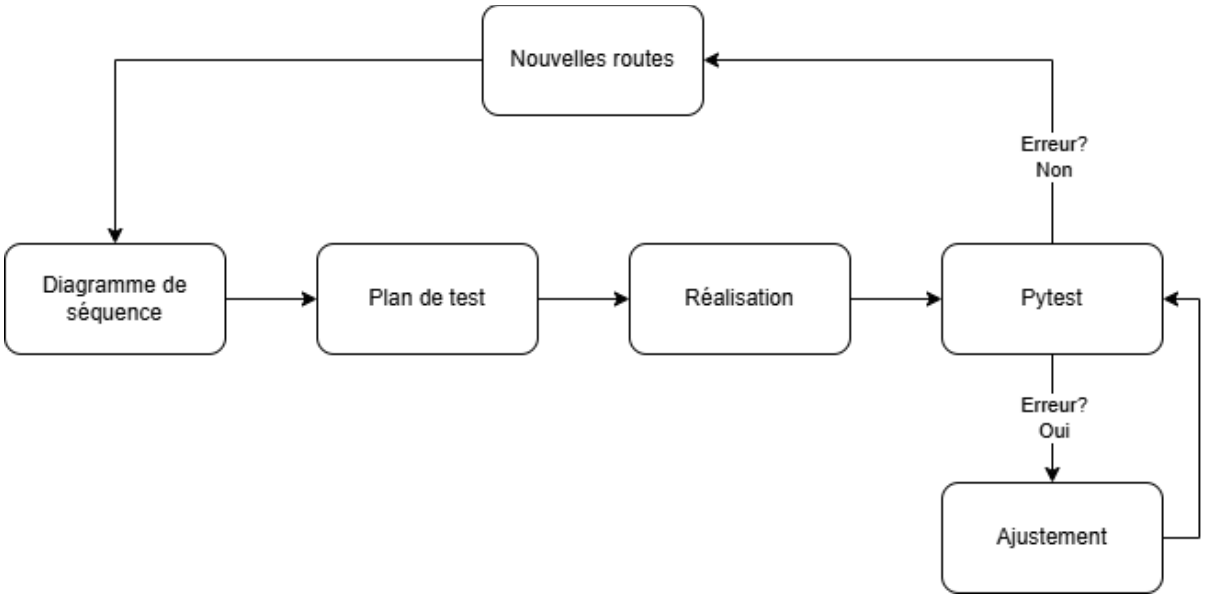

Figure 5 Cycle de réalisation

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

3.3 – Fonctionnement de l'API

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.

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

- 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 6*. 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 7*. 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 sous */main/BDD/Diagrammes/Classe*.

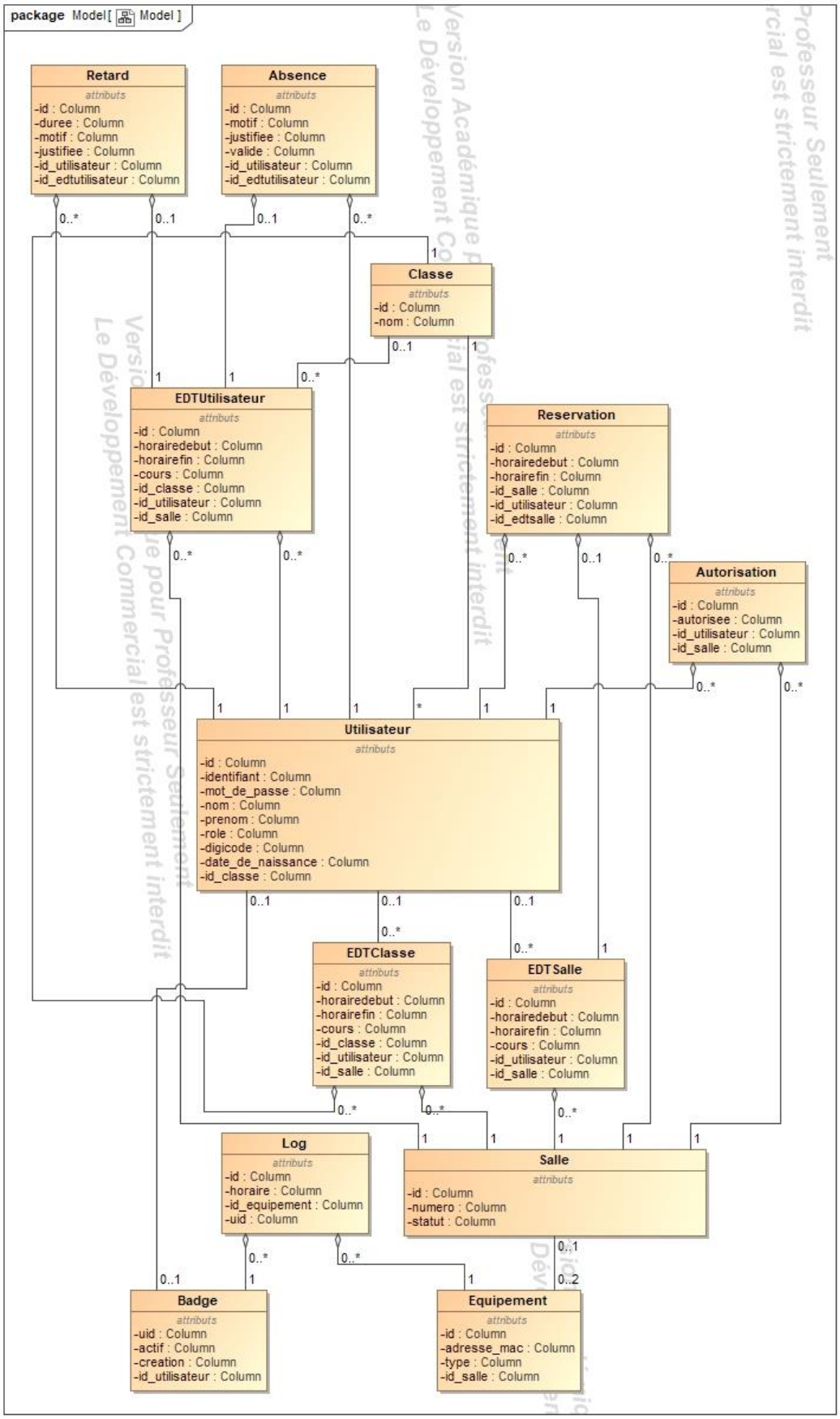

Figure 6 Diagramme de classe (simplifié) relatif à SQLAlchemy

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

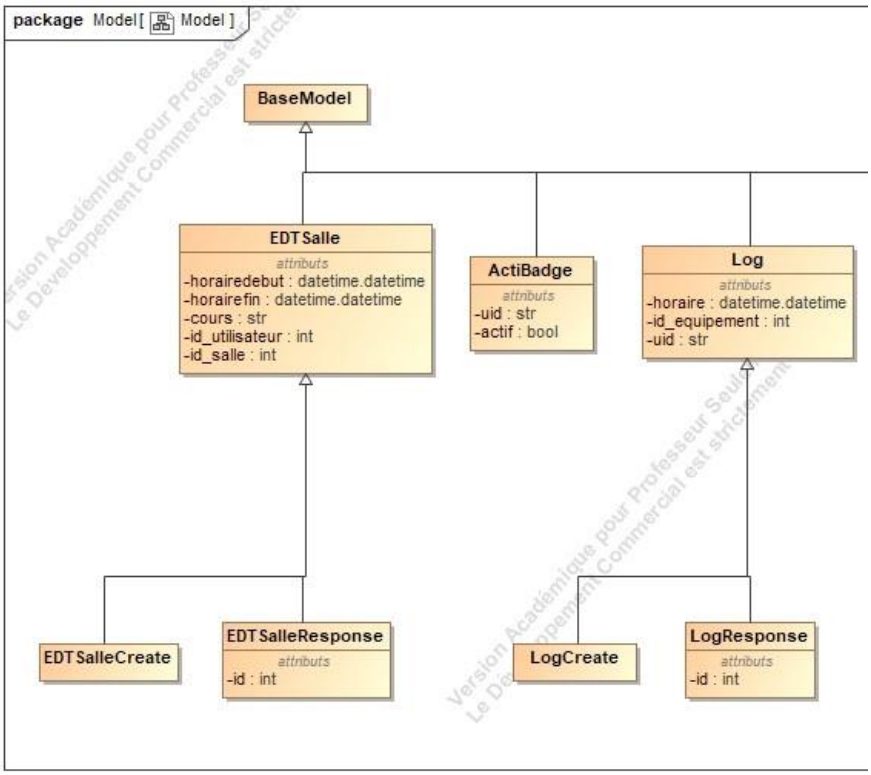

Figure 7 Diagramme de classe (partielle) relatif à Pydantic

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.

3.3.1 – Base de l'API

On commence par le fichier database.py (*voir Figure 8*). 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 8 Contenu du fichier database.py

Ensuite on créé les fichiers models.py (*voir Figure 9*) et schemas.py (*voir Figure 10*). 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 9 Contenue 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 10 Contenue partiel du fichier schemas.py

Un filtrage IP est mis en place sur l'API afin de n'autoriser seulement le VLAN 20 et 30 d'effectuer des requêtes POST, PUT, DELETE. Le VLAN 10 peut néanmoins accéder à la doc Swagger et effectuer des requêtes GET (*voir Figure 11*).

```
# Middleware IP Filtering
@app.middleware("http")
async def ip_filter_middleware(request: Request, call_next):
    # Vérifier si c'est une méthode POST
    if request.method in {"POST", "DELETE", "PUT"}:
        client_ip = request.client.host

        try:
            ip = ipaddress.ip_address(client_ip)
        except ValueError:
            raise HTTPException(status_code=400, detail="Adresse IP invalide")

        # Définir les sous-réseaux autorisés
        allowed_networks = [
                ipaddress.ip_network('172.20.0.0/16'), #Baronnerie
            ipaddress.ip_network('192.168.4.0/22'), #VLAN 20
            ipaddress.ip_network('192.168.30.0/24'), #VLAN 30
            ipaddress.ip_network('127.0.0.1/32'), #Localhost
        ]

        # Vérifie si l'IP appartient à l'un des réseaux autorisés
        if not any(ip in network for network in allowed_networks):
            raise HTTPException(status_code=403, detail="Accès interdit : IP non autorisée")

    # Continuer la requête
    response = await call_next(request)
    return response
```

Figure 11 Middleware filtrage IP

3.3.2 – Visualisation

Pour la création d'une route, on va se pencher sur une en particulière : celle de la PEA pour autoriser l'accès via badge. 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 12*).

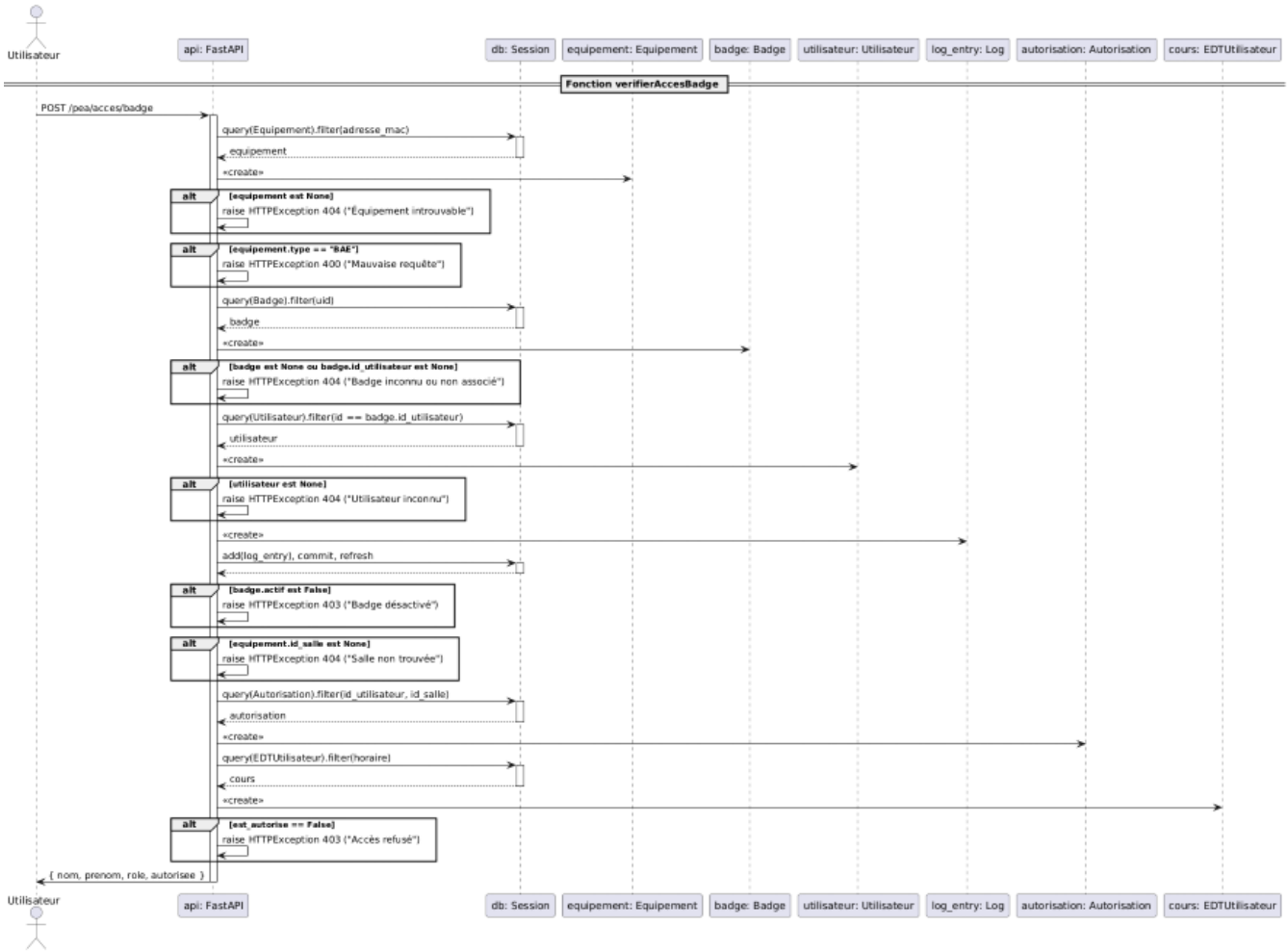

Figure 12 Diagramme de séquence pour l'accès à une salle

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 rentrer 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 :

1 - Identification du test

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

2 - Référence du module testé

routes/acces.py – Fonction verifierAcces

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** : Requête POST envoyée à la route verifierAcces
- **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
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

N° Test	Condition	BDD Préparée	Résultat attendu	Statut attendu
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

3.3.3 – 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 13*).

```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 13 Contenue partiel du fichier pea.py

Il ne me reste plus qu'à 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 14* présente le procès-verbal à la suite des tests réalisés sur verifierAccesBadge.

verifierAccesBadge		
Date du test:	12/03/2025	
Type du test:	Fonctionnel	
Objectif du test:	Vérifier le bon fonctionnement de la fonction verifierAccesBadge.	
Condition du test		
Etat initial	Environnement du test	
Pytest installé	Le test est réalisé sur une machine virtuelle Debian 12 à du fichier pytest verifierAccesBadge.py.	
Procédure du test		
Opération	Résultats attendus	Résultats obtenus
T1.1 - Adresse MAC inexistante	404 Équipement introuvable	404 Équipement introuvable
T1.2 - Adresse MAC d'une BAE	400 Mauvaise requête	400 Mauvaise requête
T1.3 - UID badge inconnu	404 Badge inconnu ou non associé	404 Badge inconnu ou non associé
T1.4 - Badge non lié à utilisateur	404 Badge inconnu ou non associé	404 Badge inconnu ou non associé
T1.5 - Utilisateur inexistant	404 Utilisateur inconnu	404 Utilisateur inconnu
T1.6 - Badge désactivé	403 Badge désactivé	403 Badge désactivé
T1.7 - Equipement sans salle	404 Salle non trouvée	404 Salle non trouvée
T1.8 - Pas d'autorisation ni de cours	403 Accès refusé	403 Accès refusé
T1.9 - Autorisation non autorisée	403 Accès refusé	403 Accès refusé
T1.10 - Autorisation autorisée	Accès autorisé + infos utilisateur	Accès autorisé + infos utilisateur
T1.11 - 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 14 Procès-verbal 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 sous */main/BDD*. / ! \

IIII – Conclusion

4.1 – Réalisation

Dans l'ensemble, le développement du projet est terminé. L'API permet désormais de gérer efficacement les accès via badge et digicode, en se basant sur les équipements en place et les emplois du temps enregistrés.

L'ensemble des fonctions critiques ont été testées avec Pytest à l'aide de mocks pour simuler les cas métiers. Il ne reste qu'une amélioration fonctionnelle à intégrer : permettre l'affichage des informations du cours lorsqu'un enseignant badge sur un équipement de type BAE.

Par ailleurs, une amélioration technique serait d'ajouter un système de journalisation afin d'enregistrer les appels de l'API dans un fichier de log, pour faciliter le suivi et le débogage.

4.2 – Ressentis personnel

Ce projet m'a permis de développer de nouvelles compétences, aussi bien techniques qu'humaines. J'ai appris à m'adapter aux besoins de chaque membre du groupe, ce qui m'a poussé à être plus à l'écoute et plus flexible. J'ai pris plaisir à travailler régulièrement, notamment en avançant chaque soir sur les différentes parties du projet. J'ai acquis de nombreuses connaissances, notamment sur la mise en place complète d'un réseau et l'intégration d'une API avec une base de données PostgreSQL.

Pour les projets à venir, je veillerai à suivre une planification plus rigoureuse et à la mettre à jour régulièrement. J'améliorerai également la communication dans le groupe, en gardant une trace écrite des échanges, et je structurerai mieux le dépôt GitHub, notamment en utilisant des branches séparées pour chaque incrément.