TD Gestion de projet - 01

Analyse et conception du projet

Introduction

Une entreprise souhaite mettre en place une application Web pour pouvoir gérer la réservation de ses salles de réunions, l'ancienne méthode étant de réserver les salles en renseignant un encart papier sur la porte.

Les collaborateurs font face aux problématiques suivantes :

- Difficulté à trouver une salle disponible
- Des salles réservées régulièrement sont parfois vides
- Les collaborateurs en déplacement ne peuvent pas libérer les salles réservées sans passer par des personnes présentes sur place.

Objectif du projet

L'objectif du projet est de développer une application Web qui permettra de gérer efficacement la réservation des salles de réunions, elle devra répondre aux besoins suivants :

- Faciliter la recherche de salles disponibles
- Optimiser l'utilisation des salles
- Permettre la gestion à distance

Les utilisateurs de l'application seront les collaborateurs ainsi que les office manager. Les collaborateurs pourront lister les salles, voir leur statut de réservation, faire une réservation et les annuler.

Les salles seront réservées suivant des périodes précises. Les Office Managers pourront effectuer les mêmes actions que les collaborateurs, ils pourront également ajouter ou supprimer des salles, temporairement empêcher la réservation de celles-ci ou annuler des réservations existantes.

Chaque salle a une capacité maximale qui lui est assignée à la création. Après blocage d'une salle, les Office Managers peuvent également changer la capacité de la salle.

Contraintes

L'application s'intégrera dans le SI actuel de l'entreprise. Il sera donc déployé par l'équipe infrastructure interne de l'entreprise. De plus, la maintenance sera assurée par les équipes

de développement de l'entreprise, par conséquent il faudra que l'application puisse être déployée selon les standards de l'entreprise.

Ainsi, l'application devra être développée en Python, en utilisant FastAPI comme framework web et SQLAlchemy pour la persistance des données, pouvoir être packagée dans une image de conteneur (au format Dockerfile, utilisable aussi bien avec Docker qu'avec Podman) et que l'on puisse configurer plusieurs instances pour chaque immeuble. De plus, chaque fonctionnalité devra être couverte par un test unitaire, et l'application devra exposer un minimum d'observabilité (état de santé, journalisation) pour faciliter son exploitation.

Livrable

Les livrables attendus pour ce projet sont :

- Un cahier des charges.
- Un planning prévisionnel estimant le nombre d'itérations nécessaires à développer cette application.
- Un repository git avec le code source de l'application.
- Un Dockerfile permettant de containeriser l'application, construisible et exécutable avec Podman.
- Un fichier d'intégration continue Gitlab CI permettant d'exécuter les tests du projet et de créer et déployer l'image du conteneur applicatif.

Contact

Pour toute question ou clarification, veuillez contacter les responsables de TD.

Planification, initialisation et structure du projet

Avant Projet et rédaction du cahier des charges

1. En groupes de 2 ou 3 préparer la rédaction du cahier des charges.
2. Lire la présentation du projet.
3. Lister les uses cases de l'application afin d'en agrémenter le cahier des charges.
4. Quels sont les risques liés à cette application ? En agrémenter également le cahier des charges. Ceux-ci seront regroupés dans 3 catégories :
    a. Risques liés à la conception et au développement
    b. Risques de planification
    c. Risques liés à la maintenance

5. Modéliser la base de données et les entités associées.
6. Écrire les user stories SCRUM et chiffrer la complexité des user stories.
7. On estimera que l'on peut effectuer 20 points par sprint et qu'un sprint coûte environ 10000€, produire une estimation de la durée du projet
8. Identifier les dépendances entre les Story et construire un diagramme de dépendances entre les stories.
9. Faire un diagramme de Gantt pour avoir une idée de ce qui peut être parallélisé ainsi que de pouvoir estimer comment la charge sera répartie entre les sprints

Initialisation du projet

Objectifs

10. Initialiser un projet Python avec uv.
11. Ajouter FastAPI, un serveur ASGI (uvicorn) et SQLAlchemy comme dépendances.
12. Configurer Alembic pour les migrations de base de données.
13. Configurer les dépendances et le framework de tests (pytest).
14. Mettre en place un point d'entrée minimal de l'API, avec un endpoint de santé /health, couvert par un premier test.
15. Configurer le moteur SQLAlchemy et la connexion à la base de données.
16. Initialiser un dépôt Git pour le projet.

Étapes d'initialisation du projet

17. Installer uv

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

18. Initialiser un projet à l'aide de uv.

```
uv init reservation_salle
cd reservation_salle
```

19. Ajouter les dépendances pour créer un projet FastAPI ainsi que pour les tests

```
uv add fastapi "uvicorn[standard]" sqlalchemy alembic
python-multipart
uv add --dev pytest pytest-asyncio httpx pytest-html
```

python-multipart est nécessaire dès que vous exposerez un endpoint de login basé sur le formulaire standard de FastAPI (OAuth2PasswordRequestForm) au TD 02 ; l'ajouter dès maintenant évite une erreur de démarrage surprise plus tard.

20. Organiser le code de l'application dans un package Python reservation_salle/ avec un point d'entrée main.py exposant l'application FastAPI et un premier endpoint de santé :

```
from fastapi import FastAPI

app = FastAPI(title="Réservation de salles")

@app.get("/health")
def health():
    return {"status": "ok"}
```

21. Démarrer le serveur en rechargement automatique :

```
uv run uvicorn reservation_salle.main:app --reload
```

Vous pouvez vérifier que le serveur est correctement démarré en vous rendant à l'adresse http://localhost:8000/health, ainsi que sur la documentation interactive générée automatiquement par FastAPI à l'adresse http://localhost:8000/docs.

22. Créer le fichier reservation_salle/database.py qui configurera le moteur SQLAlchemy et exposera la Base déclarative ainsi qu'une dépendance FastAPI pour obtenir une session :

```
import os

from sqlalchemy import create_engine
from sqlalchemy.orm import DeclarativeBase, sessionmaker

DATABASE_URL = os.environ.get("DATABASE_URL",
"sqlite:///./reservation_salle.db")

engine = create_engine(DATABASE_URL,
connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)


class Base(DeclarativeBase):
    pass


def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Important : conservez précisément les noms Base et get_db dans ce module. Les tests fournis au TD 02 (fixtures de conftest.py) importent reservation_salle.database.Base et reservation_salle.database.get_db pour brancher une base de test ; c'est le seul point de votre code, avec les routes HTTP, sur lequel ils s'appuient. Le reste (vos entités, votre RBAC...) est entièrement libre.

23. Initialiser Alembic pour la gestion des migrations de base de données :

```
uv run alembic init migrations
```

Configurer ensuite le fichier alembic.ini (ou une variable d'environnement) pour qu'il pointe vers la base de données du projet. Une deuxième modification, dans migrations/env.py, sera nécessaire une fois vos entités écrites au TD 02 : sans elle, la génération automatique de migrations ne détecte aucun changement.

24. Configurer pytest dans le fichier pyproject.toml :

```
[tool.pytest.ini_options]
testpaths = ["tests"]
python_files = ["test_*.py", "*_test.py"]
```

25. Écrire un premier test dans tests/test_health.py, pour vérifier que la chaîne complète (pytest, FastAPI, TestClient) fonctionne :

```
from fastapi.testclient import TestClient

from reservation_salle.main import app

client = TestClient(app)


def test_health():
    response = client.get("/health")
    assert response.status_code == 200
    assert response.json() == {"status": "ok"}
```

26. Exécuter les tests pour vérifier que tout fonctionne :

```
uv run pytest
```

Initialisation du dépôt Git

uv init a déjà initialisé un dépôt Git (dossier .git) et un fichier .gitignore basique pour vous ; git init ci-dessous ne fera donc rien de plus si vous l'exécutez, c'est normal.

27. Vérifier l'état du dépôt :

```
git init
git status
```

28. Compléter le fichier .gitignore généré par uv avec les entrées suivantes, pertinentes pour ce projet :

```
*.db
.pytest_cache
.idea
.DS_Store
docs/_build
report.xml
```

29. Créer le commit initial du projet :

```
git commit -m "Initialisation du projet FastAPI avec uv et
configuration des tests"
```

30. Créer la branche de développement pour suivre le principe de git flow :

```
git checkout -b develop
```