2025 - 2026

Dossier Projet CDA : Application FaceSwap

APPLICATION MICROSERVICES

ALI MARZAK

M2I FORMATION |11 rue de la Haye, 67300 Schiltigheim

SOMMAIRE

Table des matières

I- Introduction

A. Contexte

Actuellement, nous vivons dans l'âge d'or de l'Intelligence artificielle qui est devenu un outil complémentaire pour certains, un indispensable pour d'autres ou encore une source de progression infini pour les plus optimistes.

Le projet **FaceSwap** s'inscrit dans ce contexte permettant de modifier un visage dans une image via une plateforme numérique.

FaceSwap offre ainsi un espace sécurisé où chaque client a son espace privé dans lequel il peut utiliser ses services via un abonnement sécurisé.

B. Compétences couverts (REAC)

AT1 - Développer une application sécurisée :

Installer et configurer son environnement de travail en fonction du projet.

Développer des interfaces utilisateur

Développer des composants métier

Contribuer à la gestion d'un projet informatique

AT2 - Concevoir et développer une application sécurisée organisée en couches :

Analyser les besoins et maquetter une application

Définir l'architecture logicielle d'une application

Concevoir et mettre en place une base de données relationnelle

Développer des composants d'accès aux données SQL et NoSQL

AT3 - Préparer le déploiement d'une application sécurisée :

Préparer et exécuter les plans de tests d'une application

Préparer et documenter le déploiement d'une application

Contribuer à la mise en production dans une démarche DevOps

II- Analyse des Besoins et Cahier des Charges

A. Charte graphique

Couleur 1 : #ffe1bc

Couleur 2 : #ffcfd1

Couleur 3 : #f3c6f1

Logo :

B. Responsive Design

➢ Approche Mobile First

Conception graphique sur Figma avec une approche où l'on développe en commençant par la version mobile.

➢ BreakPoint Ecran

Utilisation des Breakpoints fixés par Tailwind CSS.

C. Accessibilité

a- Contraste

Assurer un bon contraste du Front de notre application avec le choix des couleurs et une vérification avec l'outil WhoCanUse.

b- Navigation Clavier

Rendre le site navigable grâce à une bonne sémantique HTML.

c- Textes Alternatifs

Ajout d'un attribut Alt pour un texte alternatif

D. Description fonctionnelle et technique

1) Spécifications Fonctionnelles détaillés

a- En tant que Client

US01 : Je veux consulter les différents abonnements disponibles

US02 : Je veux pouvoir m'inscrire et m'authentifier à mon compte

US03 : Je veux recevoir une vérification de compte par email

US04 : Je veux pouvoir changer mon mot de passe dans mon profil

US05 : Je veux pouvoir réinitialiser mon mot de passe par email

US06 : Je veux souscrire à un abonnement

US07 : Je veux utiliser le service FaceSwap après une souscription

US08 : Je veux avoir un historique de mes dernières utilisations du service FaceSwap

US09 : Je veux pouvoir payer mon abonnement via Stripe les formulaires Stripe

US10 : Je veux récupérer mes factures Stripes

b- En tant qu'Admin

US11 : Je peux bannir un utilisateur

US12 : Je peux accéder aux abonnements selon leur statut

US13 : Je peux accéder aux paiements selon leur statut

2) Fonctionnalités principales

a- Pages publiques

- OnePage

Présentation du site
Timeline d'utilisation de l'application
Affichage des tarifs d'abonnements
Faq

- Page d'Inscription et d'Authentification

Formulaire d'inscription : Email + Mot de passe + Confirmation + Agree Termes

Formulaire de connexion : Email + Mot de passe + Se souvenir de moi

b- *Pages Client*

- Page Mon Abonnement

La page possède 2 états selon si le client :

- A un abonnement : Bloque les choix des abonnements + Affiche l'abonnement et son état
- N'a pas d'abonnement : Propose les abonnements

- Page Mon Service

La page propose le service principal de FaceSwap : Formulaire avec 2 images en entrées + Historique du service.

- Page Mes factures

Page qui liste toutes les factures de l'abonnement.

c- *Pages d'Erreur*

Ajouter les 3 pages d'erreurs suivants :

- Erreur 403
- Erreur 404
- Erreur 500

3) Spécifications Techniques

a- *Technologies Recommandées*

Back End : Java Spring Boot

Front End : Angular

Framework CSS : Tailwind CSS

Base de données : MySQL

b- *Librairie recommandée*

PrimeNG pour Angular

c- *Environnement de travail*

- VSCode pour Angular
- Eclipse pour Java
- Docker Desktop

d- *Contraintes et Exigences*

- Conformité RGPD
- Mentions Légales obligatoires
- Politique de Confidentialité
- Consentement
- Monitoring et maintenance

E. Sécurité et Protection des données

1) Sécurité générale

Confidentialité : Protection des données du Client

Intégrité : Garantir l'exactitude des données

Disponibilité : Assurer l'accès aux services autorisés 24/7

2) Authentification et Contrôle d'accès

Authentification par Email vérifié lors de l'inscription

Politique de Mot de passe fort

Token rafraîchit

Fonction "Se Souvenir de moi" (optionnelle)

Gestion des rôles User + Admin avec vérification d'accès

3) Protection contre les failles

Protection contre l'injection SQL :

- Utilisation exclusive de requêtes préparées
- ORM JPA pour l'accès aux données
- Validation des paramètres d'entrées
- Principe du moindre privilège pour la BDD

Protection contre les attaques de force brutes :

- Rate Limiter

Protection contre les failles XSS :

- Validation stricte des entrées
- CSP mis en place

F. Livrables attendus

✓ Conception UX/UI sur Figma
✓ Conception UML
✓ Modélisation et Intégration de la base de données
✓ Tests et validation
✓ Documentation technique de l'API
✓ Pipeline CI/CD

III- Méthodologie & Environnement de Développement

A. Gestion de projet

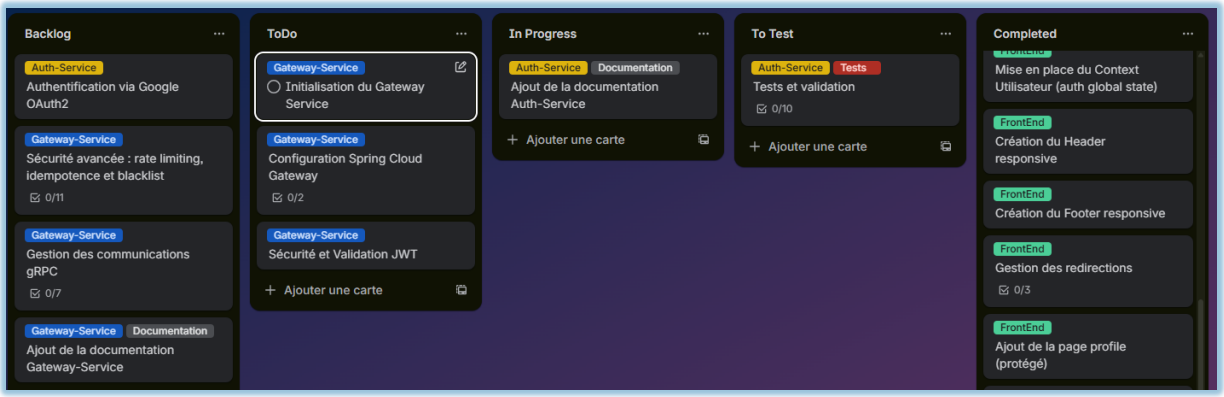
1) Méthode Agile : Scrum

Pour la gestion de mon projet, j'ai décidé d'utiliser la **méthode Agile** dont l'idée est d'apporter une souplesse et performance à mon projet.

Cette méthode est basée sur 3 piliers :

- Transparence : Garantir que tous les membres ont une compréhension commune du projet.
- Inspection : Rendre le travail examinable et évaluable régulièrement.
- Adaptation : Garantir une adaptation dans le développement pour répondre aux besoins et aux exigences des parties prenantes.

Pour la gestion de mon projet, j'ai utilisé l'outil **Trello**, qui m'a permis d'afficher et d'organiser efficacement toutes les tâches. J'ai structuré mon tableau Trello en m'inspirant d'une méthodologie Agile : **Scrum**. Cela m'a permis de visualiser le progrès du projet et de prioriser les tâches de manière efficace.



Comme on peut le voir ci-dessous, nous avons 5 colonnes :

- **Backlog :** Contient toutes les fonctionnalités à développer sur le long terme
- **To Do :** Contient toutes les tâches prioritaires pour le sprint en cours
- **In Progress :** Contient les tâches sur lesquelles je travaille actuellement
- **To Test :** Contient les tâches finies qui attendent d'être testés et validés
- **Completed :** Contient toutes les tâches qui sont complétement terminées et validées

2) Versionning Git

Dans mon projet, j'ai utilisé Git qui est un système de contrôle de version qui a joué un rôle primordial dans la gestion du code source de mon projet.

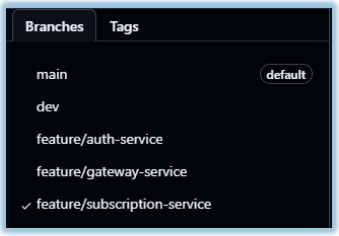
Les **fonctionnalités principales de Git** sont :

- Le contrôle de Version

Git permet de conserver un **historique complet des modifications** que j'ai pu apporter à mon code permettant ainsi de récupérer une ancienne version de mon code, de comparer des versions différentes ou encore de suivre l'évolution du projet.

- Branches et Merges

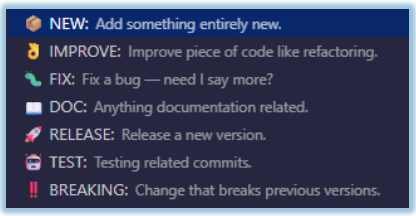
Git donne la possibilité de travailler sur des fonctionnalités du code via une version parallèle du code qu'on appelle **Branches** permettant ainsi de ne pas affecter la version principale.



Lorsque la/les fonctionnalité(s) sont terminées et correctement testées, on peut fusionner cette branche à notre branche principale en effectuant un **merge**.

- Commit

Pour les Commits de Git, j'utilise des Emoji pour mieux organiser et visualiser mes messages.



B. Environnement de Travail

1) Technologies utilisées

a- Back End

Pour gérer le coté Serveur de mon application, j'ai fait le choix d'utiliser le Framework **Spring Boot** qui est un Framework basé sur le langage Java.

Le projet est basé sur la dernière version stable Spring boot 3.5.6 avec l'utilisation de la version 25 de Java pour avoir les dernières fonctionnalités.

Pourquoi avoir choisi ce Framework pour mon Back End ?

Tout d'abord, le langage Java est connu pour sa robustesse. En effet, ce langage est derrière la plupart des applications Android et reste très utilisé dans le monde du développement.

La raison derrière cette notoriété est que le langage Java propose une écriture stricte du code avec peu de liberté demandant plus d'effort pour finalement avoir un résultat propre et solide rendant une application très robuste.

Pour ce qui est du choix du Framework, Spring boot reste le Framework Java le plus connu et proposant une large palette d'outils avec une écriture respectant les nombreux design pattern. De plus, mon backend se base sur une architecture micro services, donc il est essentiel d'utiliser les outils d'un Framework évitant de "réinventer la roue".

b- Front End

Pour gérer le coté Client de mon application, j'ai décidé d'utiliser le Framework **Angular** qui est un Framework basé sur le langage JavaScript.

Le Front End est basé sur la dernière version active et stable Angular 21.

Pourquoi avoir choisi ce Framework pour mon Front End ?

Tout d'abord, j'ai eu l'occasion de tester plusieurs Framework JS mais pas Angular qui fait partie des standards en termes de Framework JS. Pour moi, il était essentiel de l'utiliser dans un de mes projets pour le tester et avoir un avis.

De plus, Angular est connu pour sa robustesse et son efficacité. En effet, il reprend quasiment les mêmes notions que le Framework Spring Boot avec notamment l'injection de dépendances ou encore l'inversion de contrôle. Cette proximité et cette robustesse m'ont donné plus de motivations pour l'utilisation de ce Framework.

Pour la partie Front, je combine Angular avec le Framework **Tailwind CSS**. Il s'agit d'un Framework de design CSS qui se base sur une approche de classes utilitaires.

Pour ma part, l'utilisation de Tailwind avec Angular reste une solution très puissante puisqu'il offre une meilleure gestion du code et une meilleure visualisation de l'interface. En effet, j'aime beaucoup savoir le comportement visuel de chaque composant.

Pourquoi j'ai choisi d'utiliser Tailwind CSS ?

- Nommage des classes : je fais des parties des personnes qui peuvent être rapidement à court d'idée lorsqu'il s'agit de trouver des noms de classes et de vite me mêler les pinceaux. Tailwind CSS m'évite ainsi ce problème.
- Gain de temps considérable : En gérant le design directement au niveau du composant Angular, cela m'évite de faire des allers-retours vers des fichiers CSS.
- Gain de Performance : Tailwind CSS élimine directement le CSS non utilisé ce qui va réduire considérablement la taille finale des fichiers CSS et donc améliorer les performances.

2) Librairies utilisées

Pour la partie Front End, j'ai fait le choix d'utiliser la librairie **PrimeNg** qui s'associe très bien avec Angular et qui offre de nombreux composants déjà conçus et prêt à l'utilisation. Parmi ces composants, on y trouve des boutons, des entrées pour les formulaires ou encore des composants pour des images.

3) Stack technique

Pour ce qui est des logiciels utilisés, j'ai fait le choix d'utiliser **VSCode** pour le développement du Front End avec Angular car ce logiciel propose de nombreuses extensions pour la qualité du code ou encore des extensions sur le Framework Angular.

Pour le développement du Back End avec Spring boot, j'ai décidé d'utiliser **l'IDE Eclipse for Java Developers** puisque celui-ci offre un compilateur optimisé pour le langage Java qui reste un langage compilé.

Pour faciliter le développement global de mon application, j'ai décidé d'utiliser **Docker Desktop** possédant un moteur Docker Engine qui permet de créer, exécuter et gérer des conteneurs.

IV- Conception de la solution (UI/UX & Modélisation)

A. Conception UI/UX du système

UX Expérience utilisateur :

L'expérience utilisateur correspond à la facilité avec laquelle un utilisateur va parcourir une application Web.

Pour cela, il existe plusieurs choses que j'ai mises en place pour améliorer l'expérience utilisateur :

- Affordance : Elle fait référence à la manière dont les éléments de l'interface suggèrent leur utilisation. Par exemple, lorsqu'un utilisateur voit un bouton, il doit savoir que ce bouton est présent pour être cliqué ou qu'un lien hypertexte va nous rediriger vers une autre page. Pour cela, j'ai fait en sorte d'ajouter un effet ombre à mes boutons ou encore des Hover et pour les boutons avec des icones.
- Loi de Fitts : Cette loi nous dit que plus l'Utilisateur atteint rapidement son objectif, plus son expérience utilisateur sera positive. J'ai donc mis en place les règles des 3 cliques qui vont permettre à l'utilisateur d'accéder à n'importe quelle partie de mon application en moins de 3 cliques.
- Loi de Hicks : Scientifiquement, il est prouvé que plus un individu a de choix, plus la décision sera longue à prendre. Pour cela, j'ai fait en sorte que mon application soit simple et épurée, proposant un nombre d'options limité ce qui va mener l'utilisateur vers une navigation plus fluide de mon application Web.
- Choix des mots : De plus, il est très important de choisir des mots clairs et pertinents que ça soit au niveau du titre ou des descriptives ce qui va rendre l'application plus simple à comprendre et donc à naviguer. En effet, un utilisateur qui ne comprend pas un bouton aura moins envie de cliquer dessus.

UI Interface Utilisateur :

L'Interface Utilisateur se rapporte à tous l'environnement graphique ou visuelle auquel un utilisateur va faire face lorsqu'il accède à une application Web.

Pour cela, il est très important d'avoir une interface agréable et facile d'utilisation car plus un utilisateur est sur un site agréable à visiter et plus il va rester sur notre application.

Pour améliorer l'interface utilisateur, j'ai fait plusieurs choix importants :

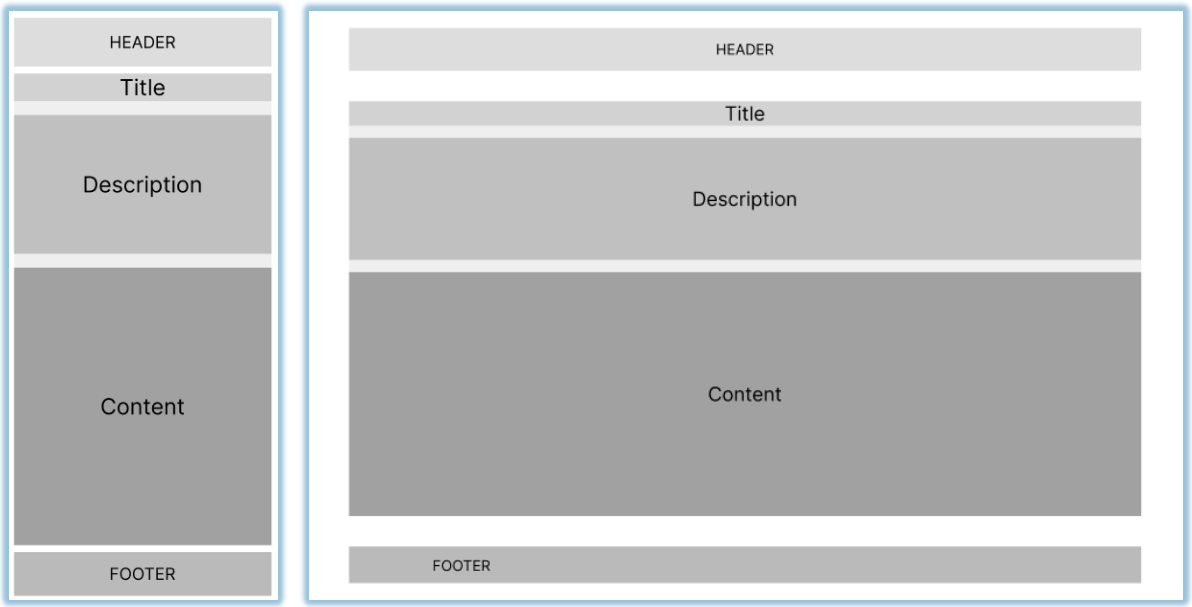
- Cohérence visuelle : J'ai décidé pour mon application d'utiliser des couleurs plus conviviales et chaleureuses. Pour cela, j'ai décidé de partir sur une couleur Orange qui se rapproche un petit peu du rouge. Pour renforcer cette cohérence visuelle, j'ai aussi utilisé les palettes de couleurs proposées par Tailwind CSS qui ont été choisis pour développer le front d'une application de manière cohérente.

- Réactivité et feedback visuel : Lorsque l'utilisateur interagit avec l'interface, il reçoit un feedback visuel immédiat, soit via un changement de couleur, une animation permettant à l'utilisateur de comprendre rapidement l'effet de ses actions. C'est aussi pour cela que j'ai décidé d'utiliser Angular qui permet de rendre mon application beaucoup plus dynamique et réactive.
- Responsive Design : En effet, que mon application s'adapte à n'importe quel type d'écran rend l'interface travaillée quel que soit l'appareil utilisé et donc l'utilisateur garde une interface agréable.

1) Zoning

Lors de la conception graphique, la première étape est le **Zoning** qui a pour rôle de structurer les pages et placer les différents éléments pour qu'ils suivent une certaine hiérarchie de criticité.

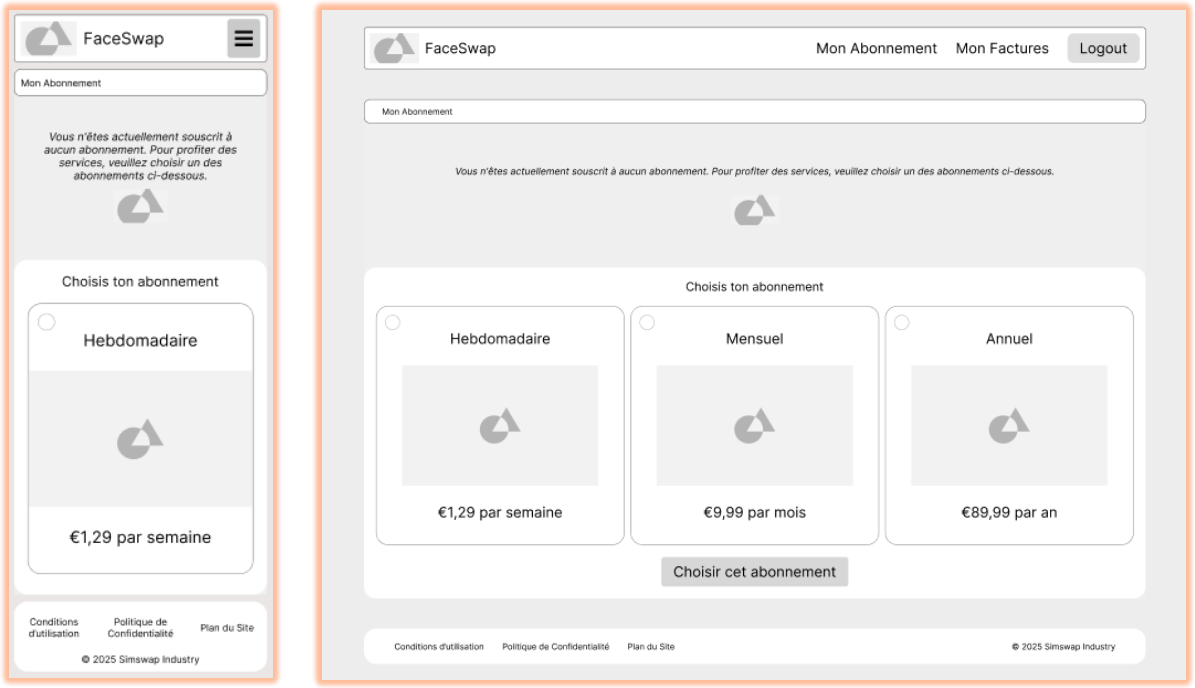
Le Zoning ne présente ni style ni d'éléments textuelles, l'objectif est seulement de placer les gros éléments comme ci-dessous avec la version mobile et la version Desktop.



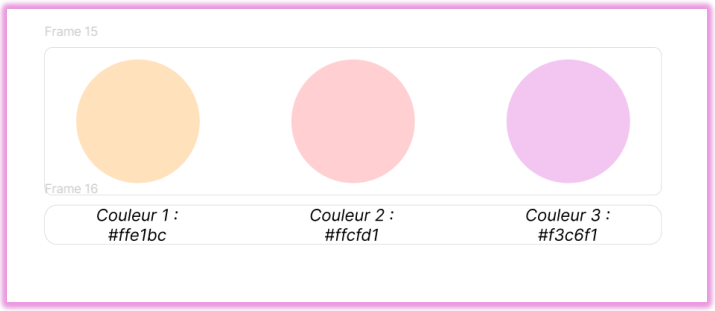
2) Wireframe

Le **Wireframe** est l'étape suivante après la validation du Zoning. Ici, nous allons mettre l'accent sur le parcours utilisateur et les fonctionnalités essentiels.

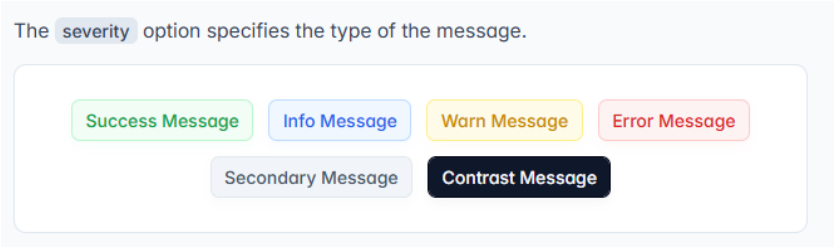
Ainsi, l'objectif sera de prévisualiser le comportement lors de la navigation et de mettre en avant les fonctionnalités. La charte graphique n'est pas mise en avant et aucune présence de design visuel.



3) Charte graphique + Librairies



Pour ce qui est des composants, j'ai décidé d'utiliser la librairie PrimeNg qui propose une multitude d'éléments personnalisables comme pour les messages ci-dessous :

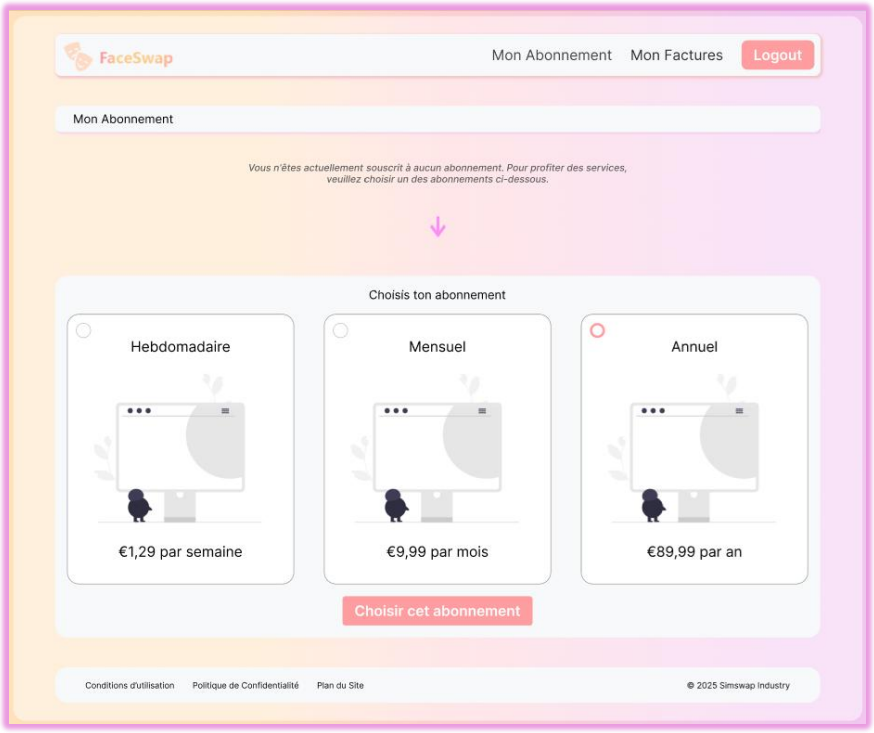


4) Maquettes graphiques

Enfin, lorsqu'on a validé le Wireframe et la charte graphique, on passe à la création de la **Maquette graphique**. C'est dans cette partie qu'on va définir l'identité visuelle de l'application (couleurs, typographies, icônes et image).

L'objectif est que l'intégration Front End soit le plus fidèle à cette maquette.

Enfin, c'est ici qu'on veille à ce que ca respecte les bonnes pratiques d'accessibilité.



B. Modélisation UML

L'**UML (Unified Modeling Language)** est un langage de modélisation graphique universel dont l'objectif est de visualiser, spécifier, construire des systèmes logiciels.

1) Diagramme de cas d'utilisation (UseCase)

Comme son nom l'indique, ce diagramme identifie quels sont les fonctionnalités accessibles par un utilisateur.

Elle permet de répondre à la question : **Qui fait quoi dans le système ?**

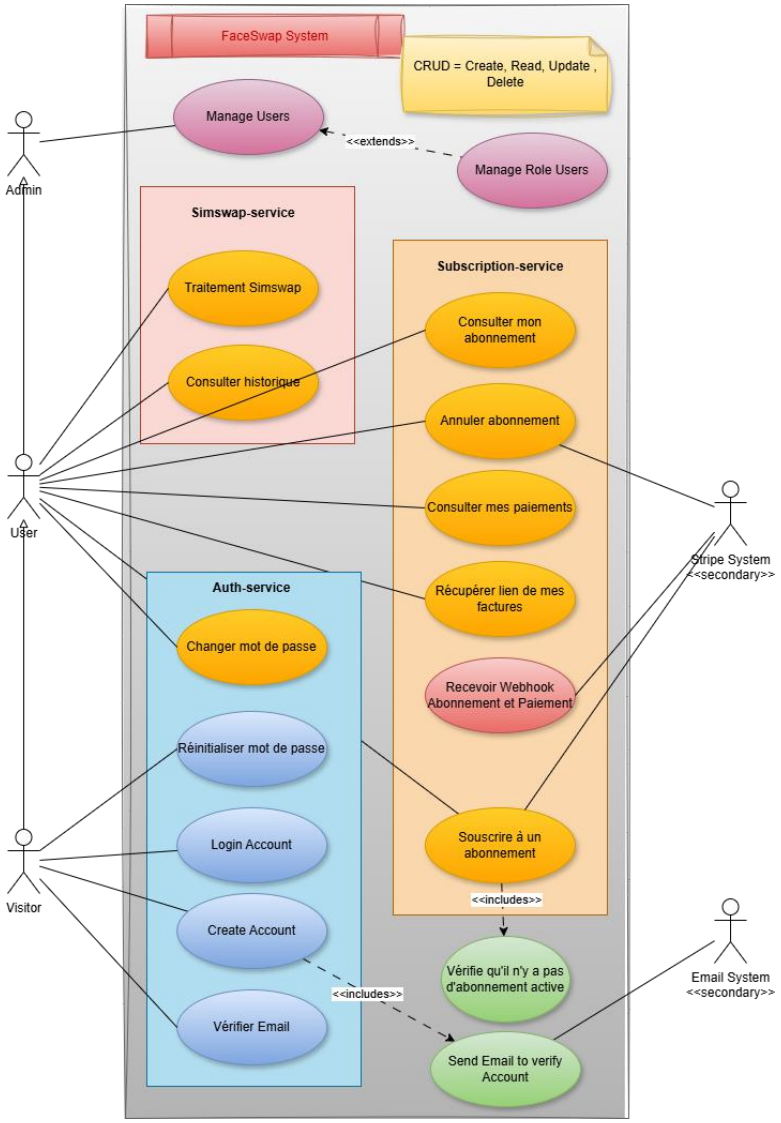


Figure 1 Diagramme de cas d'utilisation

2) Diagramme de classes

Ce **diagramme de Classe** est un diagramme très important pour la conception orientée objet. En effet, contrairement au diagramme précédent qui représente un système du point de vue des acteurs, celui-ci en montre la structure interne.

Ainsi, il fournit une représentation abstraite des objets du système qui vont interagir pour réaliser les cas d'utilisation.

Elle permet de répondre à la question : **De quoi est composé le système et quels sont leur relation ?**

J'ai décidé de vous présenter le diagramme de classe pour auth-service :

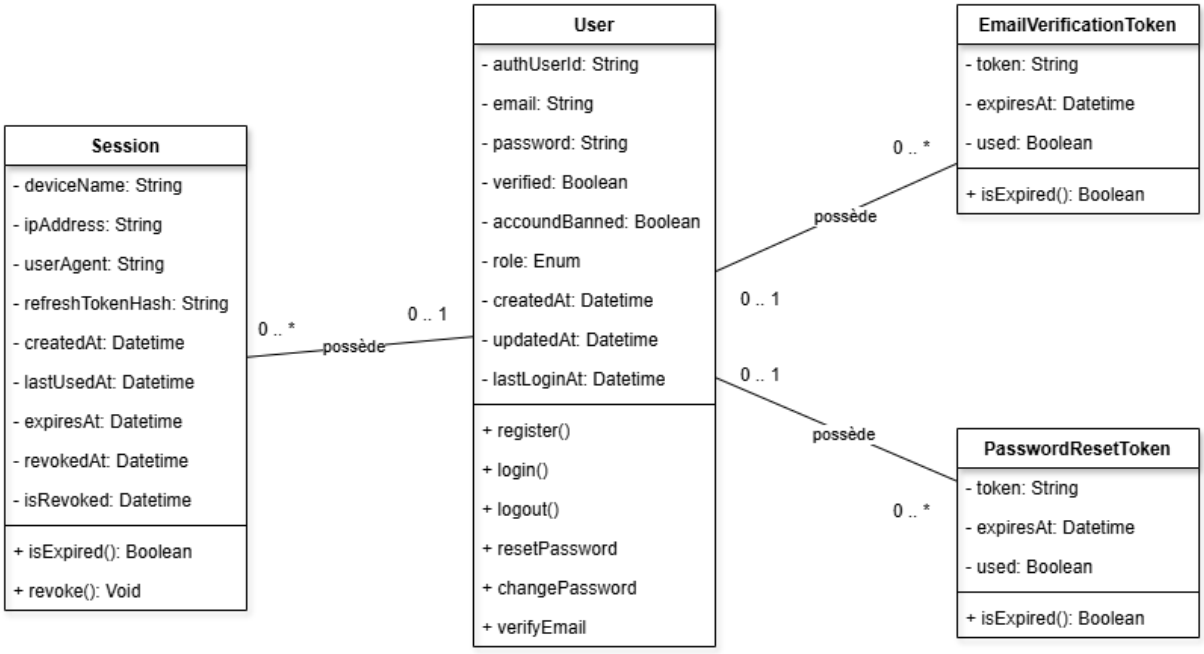


Figure 2 Diagramme de Classe (Auth-Service)

3) Diagramme de séquences

Contrairement au diagramme de classes qui représentent de manière statique, ce **diagramme de Séquence** montre comment les composants interagissent dans le temps pour réaliser une action précise.

Elle permet de répondre à la question : **Comment communiquent-ils entre eux dans le temps ?**

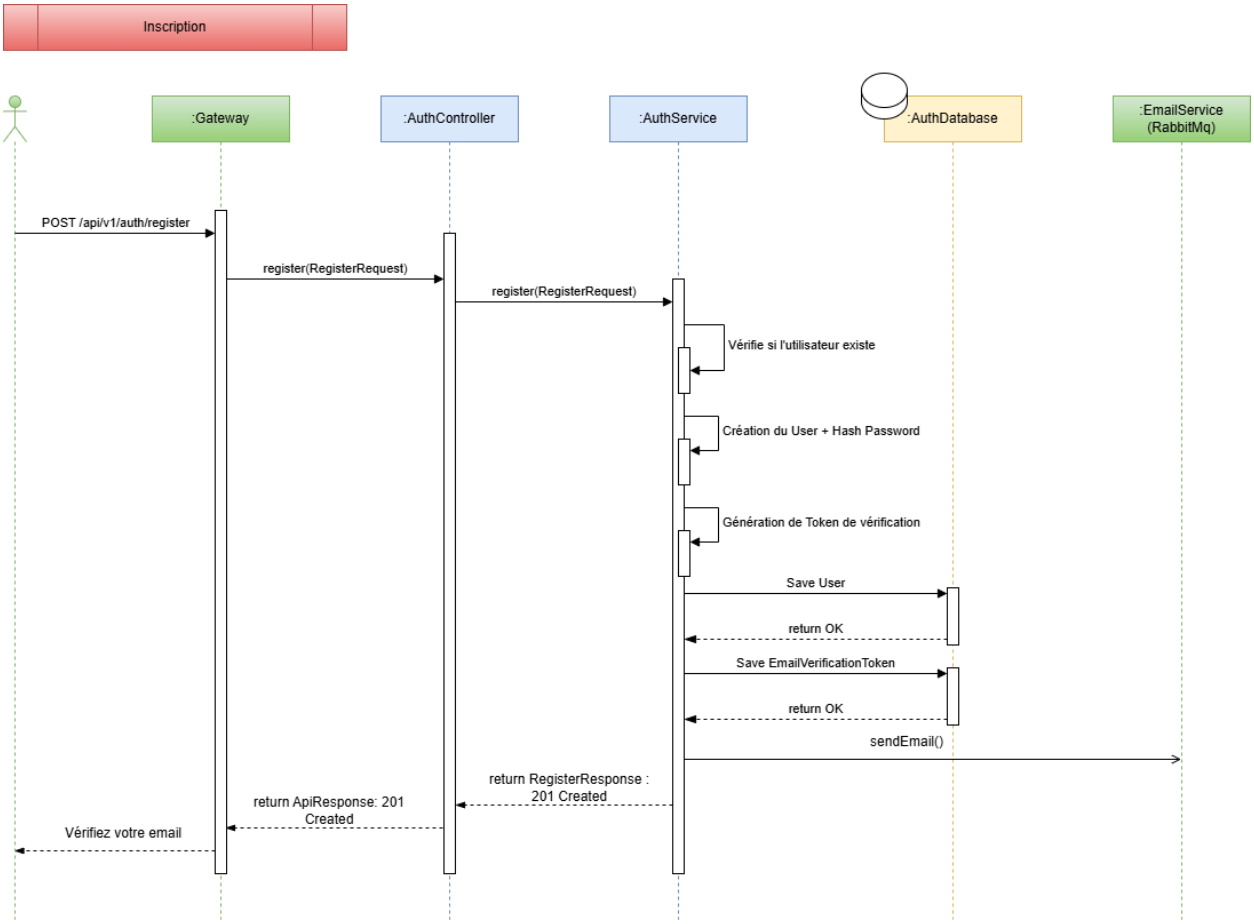


Figure 3 Diagramme de séquence (Auth-Service)

C. Modélisation de la base de données

Dans une application Web, lorsque nous sommes amenés à manipuler des informations, il est primordial d'utiliser une méthode pour la conception d'une base de données.

Dans mon application, j'ai décidé d'utiliser la méthode MERISE qui une méthode française d'analyse, de conception et de réalisations de systèmes d'informations.

Cette méthode se repose sur une séparation claire de 3 niveaux : niveau conceptuel, niveau logique et niveau physique.

Dans notre cas, nous avons tout d'abord une étude préliminaire dans laquelle on identifie les besoins et on définit les objectifs du projet.

1) Dictionnaire des données

En complément du diagramme de classe, nous avons mis en place un **dictionnaire des données** ci-dessous qui va nous permettre de mettre en place les modèles qu'on verra après.

Ci-dessous, vous avez le dictionnaire de données pour le service d'authentification :

Code mnémonique	Désignation	Type	Taille	Attributs	Remarques
id_user	Identifiant numérique d'un utilisateur	N		unsigned	Clé primaire AUTO INCREMENT
email	Email d'un utilisateur	A	100		NOT NULL, UNIQUE
password	Mot de passe d'un utilisateur	A	255		NOT NULL
is_verified	Boolean d'un compte vérifié	Booléen			DEFAULT(FALSE)
is_account_banned	Boolean d'un compte banni	Booléen			DEFAULT(FALSE)
auth_user_id	Identifiant métier de user	A			NOT NULL, UNIQUE
roles	Roles d'un utilisateur	ENUM			DEFAULT(["USER"])
created_at	Date de création d'un compte utilisateur	DateTime			
updated_at	Date de mise à jour d'un compte utilisateur	DateTime			
last_login_at	Date de dernière connexion	DateTime			
id_session	Identifiant numérique d'une session	N		unsigned	Clé primaire AUTO INCREMENT
device_name	Nom de device	A	100		NOT NULL
ip_address	Adresse IP	A			Regex, NOT NULL
user_agent	Agent utilisateur	AN			NOT NULL
refresh_token_hash	Hash du refresh Token	A	255		NOT NULL
created_at	Date de création d'une session	DateTime			
last_used_at	Dernière utilisation d'une session	DateTime			
expires_at	Date d'expiration	DateTime			
revoked_at	Date de révocation	DateTime			
is_revoked	Boolean d'une révocation	Booléen			DEFAULT(FALSE)
id_user	Identifiant numérique d'un utilisateur	N		unsigned	Clé étrangère vers User
id_email_verification	Identifiant numérique d'une vérification d'email	N		unsigned	Clé primaire AUTO INCREMENT
token	Token de vérification d'email	A	255		NOT NULL, UNIQUE
expires_at	Date d'expiration du Token	DateTime			
used	Boolean d'un token utilisé	Booléen			DEFAULT(FALSE)
id_user	Identifiant numérique d'un utilisateur	N		unsigned	Clé étrangère vers User
id_password_reset	Identifiant numérique d'un Reset Password	N		unsigned	Clé primaire AUTO INCREMENT
token	Token de reset Password	A	255		NOT NULL, UNIQUE
expires_at	Date d'expiration du Token	DateTime			
used	Boolean d'un token utilisé	Booléen			DEFAULT(FALSE)
id_user	Identifiant numérique d'un utilisateur	N		unsigned	Clé étrangère vers User

Figure 4 Dictionnaire des données (Auth-service)

2) Modèle Conceptuel de Données (MCD)

Suite à cela, on met en place un modèle qui représente les données qu'on appelle **Modèle conceptuel des données** (MCD).

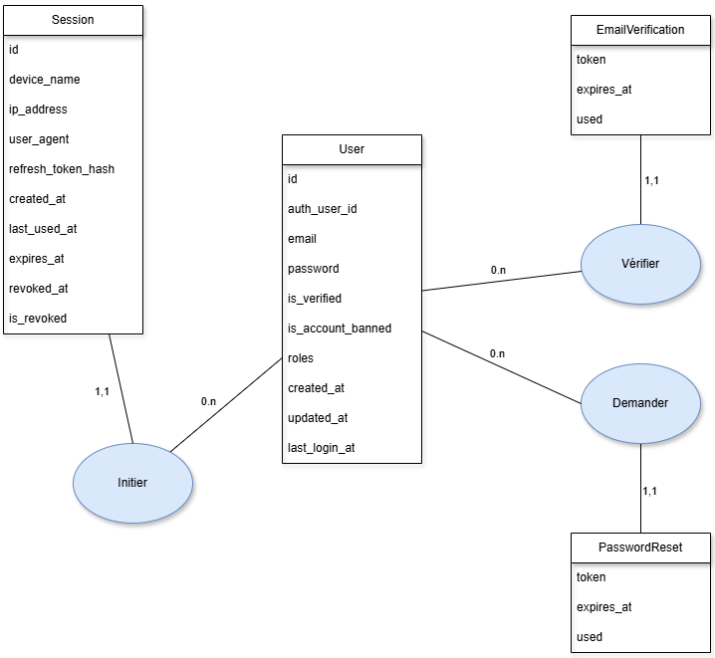


Figure 5 MCD (Auth-Service)

Dans cet exemple, nous n'avons que des associations OneToMany entre les entités.

Dans un autre projet effectué **OurEvent**, j'avais une association ManyToMany :

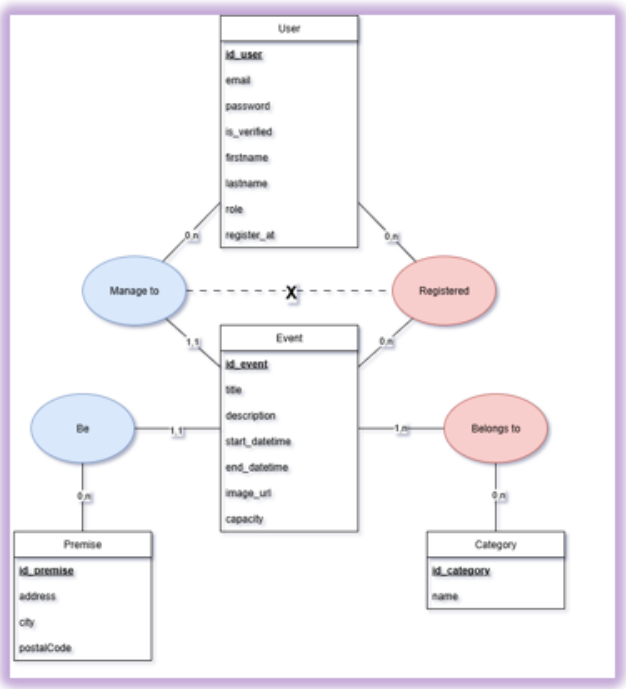


Figure 6 MCD (Projet OurEvent)

3) Modèle Logique de Données (MLD)

Le MLD est une étape clé de la conception de notre base de données puisqu'elle permet de traduire le MCD en un modèle plus détaillé qui prend en compte la structure logique de nos données.

Dans ce schéma, on aura les relations entre les différents objets qui sont représentés par des tables. De plus, c'est dans ce schéma que seront définis les clés primaires, mais aussi les clés étrangères résultant des relations.

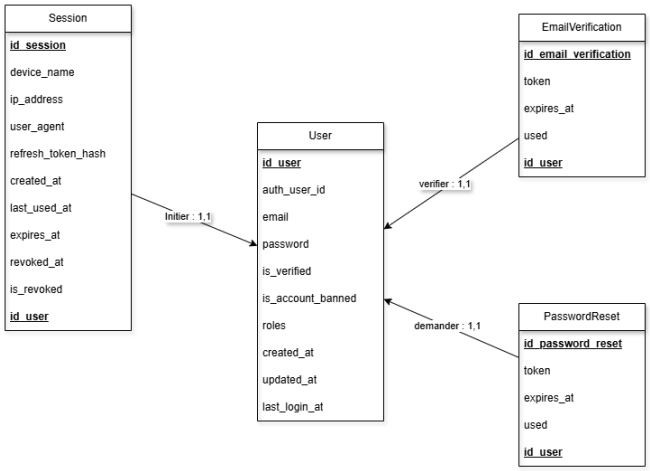


Figure 7 MLD (Auth-Service)

Comme précédemment, voici le MLD du projet **OurEvent** avec une relation ManyToMany qui va donner cette fois-ci une table associative qui contient les 2 clés étrangères.

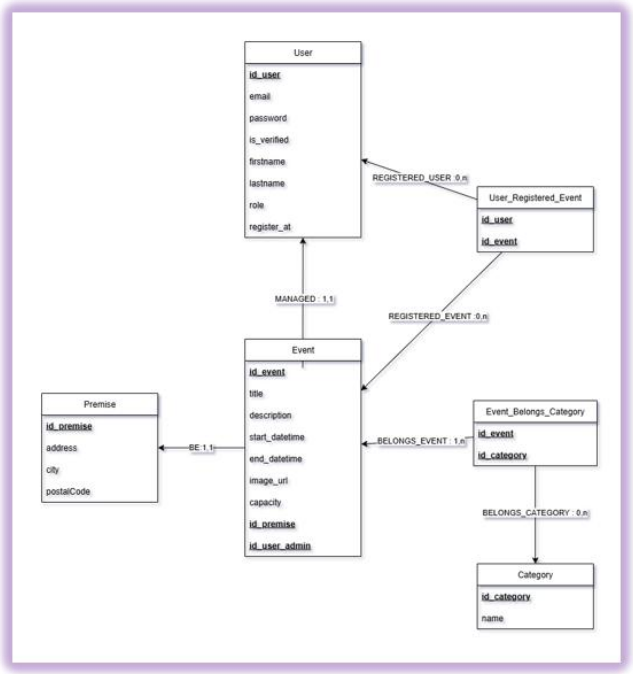


Figure 8 MLD (Projet OurEvent)

4) Modèle Physique de Données (MPD)

Le MPD est la modélisation qui se rapproche le plus du physique donc il varie selon le SGBD choisi, c'est pour cela qu'on spécifie les types de données.

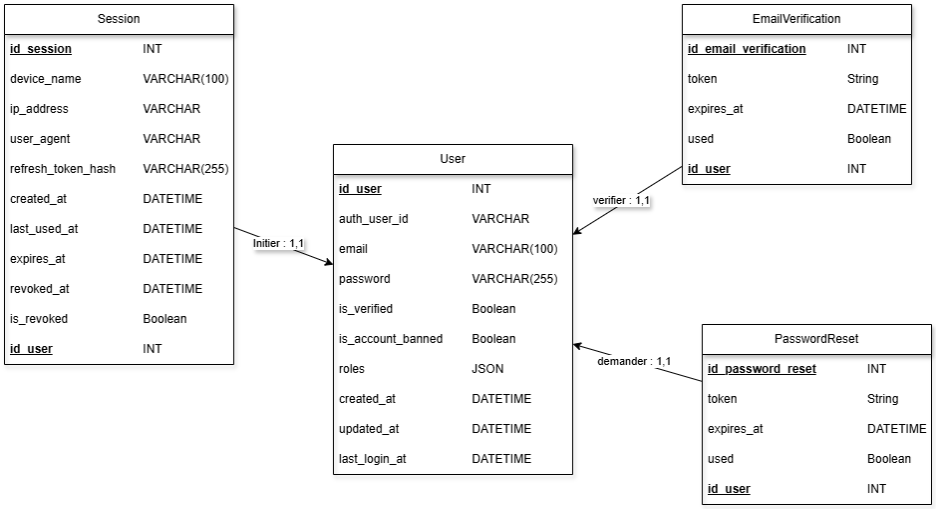


Figure 9 MPD (Auth-Service)

Comme précédemment, voici le MLD du projet **OurEvent** :

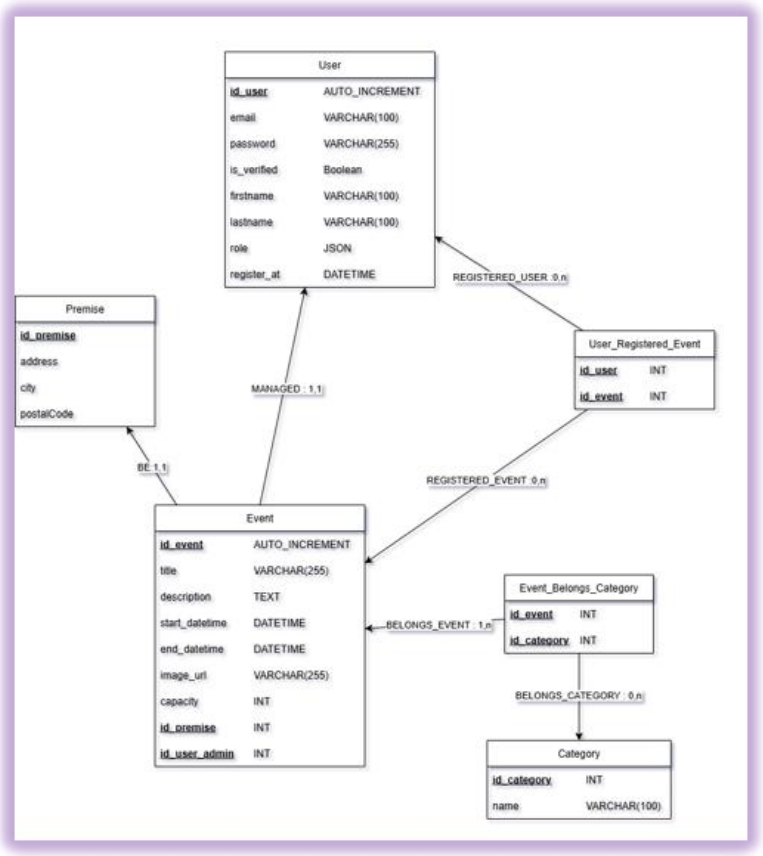


Figure 10 MPD (Projet OurEvent)

V- Architecture multicouche et bonnes pratiques

A. Choix technologiques

Pour cette application, j'ai décidé de partir sur une architecture micro service pour le Back End basée sur l'écosystème Spring Cloud.

Pour la partie Front End, j'ai décidé de partir sur une architecture modulaire orientée fonctionnalités.

B. Architecture Micro services

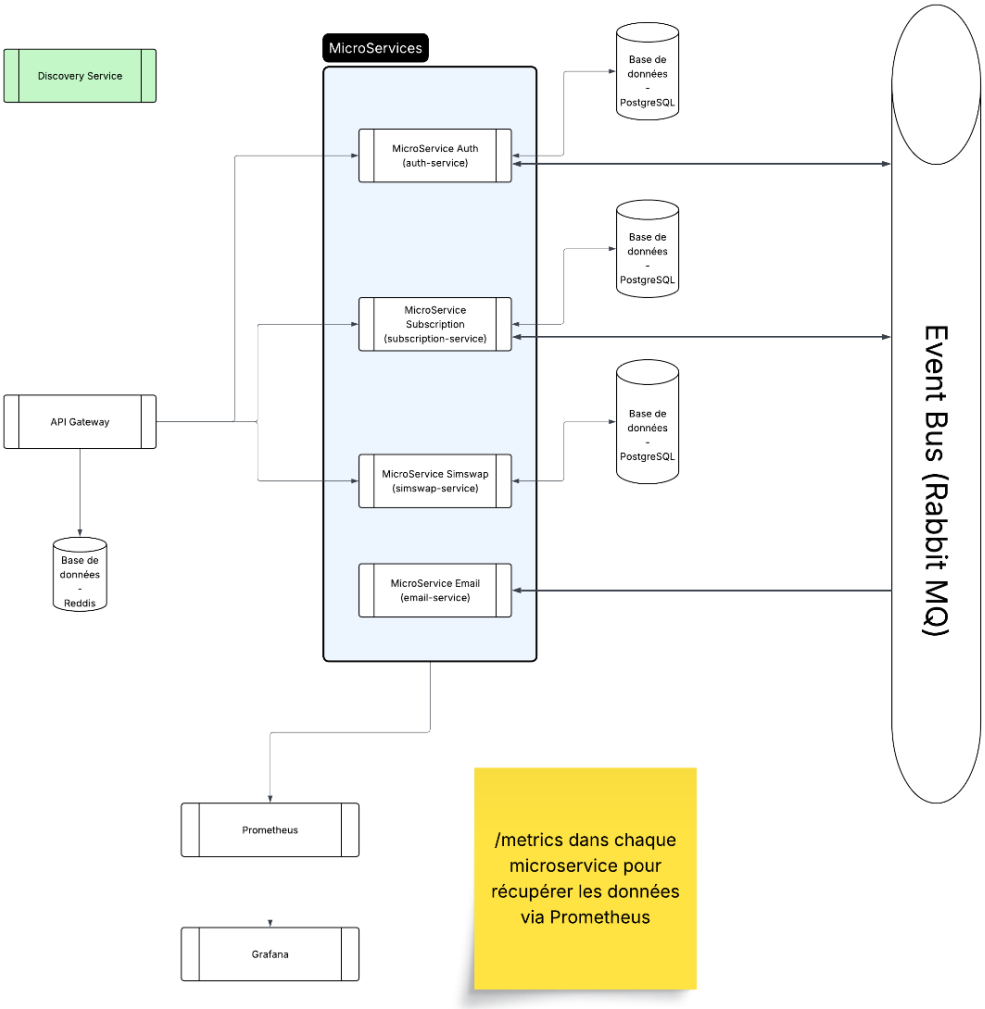


Figure 11 Architecture MicroServices

Pour ce qui est de l'architecture, j'ai décidé de séparer mon back End en 6 micro services :

- **Gateway-service :** Il s'agit du point d'entrée unique de notre application.

Ce service est développé sous Spring Cloud Gateway lui permettant de gérer le routage dynamique vers les différents services, la gestion des en-têtes CORS.

De plus, en intégrant des filtres, ce service nous permet de valider les tokens de sécurité ou encore de vérifier les rôles présents dans le token avant de distribuer la requête vers les services internes.

Ce service nous permet aussi d'ajouter un Rate Limiter et un système de cache nous permettant de garantir une meilleure sécurité et de meilleures performances.

- **Discovery-service :**

Pour une application micro services qui est souvent amenés à avoir une scalabilité et donc des services à être démarré et stoppé, il est essentiel de permettre à ces services de communiquer entre eux sans connaitre leurs adresse IP fixes. C'est pour cela que la mise en place de Discovery Service était essentielle puisqu'elle permet à chaque service de s'enregistrer auprès de cet annuaire au démarrage permettant d'être localise dynamiquement.

Cet annuaire est mis en place grâce à Netflix Eureka qu'on active via son annotation :

```java
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(DiscoveryServiceApplication.class, args);
    }

}
```

Figure 12 Activation du Discovery-Service

- **Auth-service :**

Dans ce service, j'ai fait le choix de rassembler 2 services : Auth-service et User-service.

En effet, l'application n'est pas conséquente pour avoir à séparer ces 2 éléments ce qui serait une bonne pratique d'avoir un service qui gère tous ce qui concerne l'authentification et un service qui gère les informations des utilisateurs.

Ainsi, ce service va gérer l'inscription, l'authentification, la déconnexion et tous ce qui concerne les changements de mot de passe.

Pour l'authentification, puisqu'on est dans une architecture microservice, j'ai opté pour une approche **Stateless** avec la mise en œuvre de **JWT** (JSON Web Tokens) dans un système de double jeton avec un mécanisme de rafraichissement (**Access Token** et **Refresh Token**).

- **Subscription-service :**

Ce service constituera le cœur de l'activité commerciale de mon application.

Tout comme le service précédent, j'ai fait le choix de ne pas séparer les services d'abonnement et de paiement. Ainsi, j'ai tout réuni en 1 seul service pour éviter d'ajouter des difficultés sans en avoir de réels besoins.

Son rôle est de gérer les abonnements d'un utilisateur mais aussi de gérer les paiements via Stripe.

- **Simswap-service :**

Ce service est l'élément principal de mon application qui s'occupe de répondre au service proposé.

Ce service va gérer la gestion, la vérification du quota selon l'abonnement avant de faire appel à la fonctionnalité principale de mon application.

- **Email-service :**

Ce service permet de centraliser l'ensemble des notifications et il est responsable de l'envoi des emails qu'ils s'agissent de vérification de compte ou de confirmation d'abonnement.

Ce service est conçu pour être asynchrone pour ne pas bloquer le thread principal lors de l'envoi de l'email.

C. Architecture Angular

Pour l'architecture du Front End, j'ai décidé de poursuivre vers un Architecture modulaire orientée fonctionnalités qui respecte les recommandations officielles Angular.

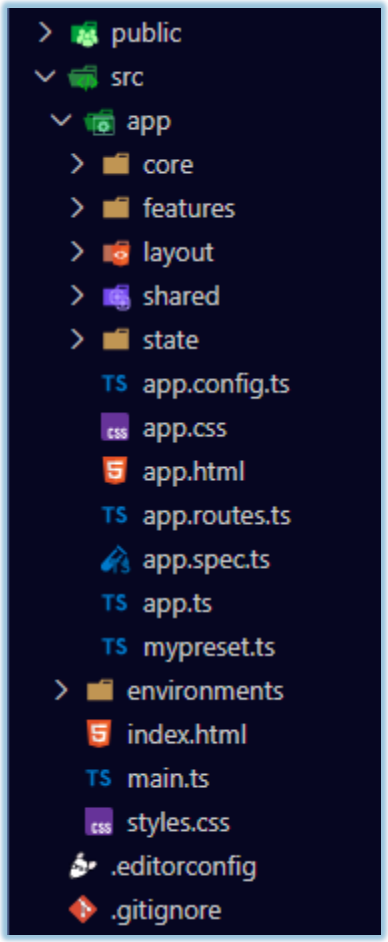


Figure 13 Architecture de dossier sous Angular

Cette architecture modulaire est une solution qui permet une séparation stricte des responsabilités, garantissant ainsi la maintenabilité et la scalabilité. Cette organisation en couches est composée des dossiers :

- **Core/ :** Ce répertoire correspond au cœur du système qui centralise tout ce qui ne doit être instancié qu'une seule fois dans l'application comme les Intercepteurs http, les Guards et les Services Globaux.
- **Features/ :** Ce répertoire correspond à la logique métier de mon application côté Front End. Chaque dossier dans celui-ci correspond à un domaine fonctionnel du projet possédant ses propres composants, modèles et services dédiés.
- **Shared/ :** Ce répertoire regroupe l'ensemble des éléments réutilisables pour optimiser le développement et utiliser la puissance du Framework.
- **Layout/ :** Ce dossier contient les composants de structure de notre application comme le Header, le Footer.

D. Design Patterns et Bonnes Pratiques

1) Injection de dépendances & Inversion de Contrôle

Ces 2 éléments sont les bases de mon architecture que ça soit sur le Back End avec Spring boot ou sur le Front End avec Angular. Ils s'agissent de pattern qui permet de déléguer la création et la gestion du cycle de vie des objets aux Frameworks, sans avoir à les instancier manuellement.

Coté Spring boot :

Pour le Framework Spring boot, voici l'exemple ci-dessous de l'utilisation de l'injection de Dépendances par constructeur qui est recommandée. Il est essentiel de dire que j'utilise aussi la dépendance « **Lombok** » qui permet d'éviter d'ajouter le constructeur dans le code grâce à ses annotations (notamment l'annotation **@RequiredArgsConstructor**).

```java
@Tag(name = "Authentication", description = "Endpoints pour la gestion de l'authentification, inscription et sessions utilisateur")
@RestController
@RequestMapping("/api/v1/auth")
@RequiredArgsConstructor
public class AuthController {
    private final AuthService authService;

    ...
}
```

Figure 14 Utilisation d'annotation Lombok

Le conteneur **IoC** (Inversion of Control) de Spring gère les composants (**Beans**) et les injecte automatiquement, ce qui découple la logique métier de l'instanciation technique

Ces 2 éléments sont essentiels pour faciliter et optimiser le développement de l'application, en facilitant aussi le développement des tests en permettant de « mocker » facilement les dépendances.

Côté Angular :

Le Framework Angular utilise un système d'injecteur hiérarchique, c'est-à-dire que si un composant demande un service, Angular le cherche d'abord localement, puis dans ses parents pour enfin atteindre l'injecteur root.

Grâce aux décorations de classe proposé par Angular (comme **@Injectable({ providedIn: 'root' })**), je m'assure que mes services utilisent le **Design Pattern Singleton** et qu'ils soient

disponibles dans toute l'application, permettant d'optimiser la mémoire et la cohérence des données.

```
import { HttpClient } from "@angular/common/http";
import { inject, Injectable } from "@angular/core";
import { Router } from "@angular/router";

@Injectable({
    providedIn: 'root',
})
export class AuthService {
    private readonly http                   = inject(HttpClient);
    private readonly router                 = inject(Router);

    ...
}
```

Figure 15 Décorateur de classe Angular pour l'injection

2) Design Pattern (Annotations)

Une des puissances du Framework Spring boot est l'utilisation des annotations qui est l'implémentation moderne de plusieurs Design Patterns.

Par exemple en utilisant les annotations @Service ou @RestController dans Spring, on enrichit les comportements des composants avec les Pattern **Proxy** & **Singleton**.

```
import org.springframework.web.bind.annotation.RestController;

@RestController
public class AuthController {
    ...
}
```

Figure 16 Annotation pour les contrôleurs

3) Séparation des responsabilités (DTOs et Services)

Dans mon application Spring, pour un respect optimisé de la conception en couches, j'ai décidé de mettre en place une séparation stricte entre la persistance, la logique métier et la présentation.

- Couche Service

Cette couche correspond à la logique métier qui sert de pont entre les contrôleurs et l'accès aux données via les Repositories.

Cette couche est essentielle pour pouvoir tester indépendamment la logique métier.

- Pattern DTO (Data Transfer Object)

L'utilisation de cette couche est une bonne pratique qui essentielle pour la sécurité et la performance. En effet, elle permet de ne pas exposer les entités de la base de données et de filtrer les champs sensibles.

E. Implémentation des fonctionnalités clés

1) Gestion des abonnements et intégration de l'API Stripe

Avant d'utiliser le service principal de mon application, j'ai mis en place un système d'abonnement proposant 3 abonnement (Hebdomadaire, Mensuel et Annuel).

Figure 17 Affichage des prix d'abonnement

Une fois que l'abonnement est choisi par le client, il est redirigé vers Stripe pour passer au paiement. J'ai décidé de ne pas intégrer Stripe dans mon application pour fournir au client au client une sensation de sécurité. En effet, un client a plus de confiance et de facilité à payer lorsqu'il complète le formulaire dans un site qu'il connait.

Une fois que le paiement est effectué, je reçois la requête Stripe me permettant de mettre à jour mes données. En effet, au préalable, je dois exposer un Endpoint pour le Webhook Stripe.

```java
@Service
@RequiredArgsConstructor
@Slf4j
public class PaymentWebhookService {
    private final UserSubscriptionService userSubscriptionService;
    private final PaymentService paymentService;

    public void handleStripeEvent(Event event) {
        log.info("Traitement webhook Stripe: {}", event.getType());

        switch (event.getType()) {
            case "checkout.session.completed":
                handleCheckoutSessionCompleted(event);
                break;
            case "payment_intent.succeeded":
                handlePaymentIntentSucceeded(event);
                break;
            case "payment_intent.payment_failed":
                handlePaymentIntentFailed(event);
                break;
            default:
                log.info("Type d'événement non géré: {}", event.getType());
        }
    }

    ...
}
```

Figure 18 Service Webhook de Paiement Stripe

Selon l'évènement et le résultat du paiement, je redirige vers le service correspondant.

Une fois que l'abonnement activée, Stripe s'occupe de gérer le paiement automatique selon l'abonnement pris. Ainsi, comme pour le paiement, il est important d'exposer aussi un Endpoint pour le Webhook Stripe, qui va s'occuper de la partie abonnement et donc se synchroniser avec nos données.

```java
@Service
@RequiredArgsConstructor
@Slf4j
public class SubscriptionWebhookService {
    private final UserSubscriptionRepository subscriptionRepository;

    public void handleStripeEvent(Event event) {
        log.info("Traitement webhook Stripe pour abonnement: {}", event.getType());

        switch (event.getType()) {
            case "customer.subscription.created":
                handleSubscriptionCreated(event);
                break;
            case "customer.subscription.updated":
                handleSubscriptionUpdated(event);
                break;
            case "customer.subscription.deleted":
                handleSubscriptionDeleted(event);
                break;
            case "invoice.paid":
                handleInvoicePaid(event);
                break;
            case "invoice.payment_failed":
                handleInvoicePaymentFailed(event);
                break;
            default:
                log.info("Type d'événement non géré: {}", event.getType());
        }
    }

    ...
}
```

Figure 19 Service Webhook d'abonnement Stripe

2) Service de notification (Emails)

Dans une application micro service, gérer l'envoi d'email dans chaque service peut vite devenir pénible, créer de la duplication de code et aussi monopoliser un thread d'un service.

Pour cela, il est très important de centraliser l'envoi d'email dans un service qui lui est dédié et dont le traitement sera effectué de manière **asynchrone**. Faire de l'asynchrone va immédiatement libérer le thread.

De plus, cela permet de ne pas avoir d'impact sur les performances globales du système s'il y a une certaine lenteur du serveur SMTP.

Pour gérer le traitement asynchrone, j'ai utilisé un Message Broker qui s'appelle **RabbitMQ** qui va jouer le rôle de file d'attente.

En effet, d'un côté, j'ai le service « **Producer** » dans lequel l'évènement survient, de l'autre côté, j'ai le service Email-Service « **Consumer** » qui écoute la file d'attente et entre ces 2 services, j'ai la file d'attente RabbitMQ « **Queue** » qui stocke le message

F. Gestion des Exceptions et Erreurs

Puisque nous sommes dans une application micro services, il est essentiel d'avoir une gestion des erreurs centralisés dans chaque services et harmonisée entre les services pour faciliter le débogage et avoir une expérience utilisateur cohérente.

Centralisation avec le Pattern « Controller Advice » :

Pour éviter d'avoir un code dupliqué et garantir une réponse d'erreur JSON identique, j'utilise l'annotation **@RestControllerAdvice**.

```java
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class GlobalExceptionHandler {
    ...
}
```

Figure 20 Fonction de centralisation d'exception + Annotation

Le code ci-dessous est un **GlobalExceptionHandler** qui intercepte les exceptions spécifiques puis les encapsule dans un objet **ApiResponse** via la fonction **buildResponse**. Qui construit la réponse avec un code d'erreur, un message clair et d'autre informations.

```java
@Slf4j
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<ApiResponse<Void>> handleIllegalArgument(IllegalArgumentException exception, HttpServletRequest request)
    {
        return buildResponse(HttpStatus.BAD_REQUEST, "VALIDATION_ERROR", exception.getMessage(), request);
    }

    @ExceptionHandler(IllegalStateException.class)
    public ResponseEntity<ApiResponse<Void>> handleIllegalState(IllegalStateException exception, HttpServletRequest request)
    {
        return buildResponse(HttpStatus.FORBIDDEN, "BUSINESS_RULE_VIOLATION", exception.getMessage(), request);
    }

    @ExceptionHandler(SecurityException.class)
    public ResponseEntity<ApiResponse<Void>> handleSecurity(SecurityException exception, HttpServletRequest request)
    {
        return buildResponse(HttpStatus.UNAUTHORIZED, "AUTHENTICATION_ERROR", exception.getMessage(), request);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ApiResponse<Void>> handleGeneral(Exception exception, HttpServletRequest request)
    {
        log.error("Erreur non gérée", exception);
        return buildResponse(HttpStatus.INTERNAL_SERVER_ERROR, "INTERNAL_ERROR", "Une erreur interne est survenue", request);
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ApiResponse<Void>> handleValidationException(MethodArgumentNotValidException exception, HttpServletRequest
request)
    {
        String errorMessage = exception.getBindingResult()
                .getFieldErrors()
                .stream()
                .map(FieldError::getDefaultMessage)
                .findFirst()
                .orElse("Requête invalide");

        return buildResponse(HttpStatus.BAD_REQUEST, "VALIDATION_ERROR", errorMessage, request);
    }

    private ResponseEntity<ApiResponse<Void>> buildResponse(HttpStatus status, String code, String message, HttpServletRequest request) {
        ApiResponse<Void> response = ApiResponse.<Void>builder()
                .timestamp(LocalDateTime.now().toString())
                .status(status.value())
                .success(false)
                .code(code)
                .message(message)
                .path(request.getRequestURI())
                .build();
        return new ResponseEntity<>(response, status);
    }
}
```

Figure 21 Gestion des Exceptions

Résilience et Mécanisme de Fallback (Circuit Breaker) :

Lorsqu'on adopte une architecture micro service, nous sommes dans l'obligation de garantir la disponibilité du système même lorsqu'un service est défaillant. Pour cela, on utilise le **pattern Circuit Breaker** avec **Resilience4j** au niveau de la Gateway.

```yaml
resilience4j:
  circuitbreaker:
    instances:
      authServiceCircuitBreaker:
        registerHealthIndicator: true
        slidingWindowSize: 10
        minimumNumberOfCalls: 5
        permittedNumberOfCallsInHalfOpenState: 3
        automaticTransitionFromOpenToHalfOpenEnabled: true
        waitDurationInOpenState: 5s
        failureRateThreshold: 50
        eventConsumerBufferSize: 10
  # Configuration Resilience4j Time Limiter
  timelimiter:
    instances:
      authServiceCircuitBreaker:
        timeoutDuration: 3s
```

Figure 22 Configuration des résiliences dans la gateway

La configuration de la Résilience se fait au niveau du fichier **application.yml** et permet par exemple de limiter le temps d'exécution des appels distants (3 secondes dans notre cas), de détecter un taux d'échec élevé et de couper automatiquement les appels vers un service défaillant afin de garantir la stabilité de l'application.

VI- Sécurisation de l'application et des Données

Lorsqu'on fait la conception d'un Système, il est primordial de penser à la sécurité de notre système. Enfin dans le monde numérique actuel, la sécurisation des systèmes d'information reste un enjeu crucial dans lequel nous avons une évolution constante des menaces. Ainsi pour répondre à ces attentes et se prémunir face à ces risques, des organismes ont vu le jour telles que **l'OWASP** (Open Web Application Security Project) et **l'ANSSI** (Agence nationale de la sécurité des systèmes d'information).

L'OWASP est une communauté en ligne à but non lucratif qui se consacre à la sécurité des applications Web. L'un des contenus qu'elle propose est le « Top 10 » qui est un rapport qu'elle met à jour régulièrement présentant les 10 risques les plus critiques en matière de sécurité des applications Web.

La dernière mise à jour du **Top 10** date de cette année 2025 :

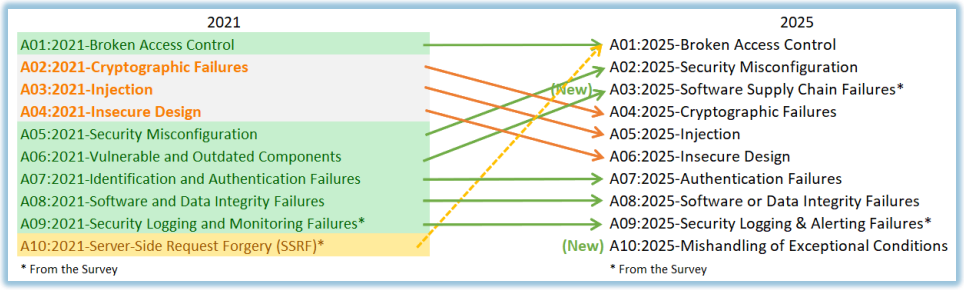


Figure 23 Top 10 OWASP 2025

Dans le TOP 10, on y trouve en 1ère position la défaillance du contrôle d'accès, ce qui a fait que j'ai décidé d'implémenter une Gateway et une sécurité via token JWT.

En 3ème position, on a la défaillance de la chaîne d'approvisionnement logicielle, qui m'a permis de prendre en compte l'importance de la sécurité et vérification dans une CI/CD.

Enfin, la dernière concerne l'importance de la gestion des exceptions.

L'ANSSI quant à elle est un service français créé par décret en 2009. Elle offre donc un cadre national pour la protection des systèmes d'information en fournissant des **bonnes pratiques**, des recommandations mais aussi des certifications.

Elle propose **10 bonnes pratiques** pour se protégeait de la plupart des risques numériques, voici quelques-unes :

- Gestion du mot de passe : Mot de passe différent pour chaque accès, mot de passe complexe, ne pas le communiquer à un tiers.
- Mis à jour régulier
- Protection face au virus et logiciels malveillants
- Evitez les réseaux publics

A. Injection SQL

Les Injections SQL, c'est quoi ?

Il s'agit d'une faille de sécurité présents dans les applications Web, qui se produit lorsqu'une personne mal intentionnée parvient à insérer ou manipuler des requêtes SQL dans une base de données par le biais d'un champ de saisie utilisateur ou par Url (méthode GET).

Cette faille permettrait à l'attaquant de contourner les contrôles d'accès pour pouvoir corrompre des données ou encore pour récupérer des données sensibles venant de la base de données.

Comment s'en protégeait ?

Pour se protégeait de cette faille, nous devons utiliser des **requêtes préparées**. La préparation des requêtes SQL est une technique qui se déroule en 2 étapes :

- La Préparation :

Cette étape consiste à tout d'abord envoyer la structure de la requête à la base de données en remplaçant les données dynamiques provenant du côté Client par des paramètres. Cette étape de préparation peut être divisée en 2 phases :

- Phase de **préparation** dans laquelle on construit la requête en spécifiant sa structure.
- Phase de **compilation** dans laquelle on vérifie la syntaxe de notre requête par le serveur.

- L'exécution :

Après avoir préparé la requête SQL, on passe à l'exécution de la requête qui consiste à lier nos valeurs provenant du coté client aux paramètres présents dans les requêtes préparées.

Cette approche permet donc de répondre au 5eme risque du Top 10.

Comment se passe la protection dans mon projet ?

Dans mon projet, je n'ai pas eu besoin d'avoir des requêtes SQL personnalisés. Pour cela, j'ai délégué la gestion de la persistance à **Spring Data JPA**.

En effet, en utilisant les interfaces **JpaRepository**, Spring Data JPA via son implémentation **Hibernate** utilise de manière automatique des **Prepared Statements**.

B. Injection XSS

Il s'agit d'une faille de sécurité qui se produit lorsqu'une personne mal intentionnée parvient à injecter du code Javascript malveillant dans des pages Web. L'injection d'un script malveillant peut amener à la collecte d'informations sensibles.

Il existe 3 types d'attaques XSS :

XSS permanent (stocké) : Dans ce cas, le script malveillant est stocké sur le serveur et est donc exécuté à chaque fois qu'un utilisateur y accède soit en visitant une page ou en effectuant une action côté Serveur.

XSS non permanent (réfléchi) : Ici, le script malveillant est injecté dans une requête HTTP et qu'il est renvoyé par le serveur sans être stocké d'où le fait qu'on dit XSS réfléchi.

Self XSS : Il s'agit d'une attaque XSS où l'utilisateur est trompé, par un attaquant via par exemple de techniques de pishing, en injectant un script malveillant depuis sa machine.

Comment se passe la protection dans mon projet ?

Pour se protégeait de la faille XSS, il existe plusieurs méthodes. Dans mon projet, j'ai utilisé 2 méthodes :

- Validation des Entrées

Pour empêcher d'avoir des injections de scripts, j'ai mis en place une validation au niveau de mes DTO côté Back avec l'utilisation de :

- Jakarta Validation qui propose des annotations pour s'assurer que les formats attendus soient respectés
- Expressions Régulières avec le regex

```java
package com.simswap.auth_service.dtos;

import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Pattern;
import lombok.Data;

@Data
public class RegisterRequest {
    @NotBlank(message = "L'email est obligatoire")
    @Email(message = "Le format d'email est invalide")
    private String email;

    @NotBlank(message = "Le mot de passe est obligatoire")
    @Pattern(
        regexp = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[^A-Za-z0-9]).{8,}$",
        message = "Le mot de passe doit contenir au minimum 8 caractères, une majuscule, une minuscule, un chiffre et un caractère spécial"
    )
    private String password;
}
```

Figure 24 Validation des entrées pour le DTO RegisterRequest

- CSP

Ensuite la 2ème méthode, qui est certainement efficace, est la mise en place du CSP (Content Security Policy) se situant au niveau du service Gateway.

```
spring:
  cloud:
    gateway:
      default-filters:
        - AddResponseHeader=X-Content-Type-Options, nosniff
        - AddResponseHeader=X-Frame-Options, DENY
        - AddResponseHeader=Referrer-Policy, strict-origin-when-cross-origin
        - AddResponseHeader=Permissions-Policy, geolocation=(), camera=(), microphone=()
        - AddResponseHeader=Content-Security-Policy, >
            default-src 'self';
            script-src 'self';
            style-src 'self' 'unsafe-inline';
            img-src 'self' data:;
            font-src 'self';
            connect-src 'self' ${FRONTEND_URL:http://localhost:4200} http://localhost:4200;
            frame-ancestors 'none';
            object-src 'none';
            base-uri 'self';
```

Figure 25 Configuration du CSP dans la gateway

Comme on peut le voir ci-dessus, j'ai mis une configuration qui injecte automatiquement des Headers de sécurité dans chacune des réponse http.

C. Attaque par force brute (Rate Limiter)

Qu'est-ce qu'une attaque par force brute ?

La **CNIL** propose une définition de cette faille :

« Une attaque par force brute (bruteforce attack) consiste à tester, l'une après l'autre, chaque combinaison possible d'un mot de passe ou d'une clé pour un identifiant donné afin se connecter au service ciblé. »

Comment s'en protégeait ?

Pour s'en protéger, il existe plusieurs méthodes :

- REGEX

Très souvent, pour sécuriser ou contrôler un accès, on met en place une authentification par mot de passe qui reste un des moyens les plus simples et le moins couteux à déployer.

Toutefois, on se trouve souvent face à un niveau de sécurité faible dû à un mauvais choix de mot de passe qui n'est pas assez suffisant pour résister à une attaque par une personne malveillante.

En réponse à cette problématique, la **CNIL** a émis une recommandation en 2017 concernant les mots de passe qui sera par la suite **mis à jour en 2022**.

Cette recommandation est la mise en place d'un « **REGEX** » qui est une séquence de caractères permettant de vérifier le format d'un mot de passe afin de renforcer sa sécurité.

Ainsi pour garantir la robustesse d'un mot de passe, il est essentiel de définir des critères de complexité et de longueur. La CNIL recommande un niveau **d'entropie de 80 bits** pour les mots de passe afin d'assurer leur résistance aux attaques.

La CNIL met donc à disposition **3 exemples de politiques de mot de passe** qui respectent cette entropie, qui sont :

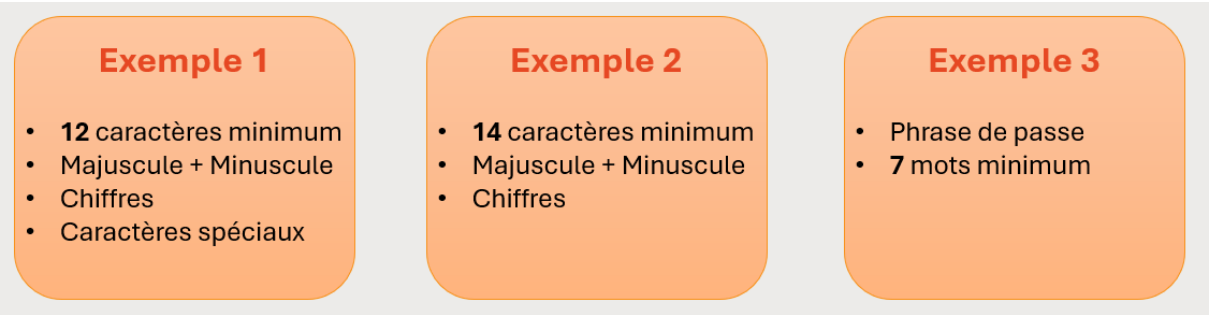


Figure 26 Modèle de mot de passe recommandé par la CNIL

- Rate Limiter

Une autre méthode pour se protégeait contre les attaques de force brute serait de contrôler le nombre de requêtes qu'un utilisateur peut effectuer sur un **service donné** et sur une **période donnée**. Il s'agit donc d'une stratégie de sécurité permettant de **prévenir ces attaques**, qu'on appelle « Rate Limiter ».

Ainsi, cela consistera à limiter le nombre de requête effectué (par exemple, 5 tentatives) sur une période spécifiée (par exemple, 1 minutes), ce qui va réduire la possibilité d'une attaque automatique.

Lorsque cette limitation est dépassée, des mesures de sécurité peuvent être mis en place tel que le blocage du compte ou encore l'ajout de protection supplémentaire tel que l'ajout d'un reCAPTCHA ou encore une double authentification.

Comment se passe la protection dans mon projet ?

Dans mon projet, j'ai mis en place le REGEX à deux reprises : une fois avec Angular pour le Front End et une autre avec Spring boot pour le Back End.

Le regex sur Angular est une mesure de sécurité supplémentaire mais elle me permet surtout d'améliorer l'interface utilisateur (UI) en guidant l'utilisateur sur le choix de son mot de passe de manière dynamique.

- Regex sur Spring boot

Sur Spring boot, je mets en place le regex au niveau du DTO.

```java
package com.simswap.auth_service.dtos;

import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Pattern;
import lombok.Data;

@Data
public class RegisterRequest {
    @NotBlank(message = "L'email est obligatoire")
    @Email(message = "Le format d'email est invalide")
    private String email;

    @NotBlank(message = "Le mot de passe est obligatoire")
    @Pattern(
        regexp = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[^A-Za-z0-9]).{8,}$",
        message = "Le mot de passe doit contenir au minimum 8 caractères, une majuscule, une minuscule, un chiffre et un caractère spécial"
    )
    private String password;
}
```

Figure 27 Regex de mot de passe sur le DTO RegisterRequest

Ce regex peut être décomposé en plusieurs parties :

- **^ et $:** Ils indiquent le début et la fin de la chaîne de caractères pour la vérification
- **(?=.*[a-z]) :** Il s'agit d'une assertion positive vérifiant qu'il y ait au moins une lettre minuscule dans la chaîne de caractère.
- **(?=.*[A-Z]) :** Il s'agit d'une assertion positive vérifiant qu'il y ait au moins une lettre majuscule dans la chaîne de caractère.
- **(?=.*\\d) :** Assertion positive vérifiant qu'il y ait au moins un chiffre dans la chaîne de caractères.
- **(?=.*[^A-Za-z0-9]) :** Assertion positive vérifiant qu'il y ait au moins un caractère spécial.
- **{12,} :** Indique que la longueur du mot de passe doit être d'au moins de 12 caractères.

Une fois que le regex est mis en place, il doit être activé avec l'annotation **@Valid**.

```java
@Tag(name = "Authentication", description = "Endpoints pour la gestion de l'authentification, inscription et sessions utilisateur")
@RestController
@RequestMapping("/api/v1/auth")
@RequiredArgsConstructor
public class AuthController {
    private final AuthService authService;

    @PostMapping("/register")
    public ResponseEntity<ApiResponse<RegisterResponse>> register(@Valid @RequestBody RegisterRequest request) {
        ApiResponse<RegisterResponse> response = authService.register(request);
        return ResponseEntity.status(HttpStatus.CREATED).body(response);
    }
```

Figure 28 Validation du Regex

Regex sur Angular

Pour la validation du Regex, j'utilise la librairie fournie par Angular pour gérer les formulaires.

```typescript
this.registerForm = this.fb.group(
    {
        email: ['', [Validators.required, Validators.email]],
        password: ['', [
                Validators.required,
                Validators.pattern(/^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?\d)(?=.*?[#?!@$%^&*-]).{8,}$/)
            ]
        ],
        confirmPassword: ['', Validators.required],
        acceptTerms: [false, Validators.requiredTrue],
    },
    { validators: [this.passwordMatchValidator] }
);
```

Figure 29 Gestion du Regex via les formulaires Angular

D. Hachage du mot de passe

Lorsqu'on intègre une authentification, il est très important qu'une personne malveillante n'est pas accès aux mots de passe des utilisateurs, cela pourrait avoir des conséquences désastreuses pour les utilisateurs et leur données personnelles.

Pour s'assurer qu'il soit impossible de récupérer un mot de passe en clair, on utilise donc une méthode qu'on appelle « **hachage** » qui consiste à transformer de manière **irréversible** un mot de passe en clair en une chaîne de caractères unique de longueur fixe, appelée **empreinte numérique** à l'aide d'un algorithme de hachage.

Il existe de nombreux algorithmes de hachage, qu'on peut catégoriser dans 2 groupes :

- Algorithmes Forts : Considérés comme très sécurisé. Exemple : **Bcrypt**, Argon.
- Algorithmes Faibles : Considérés comme peu sécurisé. Exemple : MD5, SHA.

Lorsqu'on utilise des fonctions natives JAVA pour le hachage de mot de passe, celui-ci s'occupe d'utiliser automatiquement un algorithme de hachage fort qui est en tendance. Par défaut, il utilise actuellement l'algorithme **Bcrypt**.

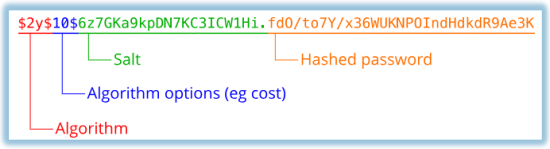


Figure 30 Empreinte numérique après le ahchage du mot de passe avec Bcrypt

Comme nous pouvons le constater, une empreinte numérique est décomposée en 4 parties :

- **Version** de l'algorithme : Elle indique l'algorithme de hachage utilisé et sa version. Dans cet exemple, **$2y$** correspond à une des versions de l'algorithme Bcrypt.
- **Facteur de coût** (Cost) : Elle détermine le nombre de fois que l'algorithme va exécuter son processus de hachage. Ainsi un coût de 10 comme ici signifie que l'algorithme effectuera **2^10 fois** le processus de hachage.
- **Sel** (Salt) : Il s'agit d'une valeur aléatoire qu'on ajoute à notre mot de passe avant son hachage. Ainsi, avec l'ajout du sel, 2 même mots de passe ne peuvent obtenir le même mot de passe haché. De plus, elle permet une protection contre les attaques par table arc-en-ciel, qui sont des tables précalculées de hachages pour des mots de passe courants.
- **Mot de passe haché** : La dernière partie correspond à notre mot de passe haché.

L'ajout d'un Cost et d'un Salt nous permet aussi de protéger contre les attaques par force brute. En effet, plus le facteur de coût est élevé, plus les ressources utilisées et la durée de hachage sont élevées. Ainsi, que le Salt qui complexifie le processus de hachage, rendant la durée de hachage plus élevée.

E. Authentification et Contrôle d'accès

Pour sécuriser les informations des clients et les échanges entre l'interface utilisateur et l'api, j'ai décidé d'implémenter un système d'authentification basé sur les JWT. Ce choix a été fait sur le fait d'utiliser une architecture **Stateless** qui est un indispensable pour une logique de scalabilité et d'application micro services.

Qu'est-ce qu'un jeton JWT (JSON Web Token) ?

Un jeton JWT est une chaine de caractères encodée en Base64. Toutefois sa sécurité ne réside pas sur son encodage puisqu'on peut facilement lire les données mais plutôt sur sa signature.

Figure 31 Description d'un token JWT

Comme on peut le constater ci-dessus, le Token est divisé en 3 parties distinctes :

- **Header :** Il indique le type de jeton et l'algorithme de hachage utilisé pour la signature (HS256).
- **Payload :** A l'intérieur sont stockées les Claims où l'on trouve des données non sensibles comme l'id, email, et rôles, permettant de vérifier les autorisations via la Gateway.
- **Signature :** Celle-ci est générée à partir des 2 premières parties avec une clé secrète en utilisant l'algorithme de hachage.

Pour l'authentification, j'ai mis en place un système à double jeton JWT :

- **Access Token :** Il s'agit d'un jeton utilisé pour chaque appel API. Sa durée de vie est courte (15 minutes).
- **Refresh Token :** Il s'agit d'un jeton stocké dans le local Storage. Sa durée de vie est plus longue. Elle permet de demander le jeton précédent lorsque celui-ci expire, évitant au client de se reconnecter manuellement.

F. CORS

La configuration du CORS (Cross-Origin Resource Sharing) est une étape essentielle pour la sécurité et le bon fonctionnement d'une architecture découplé (Front End / Back End).

Cette configuration s'inscrit dans la lutte du **1er risque du Top 10**.

En effet, j'ai décidé que ces 2 éléments aient des origines différentes. Par défaut, les navigateurs bloquent les requêtes Cross-Origin pour prévenir les scripts malveillants d'accéder à des données sensibles. Pour le permettre, il est important de configurer une politique CORS stricte au niveau du service Gateway comme ci-dessous.

```yaml
spring:
  cloud:
    gateway:
      globalcors:
        cors-configurations:
          '[/**]':
            allowedOrigins:
              - "http://localhost:3000"
              - ${FRONTEND_URL:http://localhost:4200}
            allowedMethods:
              - GET
              - POST
              - PUT
              - DELETE
              - OPTIONS
              - PATCH
            allowedHeaders:
              - "*"
            exposedHeaders:
              - Authorization
              - Content-Type
            allowCredentials: true
            maxAge: 3600
```

Figure 32 Configuration du CORS dans la gateway

A partir du de la configuration, je restreins l'accès à l'URL de mon Front End, limite les interactions aux méthodes http définies, permet au navigateur de lire les Headers et aussi de partager les cookies entre le client et serveur.

VII- Qualité, DevOps et Déploiement

A. Tests applicatifs

Dans une application micro services, il est indispensable de mettre en place une stratégie de tests permettant de garantir la fiabilité de mon code et de prévenir d'éventuels régressions.

Pour obtenir des tests lisibles, maintenables et standardisés, j'ai décidé d'appliquer le pattern AAA qui consiste à séparer les responsabilités dans un test :

- **Arrange :** Cette phase consiste à préparer l'environnement de test (données d'entrée, objets, Mocks)
- **Act :** Cette phase consiste à exécuter la fonctionnalité que l'on souhaite tester
- **Assert :** Il s'agit de la phase de vérification où l'on compare le résultat obtenu au résultat attendu.

Sur Spring boot, les fichiers de Tests doivent être placé dans le dossier Test et les annotations de Test doivent être ajoutés pour permettre au Framework d'organiser de manière optimisé les Tests.

1) Tests unitaires

Comme nous pouvons le constater dans l'exemple ci-dessous qui correspond au test de l'inscription lorsqu'on souhaite créer un nouvel utilisateur quand l'email n'est pas déjà utilisé.

```java
@Test
void register_ShouldCreateNewUser_WhenEmailIsNotUsed() {

    // ARRANGE : Préparation des données
    when(userRepository.findByEmail(registerRequest.getEmail()))
            .thenReturn(Optional.empty());

    when(passwordEncoder.encode(registerRequest.getPassword()))
            .thenReturn("encodedPassword");

    when(userRepository.save(any(User.class)))
            .thenReturn(testUser);

    when(emailVerificationRepository.save(any(EmailVerificationToken.class)))
            .thenReturn(new EmailVerificationToken());

    // ACT : Exécution du service à tester
    ApiResponse<RegisterResponse> response = authService.register(registerRequest);

    // ASSERT : Vérification des résultats
    assertNotNull(response);
    assertTrue(response.isSuccess());
    assertEquals(201, response.getStatus());
    assertEquals("Inscription réussie", response.getMessage());
    assertNotNull(response.getData());
    assertEquals(registerRequest.getEmail(), response.getData().getEmail());

    // Vérification que les méthodes ont bien été appelées
    verify(userRepository, times(1)).findByEmail(registerRequest.getEmail());
    verify(userRepository, times(1)).save(any(User.class));
    verify(emailVerificationRepository, times(1)).save(any(EmailVerificationToken.class));
}
```

Figure 33 Exemple d'un test unitaire du register

2) Tests d'intégration

Contrairement aux tests unitaires, les tests d'intégration vérifient le bon fonctionnement de plusieurs composants ensembles (contrôleur, service, …).

Dans l'exemple ci-dessous, je teste l'endpoint de création de compte de mon API. Je teste l'envoi d'une requête Post en Json avec des données valides et vérifie que le contrôle me retourne bien la réponse attendue.

Ainsi, ce test me permet de confirmer le fonctionnement de mon endpoint, de mon service et de ma validation de donnée.

```java
@Test
void register_ShouldReturn201_WhenDataIsValid() throws Exception {

    // ARRANGE
    RegisterRequest request = new RegisterRequest();
    request.setEmail("newuser@example.com");
    request.setPassword("Motdepasse1234&");

    // ACT & ASSERT
    mockMvc.perform(post("/api/v1/auth/register")
            .contentType(MediaType.APPLICATION_JSON)
            .content(objectMapper.writeValueAsString(request)))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.success").value(true))
            .andExpect(jsonPath("$.status").value(201))
            .andExpect(jsonPath("$.message").value("Inscription réussie"))
            .andExpect(jsonPath("$.data.email").value("newuser@example.com"))
            .andExpect(jsonPath("$.data.message").value(containsString("vérification")));
}
```

Figure 34 Exemple d'un test d'intégration du register

B. Conteneurisation avec Docker

Dans le monde du développement web, mettre une application dans un conteneur Docker est devenue une formalité puisque celle-ci balaye tous les problèmes liés aux machines et au versionnage permettant d'avoir une application fonctionnelle partout.

Il est devenu indispensable de conteneuriser son application surtout lorsqu'il s'agit d'une application micro services. En effet, dans ce genre d'architecture, nous avons besoin que les services soient isolés et qu'ils aient leur propre dépendances et configurations. De plus, utilisé une architecture micro services signifie que nous avons des exigences en termes de scalabilité. Ainsi, docker va nous permettre de démarrer de nouvelles instances plus rapidement et d'avoir une scalabilité beaucoup plus simple.

Pour conteneurisation une application, nous avons besoin de mettre en place un **Dockerfile** qui aura pour rôle de nous générer une image Docker. Dans le cas d'une architecture micro services, nous allons générer une image Docker pour chacun des services, ainsi 1 Dockerfile chacun.

J'ai opté pour du **multi-stage** dont l'objectif est de réduire la taille de l'image finale :

- **Builder** : S'occupe de la phase de compilation du service :
  - On récupère l'image, contenant Maven et un JDK 25, qui deviendra la base de notre build
  - On définit le répertoire de travail
  - On copie le fichier pom.xml dans ce répertoire et on télécharge les dépendances. J'ai décidé de copier seul le fichier pom.xml pour tirer profit du cache Docker puisque les dépendances sont moins amenées à être modifié que le code source.
  - On copie le dossier contenant le code source et on compile le projet sans prendre en compte les Tests puisqu'on les vérifie via le pipeline CI/CD. Cette compilation va donc générer un fichier JAR.
- **Runner** : S'occupe de la phase d'exécution du service :
  - On récupère l'image, contenant uniquement un JRE et Alpine Linux, qui deviendra la base de notre image finale
  - Par mesure de sécurité et selon la recommandation officielle Docker, il est important de créer un utilisateur non root pour éviter une exécution en root.
  - On définit le répertoire de travail
  - On copie le fichier JAR qui se trouve dans le Builder (précédent)
  - On mettre en place la configuration JVM permettant une gestion mémoire dynamique et des diagnostics en cas d'erreur mémoire
  - On donne les droits à l'utilisateur non root
  - On procède à un changement d'utilisateur pour que l'application s'exécute en tant qu'utilisateur non privilégié.
  - On expose le port et on démarre l'application Spring boot.

C. Pipeline CI/CD avec GitHub Actions

1) Pipeline CI

Dans une application, une **pipeline CI** est essentielle pour valider automatiquement le code à chaque modification en cochant une succession d'étape fourni au préalable.

Etant donné que je suis dans une architecture micro services et que j'utilise la même technologie pour chacun des services. J'ai décidé de mettre en place un CI réutilisable qui contient toute la logique CI commune, respectant ainsi le **principe DRY** (Don't Repeat Yourself) et qui sera appelé par chaque micro service.

Dans mon cas, il y a 1 seul job CI qui contient plusieurs étapes appelés "steps" :

- Premièrement, on exécute la CI sur un Runner Linux, dans notre cas Ubuntu.
- Ensuite on récupère la version du code du dépôt GitHub pour pouvoir l'analyser
- Puis on passe à l'installation de Java 25
- On place en cache les dépendances Sonar permettant de réduire le temps d'analyse
- On exécute les Tests et on analyse la qualité du code
- On vérifie la compilation du code sans les tests qui ont déjà été effectués

2) Pipeline CD

Après la validation du code par le pipeline CI, la **pipeline CD** prend le relais pour transformer notre code source en un artefact prêt à l'emploi, push l'image vers le **Docker Hub** et ensuite facultativement le déployer sur un serveur distant.

Tout comme la pipeline CI, j'ai fait le choix de mettre en place un CD réutilisable pour que chacun des services ait la même configuration.

Ma pipeline CD se décompose donc en plusieurs phases :

1) Déclenchement et Filtrage : cette pipeline ne se déclenche seulement si la CI associée est un succès. J'ai intégré un filtrage par branche de tels sorte que la CD ne se poursuivent que pour les codes dites stables (main, release et dev).
2) Gestion dynamique du Tag : Permettre d'avoir un tag ordonné et intelligent sur les images Docker selon la branche d'origine.
3) Optimisation du Build avec Docker **Buildx** : cet outil avec le système de mise en cache Docker nous permet de réduire considérablement les temps de déploiement notamment lors de la reconstruction d'une image.
4) Audit de sécurité avec Trivy : Avant le push de l'image Docker, cette dernière est scannée par Trivy pour détecter des vulnérabilités.
5) Stockage dans le registre (**GHCR**) : Lorsque toutes les étapes précédentes sont validées, l'image est poussée vers le GitHub Container Registry où les images sont stockées de manière centralisée et sécurisée.
6) Enfin facultativement, nous avons le déploiement vers un serveur distant.

D. Documentation & Monitoring

1) Documentation Swagger/OpenAPI

Pour une application Web notamment lorsqu'on a une API, il est essentiel de documenter l'API et pour cela on fait appel à des outils qui nous le permet.

En effet, il existe 2 outils principaux :

- **OpenAPI** : Il s'agit d'une spécification standard permettant de décrire une API Rest de manière formelle et lisible. Son objectif principal est de décrire toutes les routes d'une API et de les documenter (Type de paramètres, réponses envoyées, etc.…). Elle a aussi d'autres fonctionnalités comme la génération de documentation, la validation de l'API, etc ...
- **Swagger** : Il s'agit d'un ensemble d'outils qui vont compléter la spécification OpenAPI. Parmi ces outils, on trouve Swagger UI qui permet d'avoir une interface web interactive pour mieux visualiser l'Api, de pouvoir tester les endpoints ou encore envoyer des requêtes HTTP.

En voyant ces descriptions, on comprend que les 2 outils sont complémentaires et qu'il est fortement recommandé de combiner ces 2 outils.

Dans le Framework Spring boot, il existe une libraire "springdoc OpenAPI" qui fournit une solution complète adapté à une application spring. Cette librairie analyse ton code automatiquement, ce qui lui permet de générer automatiquement un document OpenAPI. De plus, cette librairie intègre l'outils Swagger UI.

Ainsi, cette librairie fournit une documentation sans avoir à faire une configuration.

Comment la librairie Springdoc OpenAPI fonctionne ?

1) Lorsque l'application Spring boot démarre, Springdoc s'active après avoir vérifié la présence de Spring Web et d'élément essentiels pour une API.
2) Ensuite Springdoc s'occupe de générer automatiquement des Beans internes qui sont injectés dans l'application Spring et permettant d'utiliser ses nombreux services.
3) Une fois que Spring s'est occupé d'enregistrer tous ces contrôleurs via les annotations qui lui sont disposés (@RestController, @RequestMapping, @GetMapping, etc.…), Springdoc va s'occuper d'analyser ces annotations pour générer une documentation.
4) Une fois qu'il a analysé les endpoints, il analyse les paramètres de ces endpoints pour pouvoir les interpréter et analyser d'autres métadonnées.

5) Une fois que les analyses sont terminées, springdoc expose une route GET "/v3/api-docs" qui retourne un fichier JSON appelé JSON OpenAPI et qui est généré dynamiquement lors de la requête.
6) Enfin via le Swagger UI présent dans Springdoc, nous avons une route "/swagger-ui/index.html" qui nous permet d'accéder à une page HTML où l'on peut visualiser notre API. Cette page HTML charge les données en faisant un appel à la route "/v3/api-docs".

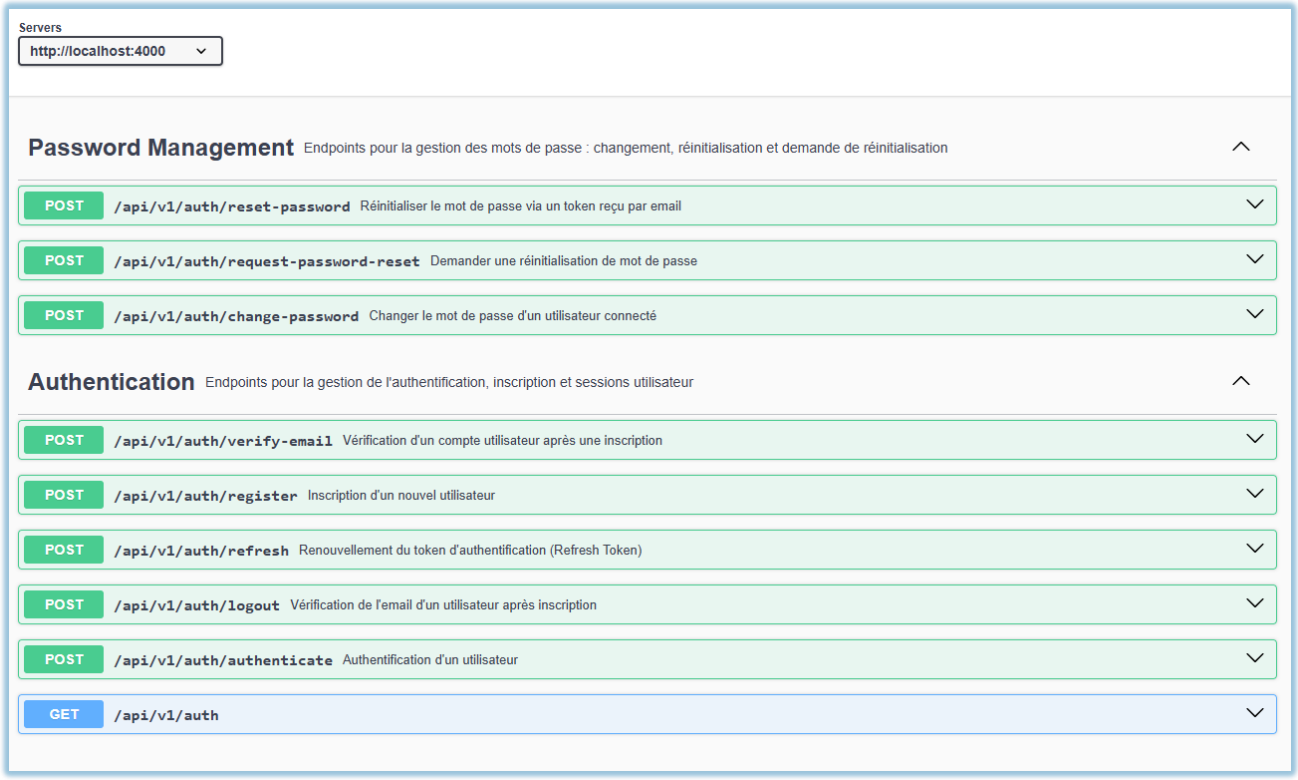


Figure 35 Exemple de la documentation pour Auth-Service

2) Monitoring

Dans une application web, le monitoring est un élément essentiel de la maintenance. En effet, cela permet de voir en temps réel, les métriques de nos serveurs et les logs permettant ainsi de prévenir ou de résoudre des problèmes lors de la phase de maintenance. Il est encore plus indispensable lorsque l'application suit une architecture microservices ou distribués.

C'est pourquoi j'ai décidé de mettre en place une stack complète qui est la **stack "Grafana Observability"**. Celle-ci est composée de :

- Prometheus : Pour les métriques
- Tempo : Pour les traces distribuées
- Loki : Pour les logs
- Grafana : Pour la visualisation et la corrélation.

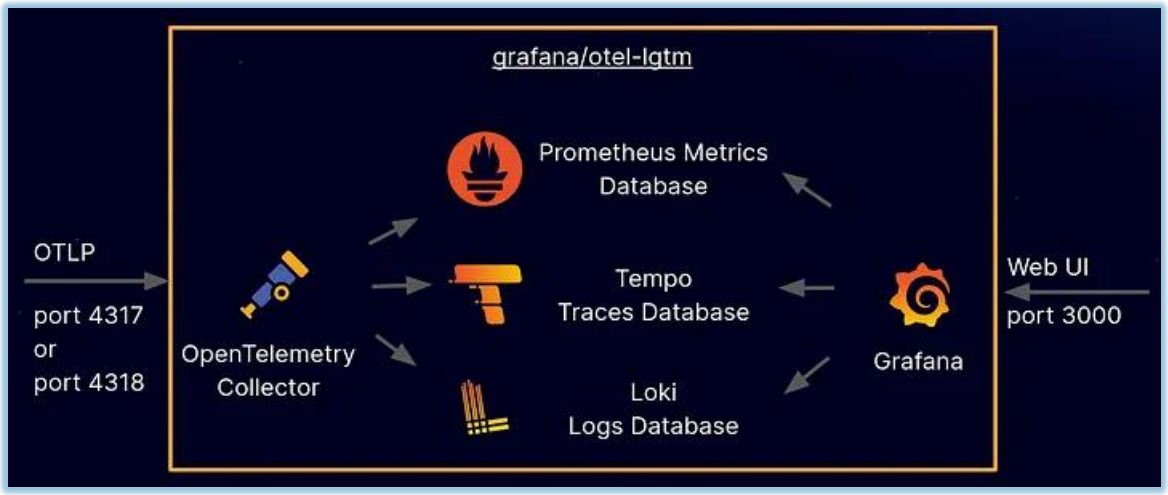


Figure 36 Illustration de la Stack "Grafana Observability"

a- Metrics via Prometheus

Il s'agit d'un système de collecte de métriques permettant de suivre l'état d'un système et stocker ces données sous forme de séries temporelles.

Comment fonctionne Prometheus ?

1) Tout d'abord, pour que Prometheus fonctionne dans notre application, il est essentiel que l'application expose les métriques. Pour cela, il existe des dépendances tels que Actuator (expose des données via des endpoints mais ne les structure pas et ne les stocke pas) et Micrometer (collecte, structure les métriques au format Prometheus et les stockent côté JVM).

2) Une fois que les endpoints et les métriques sont disponibles, Prometheus va fonctionner en mode PULL. En effet, il va interroger périodiquement les applications et ensuite stocker ces données sous forme de time series en les associant à des labels tels que les services, les instances, etc ....

3) Prometheus utilise le langage PromQL pour manipuler les métriques et faire des calculs.

Quel est le rôle de Prometheus ?

1) Détection de pannes
2) Surveillance de la performance et de la charge
3) Alerting en utilisant d'autre outils comme AlertManager.

b- Traces via Tempo

Tempo est un backend de traces distribuées qui permet de suivre une requête de bout en bout. En effet, les logs seules sont loin d'être suffisant pour avoir une traçabilité sur une requête, c'est pour cela que Tempo permet de palier à ce problème.

Avec Tempo, il existe 2 notions très importants à connaitre :

- Trace : Elle représente le cycle de vie complet d'une requête et permet d'être identifié par un TraceId. Une Trace peut contenir plusieurs Spans.
- Span : Il représente une opération unique, un service. Les spans sont organisés hiérarchiquement et chronologiquement permettant ainsi de reconstruire le chemin exact de la requête.

En utilisant la dépendance Micrometer, celle-ci va automatiquement générer des Spans pour :

- Des appels entrants et sortantes
- L'accès à une base de données
- L'accès à des méthodes

Un Span sera défini par :

- TraceId
- SpanId
- Durée
- Statut
- Tags

L'exportation des traces vers Tempo se fait à l'aide du protocole OTLP (HTTP ou gRPC).

c- *Logging via Loki*

Loki est un système de centralisation des logs. En effet, lorsque nous sommes face à une application micro services où chacun des services produit ses propres logs, il est plus judicieux de centraliser tous les logs en 1 seule endroit permettant d'avoir un aperçu global. En plus, d'être un système de centralisation, il est aussi un système de stockage et de recherche de logs.

Pour que Loki soit utiles dans notre application, il est essentiel que cette dernière soit en capacité de générer des logs. Pour cela, j'ai utilisé la dépendance "loki-logback-apender" qui me permettra de générer des logs seulement en utilisant l'annotation Spring @Sflj4. De plus, j'ai décidé de personnaliser mes logs via le document "logback-spring.xml" pour y ajouter des informations de traçabilité et d'identification de service.

Loki utilise le langage LogQL pour faire des recherches de logs.

d- *Grafana*

Enfin, Grafana s'occupe de tout centraliser dans un dashboard comme ci-dessous :

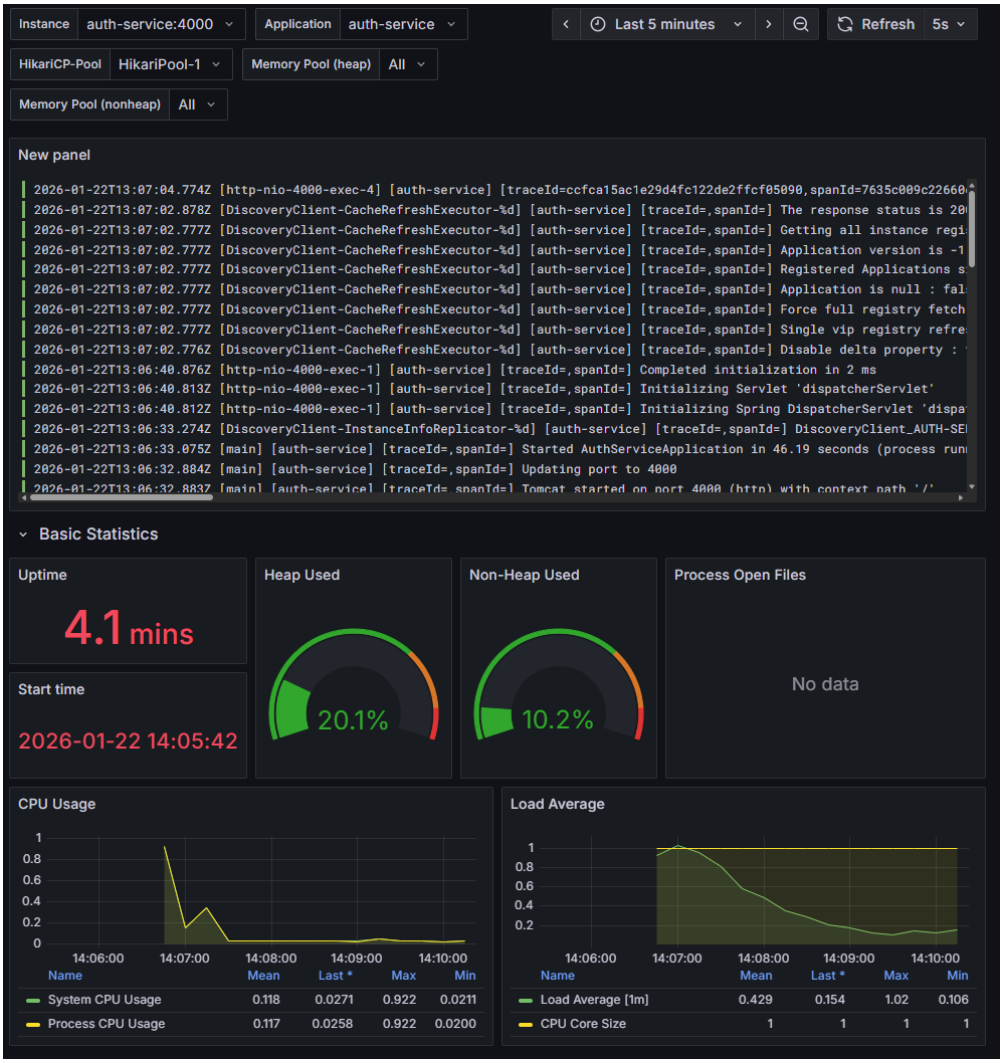


Figure 37 Dashboard Grafana (Logs, Metrics)

VIII- Bilan et Perspectives d'améliorations

Pour conclure, la découverte d'une architecture micro services a été très enrichissant pour ma part. Ma philosophie est que c'est dans la difficulté qu'on progresse. Dans mon cas, ce projet m'a permis de comprendre l'utilité de cette architecture pour des applications complexes où la scalabilité doit être présent. De plus, ce projet m'a permis de définir des objectifs futurs plus concrets comme la mise en place d'un développement « Clean Code » ou encore d'un monitoring stable et robuste permettant d'améliorer la qualité du développement.

Ensuite, la découverte du Framework Angular a été un véritable coup de cœur. Etant un grand utilisateur de React Js, j'ai été réellement surpris de la robustesse d'Angular avec l'ensemble des outils et des technologies mis à disposition.

Pour ce qui est des perspectives d'améliorations, il serait intéressant de séparer davantage les services pour mieux respecter le principe de micro services. Pour mon cas, je n'en ai pas trouvé l'intérêt puisque l'application n'était pas si conséquente pour en séparer.

De plus, suite à cela, ajouter des communications interservices et ajouter une sécurité pour ces communications.

IX- Veilles technologiques

En tant que Concepteur et Développeur d'applications, il est essentiel de faire une veille technologique me permettant d'anticiper les évolutions du secteur mais aussi de maintenir ma présence dans le cadre d'une application sécurisée. Pour ma veille, j'ai décidé de me concentrer sur la sécurité, le design Front End et le devOps.

Pour cela, mes sources de veilles sont diverses :

- Tiktok : Malgré qu'elle soit non conventionnel, celle-ci me permet de me détacher de cette idée de travail, me permet de découvrir des créateurs spécialisés en design UI/UX et donc développer mes inspirations et repérer des astuces.
- LinkedIn : On y trouve des posts qui partagent des expériences, des solutions de Clean Code, des groupes spécialisés dans un langage et partageant les dernières nouveautés avec notamment les dernières failles.
- Youtube : on y trouve des chaînes spécialisées sur des langages et Framework, qui partagent les dernières nouveautés avec des tutos et des idées.

X- Remerciements

Je souhaite tout d'abord remercier l'ensemble des formateurs et intervenants qui nous ont accompagnés, durant cette formation de Concepteur et Développeur d'applications, pour leur partage.

Je tiens aussi à remercier M2I Formation et leur personnel pour l'ensemble des matériaux mise à notre disposition, leur présence et leur soutien lorsqu'on en avait besoin.

Je souhaite particulièrement remercier Christel EHRHART pour son aide et son implication tout au long de cette formation et qui nous a fourni des connaissances solides notamment pour la conception et gestion de projet.

Sans oublier Romain Sessa que je remercie pour m'avoir permis de découvrir le langage Java, ce qui m'a motivé à faire ce projet.

XI- Références

Technologie	Référence	Utilité
Spring boot	Dcomuentation Spring boot	Partie Microservice
Spring Cloud	Documentation Spring Cloud	Configuration de la Gateway
Angular	Documentation Angular 21	Partie Front
Tailwind CSS	Documentation Talwind CSS	Style Front
PrimeNg	Guide PrimeNg v21	Complément Angular

XII- ANNEXES

A. Pipeline CI/CD

1) Pipeline CI

```yaml
name: CI Auth Service

on:
    push:
        paths:
            - 'auth-service/**'
    pull_request:
        paths:
            - 'auth-service/**'

jobs:
    ci:
        uses: ./.github/workflows/reusable-ci.yml
        with:
            service-path: 'auth-service'
        secrets:
            SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

Figure 38 CI Auth-Service

```yaml
name: Reusable CI Workflow

on:
    workflow_call:
        inputs:
            service-path:
                required: true
                type: string
        secrets:
          SONAR_TOKEN:
            required: true
            description: "SonarCloud authentication token"

jobs:
    ci:
        runs-on: ubuntu-latest
        steps:
            - name: Checkout code
              uses: actions/checkout@v4

            - name: Set up JDK 25
              uses: actions/setup-java@v4
              with:
                  distribution: 'temurin'
                  java-version: '25'
                  cache: 'maven'

            - name: Cache SonarQube packages
              uses: actions/cache@v4
              with:
                path: ~/.sonar/cache
                key: ${{ runner.os }}-sonar
                restore-keys: ${{ runner.os }}-sonar

            - name: Execute Test and Code Quality Analysis
              env:
                SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
              run: |
                cd ${{ inputs.service-path }}
                mvn verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar jacoco:check \
                    --file pom.xml \
                    -DskipTests \
                    -Dsonar.projectKey=Mzk-Ali_app_ms_simswap \
                    -Dsonar.host.url=https://sonarcloud.io \
                    -Dsonar.token=$SONAR_TOKEN
            - name: Build and Test Service
              run: |
                cd ${{ inputs.service-path }}
                mvn clean verify -DskipTests
```

Figure 39 CI réutilisable

2) Pipeline CD

```yaml
name: CD Auth Service

on:
    workflow_run:
        workflows: ["CI Auth Service"]
        types:
            - completed

jobs:
    cd:
        if: |
            github.event.workflow_run.conclusion == 'success' &&
            (github.event.workflow_run.head_branch == 'main' ||
            github.event.workflow_run.head_branch == 'dev' ||
            startsWith(github.event.workflow_run.head_branch, 'release/'))
        runs-on: ubuntu-latest

        steps:
            - name: Extract branch name and generate tag
              id: extract
              run: |
                BRANCH_NAME="${{ github.event.workflow_run.head_branch }}"
                IMAGE_TAG=""
                LATEST_TAG=""

                if [[ "$BRANCH_NAME" == "dev" ]]; then
                    IMAGE_TAG="dev-${{ github.run_number }}"
                    LATEST_TAG="dev"

                elif [[ "$BRANCH_NAME" == "main" ]]; then
                    IMAGE_TAG="prod-${{ github.run_number }}"
                    LATEST_TAG="prod"

                elif [[ "$BRANCH_NAME" == release/* ]]; then
                    RELEASE_NAME="${BRANCH_NAME#release/}"
                    IMAGE_TAG="staging-${RELEASE_NAME}-${{ github.run_number }}"
                    LATEST_TAG="staging"

                elif [[ "$BRANCH_NAME" == feature/* ]]; then
                    FEATURE_NAME="${BRANCH_NAME#feature/}"
                    IMAGE_TAG="feat-${FEATURE_NAME}-${{ github.run_number }}"

                elif [[ "$BRANCH_NAME" == fix/* ]]; then
                    FIX_NAME="${BRANCH_NAME#fix/}"
                    IMAGE_TAG="fix-${FIX_NAME}-${{ github.run_number }}"

                else
                    IMAGE_TAG="build-${{ github.run_number }}"
                fi

                echo "Using image tag: $IMAGE_TAG"
                echo "image_tag=$IMAGE_TAG" >> "$GITHUB_OUTPUT"

                if [[ -n "$LATEST_TAG" ]]; then
                echo "latest_tag=$LATEST_TAG" >> "$GITHUB_OUTPUT"
                fi

            - name: Call Reusable CD Workflow
              uses: ./.github/workflows/reusable-cd.yml
              with:
                service: 'auth-service'
                image-tag: ${{ steps.extract.outputs.image_tag }}
                latest-tag: ${{ steps.extract.outputs.latest_tag }}
```

Figure 40 CD Auth-Service

```yaml
name: Reusable CD Workflow

on:
    workflow_call:
        inputs:
            service:
                required: true
                type: string
            image-tag:
                required: true
                type: string
                default: 'latest'
            latest-tag:
                required: false
                type: string

jobs:
    cd:
        runs-on: ubuntu-latest
        steps:
            - name: Checkout code
              uses: actions/checkout@v4

            - name: Set up Docker Buildx
              uses: docker/setup-buildx-action@v4

            - name: Cache Docker layers
              uses: actions/cache@v4
              with:
                path: /tmp/.buildx-cache
                key: ${{ runner.os }}-docker-${{ inputs.service }}-${{ inputs.image-tag }}
                restore-keys: |
                  ${{ runner.os }}-docker-${{ inputs.service }}-

            - name: Build Docker Image with Cache
              run: |
                cd ${{ inputs.service }}
                docker buildx build \
                    --cache-from=type=local,src=/tmp/.buildx-cache \
                    --cache-to=type=local,dest=/tmp/.buildx-cache-new \
                    -tag ghcr.io/${{ github.repository_owner }}/${{ inputs.service }}:${{ inputs.image-tag }} \
                    --load .

            - name: Login to GitHub Container Registry
              uses: docker/login-action@v3
              with:
                  registry: ghcr.io
                  username: ${{ github.repository_owner }}
                  password: ${{ secrets.GITHUB_TOKEN }}

            - name: Scan Image with Trivy
              uses: aquasecurity/trivy-action@master
              with:
                image-ref: ghcr.io/${{ github.repository_owner }}/${{ inputs.service }}:${{ inputs.image-tag }}
                format: 'table'
                ignore-unfixed: true

            - name: Push Docker Image
              run: |
                docker push ghcr.io/${{ github.repository_owner }}/${{ inputs.service }}:${{ inputs.image-tag }}

            - name: Tag and push as latest-tag
              if: ${{ inputs.latest-tag != '' }}
              run: |
                echo "Retagging image as '${{ inputs.latest-tag }}'"
                docker tag ghcr.io/${{ github.repository_owner }}/${{ inputs.service }}:${{ inputs.image-tag }} \
                           ghcr.io/${{ github.repository_owner }}/${{ inputs.service }}:${{ inputs.latest-tag }}

                docker push ghcr.io/${{ github.repository_owner }}/${{ inputs.service }}:${{ inputs.latest-tag }}

            - name: Deploy to development environment
              run: |
                echo "Deploying ${{ inputs.service }} with tag ${{ inputs.image-tag }} to development environment"
```

Figure 41 CD réutilisable

B. Docker Compose

1) Monitoring

```yaml
services:
  prometheus:
    image: prom/prometheus
    container_name: prometheus
    restart: unless-stopped
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
      - ./prometheus/file_sd_configs:/etc/prometheus/file_sd_configs
      - ./prometheus/rules:/etc/prometheus/rules
    networks:
      - observability

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    restart: unless-stopped
    env_file:
      - .env
    volumes:
      - ./grafana/provisioning:/etc/grafana/provisioning
      - ./grafana/dashboards:/var/lib/grafana/dashboards
    ports:
      - "3000:3000"
    networks:
      - observability
    depends_on:
      - prometheus
      - loki
      - tempo

  tempo:
    image: grafana/tempo:latest
    container_name: tempo
    restart: unless-stopped
    command: [ "-config.file=/etc/tempo/tempo.yml" ]
    volumes:
      - ./tempo:/etc/tempo
    ports:
      - "3200:3200"
      - "4317:4317"
      - "4318:4318"
    networks:
      - observability

  loki:
    image: grafana/loki:main
    container_name: loki
    command: [ "-config.file=/etc/loki/local-config.yaml" ]
    ports:
      - "3100:3100"
    networks:
      - observability

  node-exporter:
    image: prom/node-exporter:latest
    container_name: node-exporter
    restart: unless-stopped
    ports:
      - "9100:9100"
    networks:
      - observability

networks:
  observability:
    external: true
```

Figure 42 Docker Compose du Monitoring

2) Application

```yaml
services:
  discovery-service:
    build:
      context: ./discovery-service
      dockerfile: Dockerfile
    container_name: discovery-service
    environment:
      - SPRING_PROFILES_ACTIVE=${SPRING_PROFILES_ACTIVE}
      - EUREKA_CLIENT_REGISTER_WITH_EUREKA=false
      - EUREKA_CLIENT_FETCH_REGISTRY=false
      - OTEL_EXPORTER_OTLP_ENDPOINT=${OTEL_EXPORTER_OTLP_ENDPOINT}
      - OTEL_SERVICE_NAME=discovery-service
      - OTEL_METRICS_EXPORTER=none
      - OTEL_TRACES_EXPORTER=otlp
    ports:
      - "8761:8761"
    networks:
      - default
      - observability
    env_file:
      - .env
    deploy:
      resources:
        limits:
          cpus: '0.5'   # Limite l'usage CPU à 50% d'un CPU
          memory: 512M  # Limite l'usage mémoire à 512MB
        reservations:
          cpus: '0.25'  # Réserve au moins 25% d'un CPU
          memory: 256M  # Réserve au moins 256MB de mémoire
  gateway-service:
    build:
      context: ./gateway-service
      dockerfile: Dockerfile
    container_name: gateway-service
    environment:
      - FRONTEND_URL=${FRONTEND_URL}
      - SPRING_PROFILES_ACTIVE=${SPRING_PROFILES_ACTIVE}
      - SPRING_APPLICATION_NAME=gateway-service
      - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://discovery-service:${DISCOVERY_PORT}/eureka/
      - EUREKA_INSTANCE_PREFER_IP_ADDRESS=true
      - JWT_SECRET=${JWT_SECRET}
      - SPRING_DATA_REDIS_HOST=${REDIS_HOST}
      - SPRING_DATA_REDIS_PORT=${REDIS_PORT}
      - SPRING_DATA_REDIS_TIMEOUT=${REDIS_TIMEOUT}
      - PROMETHEUS_PORT=8080
      - LOKI_URL=${LOKI_URL}
      - OTEL_EXPORTER_OTLP_ENDPOINT=${OTEL_EXPORTER_OTLP_ENDPOINT}
      - OTEL_SERVICE_NAME=gateway-service
      - OTEL_METRICS_EXPORTER=none
      - OTEL_TRACES_EXPORTER=otlp
    ports:
      - "${GATEWAY_PORT}:8888"
    networks:
      - default
      - observability
    depends_on:
      discovery-service:
        condition: service_started
      redis-gateway:
        condition: service_healthy
    env_file:
      - .env
    deploy:
      resources:
        limits:
          cpus: '0.5'   # Limite l'usage CPU à 50% d'un CPU
          memory: 512M  # Limite l'usage mémoire à 512MB
        reservations:
          cpus: '0.25'  # Réserve au moins 25% d'un CPU
          memory: 256M  # Réserve au moins 256MB de mémoire
```

Figure 43 Partie du Docker Compose Application