**Projet de Fin d’Etudes**

Master Sciences et Techniques Sciences des données et systèmes intelligents

Etude et développement d’un outil de gestion des affectations et du suivi des projets



**Lieu de stage: Hightech Payment Systeme Worldwide**

|  |  |
| --- | --- |
| **Réalisé par :** | **Encadré par :** |
| * ECHEOUATI El Mehdi | * Pr. DAHIR Sarah * M. HASSANE Khalid |

**Soutenu le 00/07/2025 devant le jury composé de :**

Pr.

Pr.

Pr. DAHIR Sarah

# REMERCIEMENT

J'exprime ma profonde gratitude et ma reconnaissance respectueuse envers mon encadrante,

Pr. Sarah dahir,

Pour sa volonté de m'encadrer, le temps précieux qu'il m'a accordé et les précieux conseils qu'il m'a prodigués.

Je tiens également à remercier chaleureusement

Monsieur Khalid hassane,

Chef de projet et encadrant au sein de l’entreprise, pour son soutien et son encouragement tout au long de mon projet. Sa contribution et son appui ont été essentiels à la réussite de mon travail.

Je remercie également,

M. asmaa jaloully,

Mon encadrante en entreprise, pour la qualité de son suivi durant l’ensemble du projet.

Enfin, je tiens à exprimer ma reconnaissance envers tous les membres du jury pour avoir accepté d'évaluer mon travail.

# RESUME

# ABSTRACT

# DEDICACE

# TABLE DE MATIERES

[REMERCIEMENT 2](#_Toc200999482)

[RESUME 3](#_Toc200999483)

[ABSTRACT 4](#_Toc200999484)

[DEDICACE 5](#_Toc200999485)

[TABLE DE MATIERES 6](#_Toc200999486)

[LISTE DES FIGURES 9](#_Toc200999487)

[LISTE DES TABLEAUX 10](#_Toc200999488)

[LISTE DES ABREVIATIONS 11](#_Toc200999489)

[INTRODUCTION GENERALE 12](#_Toc200999490)

[Contexte et problématique 12](#_Toc200999491)

[Contributions 12](#_Toc200999492)

[Organisation 12](#_Toc200999493)

[CHAPITRE I : CONTEXTE GENERAL DU PROJET 13](#_Toc200999494)

[Introduction 13](#_Toc200999495)

[Présentation générale de HPS 13](#_Toc200999496)

[L’industrie de la monétique 13](#_Toc200999497)

[La solution monétique PowerCARD 15](#_Toc200999498)

[Services Complémentaires offerts par HPS 19](#_Toc200999499)

[Organigramme de HPS 20](#_Toc200999500)

[Présentation du département d’accueil 21](#_Toc200999501)

[Présentation du projet 22](#_Toc200999502)

[Eude de l’existant 22](#_Toc200999503)

[Problématique 22](#_Toc200999504)

[Solution proposée 22](#_Toc200999505)

[Conduite du projet 23](#_Toc200999506)

[Méthodologie Agile/SCRUM 23](#_Toc200999507)

[Sprints exécutés 24](#_Toc200999508)

[Planification (diagramme de Gant) 24](#_Toc200999509)

[Conclusion 25](#_Toc200999510)

[CHAPITRE II : Etat de l’art 26](#_Toc200999511)

[Introduction 26](#_Toc200999512)

[Architectures logicielles : Monolithique vs Microservices 26](#_Toc200999513)

[Justification du choix de l’architecture microservices 28](#_Toc200999514)

[Architecture micro services 28](#_Toc200999515)

[Principes de l’architecture microservices 28](#_Toc200999516)

[Schéma de l’architecture microservices 29](#_Toc200999517)

[Spring boot dans le contexte micro services 31](#_Toc200999518)

[Avantage de Spring Boot pour les microservices 31](#_Toc200999519)

[Structure typique d’un microservice spring boot 32](#_Toc200999520)

[Communication entre microservices 33](#_Toc200999521)

[Communication synchrone 33](#_Toc200999522)

[Communication asynchrone 34](#_Toc200999523)

[Choix du mode de communication dans notre projet 35](#_Toc200999524)

[Sécurité des Systèmes Distribués 36](#_Toc200999525)

[Système d’authentification : Stateful vs Stateless 36](#_Toc200999526)

[JSON Web Token 37](#_Toc200999527)

[a) Structure d’un JWT 38](#_Toc200999528)

[b) Encodage, hachage et signature 39](#_Toc200999529)

[c) Vérification du JWT et confiance numérique 39](#_Toc200999530)

[d) Limites du modèle HMAC et avantages du modèle RSA 39](#_Toc200999531)

[e) Gestion des secrets dans une architecture distribuée 39](#_Toc200999532)

[Le protocole OAuth2 40](#_Toc200999533)

[Le protocole OpenID Connect 40](#_Toc200999534)

[Sécurité avec spring Security 40](#_Toc200999535)

[Keycloak 41](#_Toc200999536)

[Gestion des données (spring data JPA) 42](#_Toc200999537)

[Infrastructure et déploiement 42](#_Toc200999538)

[Containerisation 42](#_Toc200999539)

[Orchestration 42](#_Toc200999540)

[Conclusion 42](#_Toc200999541)

[CHAPITRE III : ANALYSE ET CONCEPTION 43](#_Toc200999542)

[Introduction 43](#_Toc200999543)

[Analyse des besoins 43](#_Toc200999544)

[Besoins fonctionnels 43](#_Toc200999545)

[Besoins non fonctionnels 43](#_Toc200999546)

[Modélisation des besoins fonctionnels 43](#_Toc200999547)

[Diagramme de cas d’utilisation 43](#_Toc200999548)

[Diagramme de séquences 43](#_Toc200999549)

[Diagramme de classes 43](#_Toc200999550)

[Conclusion 43](#_Toc200999551)

[CHAPITRE IV : MISE EN ŒUVRE 44](#_Toc200999552)

[Introduction 44](#_Toc200999553)

[Outils de développement 44](#_Toc200999554)

[Front end 44](#_Toc200999555)

[Back end 44](#_Toc200999556)

[Réalisation 44](#_Toc200999557)

[Conclusion 44](#_Toc200999558)

[CONCLUSION GENERALE 44](#_Toc200999559)

[Synthèse 44](#_Toc200999560)

[Perspective 44](#_Toc200999561)

[Références 44](#_Toc200999562)

# LISTE DES FIGURES

# LISTE DES TABLEAUX

# LISTE DES ABREVIATIONS

# INTRODUCTION GENERALE

## Contexte et problématique

## Contributions

## Organisation

# CHAPITRE I : CONTEXTE GENERAL DU PROJET

## Introduction

## Présentation générale de HPS

Créée en 1995 et basée à Casablanca, **HPS (Hightech Payment Systems)** est une entreprise marocaine de renommée internationale, spécialisée dans la fourniture de solutions de paiement électronique sécurisées. Elle conçoit, développe et commercialise des logiciels destinés aux institutions financières, aux processeurs de paiement, aux opérateurs de cartes, aux gouvernements et aux commerçants.

La société est surtout connue pour sa plateforme **PowerCARD**, une solution modulaire et intégrée qui couvre l'ensemble de la chaîne de valeur des paiements électroniques : émission, acquisition, switching, gestion des fraudes, etc. Cette solution est utilisée dans plus de 90 pays à travers le monde.

HPS s’impose aujourd’hui comme un acteur majeur de la **fintech** au niveau international. Cotée à la Bourse de Casablanca, l’entreprise connaît une croissance continue et investit fortement dans l’innovation et la transformation digitale.

|  |  |
| --- | --- |
| Raison sociale | Hightech Payment Systems |
| Forme juridique | Société Anonyme |
| Directeur General | M. Mohamed Horani |
| Date de création | 1995 |
| Introduction à la bourse | 2006 |
| Domaine d’activités | Monétique : Fournisseur de solutions de  Paiement électronique multi-canal |
| Siège social | Casablanca, Maroc |
| Filiales | HPS Europe HPS Dubai HPS Singapore |
| Site web | www.hps-worldwide.com/ |
| Logo |  |
| Chiffre d’affaires | 1007,1 millions de MAD |

Fiche Signalétique

### L’industrie de la monétique

**Introduction à la monétique**

La monétique est définie comme l’ensemble des traitements électroniques, informatiques et télématiques nécessaires à la gestion de cartes de paiement ainsi que des transactions associées et des transferts de fonds.

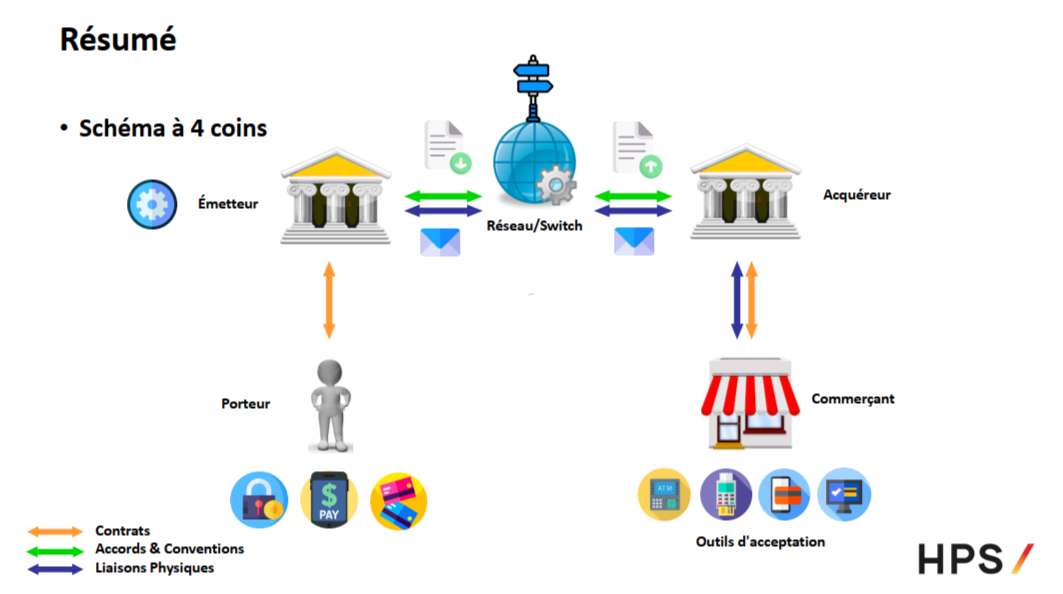
En d’autres termes, la monétique regroupe les processus nécessaires à la création d’une carte, la lecture des informations associées et la gestion des transactions monétaire. Concrètement, la monétique est un processus qui relie un utilisateur final (porteur) doté d'un terminal mobile (carte) et un centre d'information, où sont stockées les données du porteur.

La monétique est un synonyme de paiement électronique, c’est un domaine qui regroupe toute forme de transaction utilisant soit une carte magnétique, soit une carte à puce, soit le paiement par Internet et parfois même la combinaison de deux formes de paiement. Les acteurs de la monétique sont principalement les banques, les organismes émetteurs de carte (Visa, MasterCard, etc.), les sociétés de services et d’implantation des architectures technologiques de support au paiement électronique.

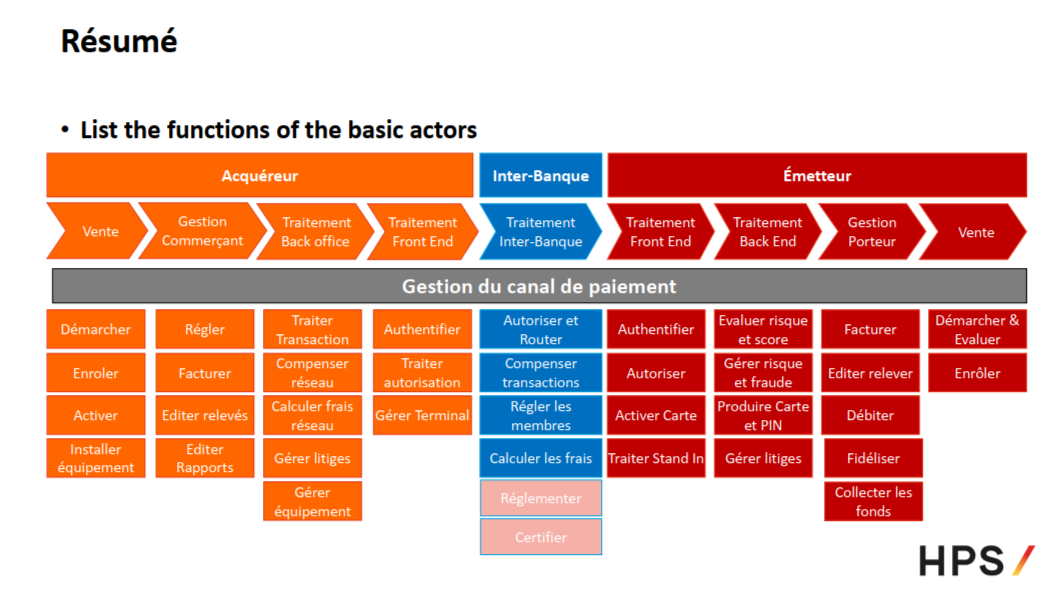
**Le déroulement d’une transaction électronique**

Les principaux acteurs intervenants lors d'une transaction de paiement électronique sont les suivants :

* Le porteur : est une personne physique qui porte la carte de paiement. Dans une transaction bancaire, il s’agit du client
* L’émetteur : est un organisme financier ou assimilé qui émet la carte de paiement. Dans une transaction bancaire, l’émetteur est la banque du client.
* L’accepteur : est une personne physique ou morale qui accepte le moyen de paiement grâce à un système accepteur. Dans une transaction bancaire, l’accepteur est assimilé au commerçant équipé d’un terminal de paiement électronique (TPE).
* L’acquéreur : est l’organisme financier ou assimilé qui va acquérir les données de la transaction. Dans une transaction bancaire, il s’agit de la banque du commerçant. Les DAB ou GAB sont aussi assimilés à des commerçants.
* L’organisme d’interchange : Il garantit l’interopérabilité entre les différents acquéreurs et émetteurs qui y adhèrent.



Les acteurs de la chaine de valeur de paiement



Les acteurs de base et leurs fonctions

### La solution monétique PowerCARD

Au cœur de l’offre technologique de HPS se trouve **PowerCARD**, une plateforme complète et intégrée dédiée à la gestion des paiements électroniques. Cette solution phare se distingue par sa capacité à couvrir l’ensemble de la chaîne de valeur du paiement, depuis l’émission des cartes jusqu’à la gestion des transactions, en intégrant des fonctionnalités avancées de sécurité et de lutte contre la fraude.

La **fraude dans les paiements électroniques** désigne toute activité illégitime visant à détourner des fonds ou exploiter les systèmes de paiement à des fins malveillantes, tels que le vol de données de carte, l’usurpation d’identité ou la manipulation de transactions. Dans ce contexte, PowerCARD intègre des mécanismes sophistiqués de **détection, prévention et gestion des fraudes**, permettant aux institutions financières de protéger les données sensibles et de renforcer la confiance des utilisateurs.

✅ **Une couverture complète du cycle de paiement**

PowerCARD prend en charge toutes les étapes du processus de paiement :

* L’émission et la personnalisation des cartes bancaires,
* L’autorisation et le traitement des transactions,
* Le contrôle en temps réel des risques,
* La détection et la prévention des fraudes.

Cette approche globale permet aux institutions financières de centraliser et de sécuriser leurs opérations de paiement via une plateforme unique.

🔄 **Une plateforme ouverte, modulaire et omnicanal**

PowerCARD a été conçue pour répondre à la diversité des besoins du marché :

* Elle prend en charge **tous les canaux de paiement** : guichets automatiques (ATM), terminaux de paiement (TPE), paiements en ligne, paiements mobiles, etc.
* Elle gère **tous types de cartes** : cartes de crédit, de débit, cartes prépayées, cartes de fidélité, cartes virtuelles, etc.
* Elle repose sur une **architecture modulaire**, offrant aux clients la possibilité d’intégrer uniquement les fonctionnalités nécessaires à leurs activités.

🚀 **Une solution innovante et évolutive**

Grâce à son ouverture et à sa capacité d’adaptation, PowerCARD permet aux banques, fintechs, et autres opérateurs de paiement de proposer à leurs clients des services modernes tels que :

* Les paiements mobiles via NFC ou QR code,
* Les portefeuilles électroniques (e-wallets),
* L’authentification forte des paiements en ligne,
* La gestion multi-devises et multi-institutions.

🔸 **Les différentes solutions PowerCARD**

PowerCARD est constituée de plusieurs **modules spécialisés**, chacun conçu pour couvrir un aspect précis du processus de paiement. Ces modules fonctionnent de manière indépendante ou complémentaire, selon les besoins de l’organisation. Voici quelques exemples de modules clés :

|  |  |  |
| --- | --- | --- |
| PowerCARD-Issuer | Services complets d'émission et gestion de tout type de carte Création Client, Souscription, Activation & Rétention Emission total d'un Produit Outil puissant de gestion des Frais Gestion de Fraude & Risque, Gestion des litiges, Réclamations Capacité de gestion efficace et puissante du service Client | Card Issuing | HPS Worldwide |
| PowerCARD-Acquirer | Service pour gérer des commerçants tenant compte de leurs modèles de fonctionnement Démarchage commerçant, Enrôlement, Activation & Rétention Modèles de facturation très riches, DCC & Gestion des frais réseaux Facturation avancée, Compensation et Règlement commerçant Gestion de Fraude & Risque, Gestion des litiges, Réclamations | Merchant Acquiring | HPS Worldwide |
| **PowerCARD-ACH** | Configuration et gestion de : l'autorisation, la compensation & le règlement entre le switch et ses membres Gestion des membres, de leurs comptes et des règlements Frais et Commissions Standard Local Interchange System (LIS) Standard Switch Interface Description (SID) | PowerCARD-ACH | HPS Worldwide |
| **PowerCARD-Switch** | Routage, stand-in, processus d'autorisation Gestion de l'autorisation Routage des messages Store And Forward (SAF) Intégration et gestion du multicanal |  |
| **PowerCARD-xPOS** | Gestion de tous types de terminal de paiement électroniques et des autorisations Gestion du Parc de Terminaux Acquisition des données Services à valeur ajoutée (DCC, Paiement de Factures, Cash-Back, Détaxe ) | POS Management | HPS Worldwide |
| **PowerCARD-ATM** | Solution qui permet de gérer efficacement le parc de GAB de tout type Déclaration et supervision des GAB Gestion du Routage et des services GAB Services à valeur ajoutée sur GAB (Paiement de Facture, dépôt Cash, Retrait sans Carte…) | ATM Management | HPS Worldwide |
| **PowerCARD-Wallet** | Solution de paiement mobile entre différentes entités P2P, P2M, G2P, P2B Gestion de Wallet (Portefeuille) Gestion de compte prépayé Comptabilité | Mobile Payment | HPS Worldwide |
| **PowerCARD-eSecure** | Sécuriser le paiement d'un porteur « en ligne » en assurant une couche d'authentification supplémentaire au moment de l'autorisation. Procédure d'enrôlement & gestion des produits Méthodes d'authentification multiple Visa & MasterCard Compliance |  |
| **PowerCARD-Tokenization** | Solution complète pour émettre, approvisionner, stocker et gérer les tokens. Tokenisation / De-tokenisation inclus gestion de risque et règles de routage 22 Gestion du stock des Tokens Gestion du cycle de vie des Tokens et de la relation client Plusieurs Scenarios de déploiement Interfaces avec CRM, Provisionnement Mobile, Système d'autorisation et Réseau de paiement | Mobile Payment | HPS Worldwide |
| **PowerCARD-Fraud** | Une solution Stand-alone pour la surveillance en temps rée l des autorisations provenant de n'importe quels systèmes Détection de la fraude en Temps reel ou quasi réel Système à base de règles Gestions des alertes via SMS et Email Gestion des dossiers | Fraud Monitoring | HPS Worldwide |
| **PowerCARD-WebPublisher** | Portails qui offrent des fonctions self-services pour commerçant, porteur, entreprise, agence... Website Content Configuration, Personalization & Segmentation Website Publishing Multiple 3rd Party Systems Real-time Data Aggregation Native Interfaces With Other PowerCARD Solutionse, Système d'autorisation et Réseau de paiement |  |
| **PowerCARD-BI** | Ensemble de graphes, de données présentées en tableau de bord permettant d'avoir une idée claire de la santé du métier et KPIs. Tableaux de bord prédéfinis et personnalisables Analyse et forage de données Intégration native avec les solutions PowerCARD-Issuer et PowerCARD-Acquirer Supports d'enrichissement de données par des systèmes tiers | Payment Analytics | HPS Worldwide |

### Services Complémentaires offerts par HPS

En plus de sa plateforme PowerCARD, HPS propose une gamme de services complémentaires à forte valeur ajoutée, conçus pour accompagner ses clients tout au long de leurs projets monétiques. Ces services renforcent l’expertise de HPS dans le secteur des paiements électroniques et témoignent de sa capacité à répondre aux besoins spécifiques de chaque organisation.

🔹 Conseil en monétique

Créée par des spécialistes du domaine, HPS a su capitaliser sur son expertise historique pour fournir des services de conseil stratégique et opérationnel. Forte de nombreuses années d’expérience, l’entreprise intervient notamment dans :

* La mise en place de centres monétiques privés ou interbancaires,
* La participation à des comités consultatifs internationaux,
* Le développement de solutions logicielles innovantes dans le domaine des paiements.

🔹 Audit des systèmes

Grâce à sa maîtrise des évolutions technologiques, HPS propose également des prestations d’audit pour évaluer l’efficacité et la sécurité des systèmes monétiques. Ces audits sont réalisés conformément aux normes internationalement reconnues et visent à :

* Vérifier la robustesse de l’architecture technique,
* Identifier les vulnérabilités potentielles,
* Garantir l’alignement des offres avec les exigences du marché.

🔹 Mode ASP (Application Service Provider)

Le modèle ASP permet aux clients d’externaliser l’hébergement des serveurs d’application et des bases de données chez HPS. L’accès à la plateforme PowerCARD se fait alors à travers une connexion sécurisée à distance. Ce mode de déploiement offre plusieurs avantages :

* Réduction des coûts liés aux investissements en infrastructure,
* Facturation basée sur l’usage (transactions, nombre de cartes ou de commerçants),
* Supervision, administration et maintenance assurées par les équipes HPS ou leurs partenaires.

🔹 Adaptation aux besoins spécifiques

Bien que HPS propose des produits standards, l’entreprise met un point d’honneur à les adapter aux exigences spécifiques de chaque client. Cette personnalisation peut porter sur :

* L’intégration aux systèmes d’information existants,
* L’adaptation aux réglementations locales,
* Le choix des langues et devises,
* Les particularités des calculs d’intérêts, etc.

🔹 HPS Academy

HPS Academy est une initiative mise en place par HPS pour accompagner la montée en compétences de ses collaborateurs, partenaires et clients. Ce service propose des programmes de formation spécialisés dans les domaines de la monétique, des systèmes de paiement et des technologies PowerCARD. L’objectif est de favoriser la transmission du savoir-faire HPS, de garantir une bonne maîtrise des solutions proposées, et de soutenir les projets de transformation digitale des institutions financières.

Cette flexibilité repose sur une architecture produit cohérente et sur un accompagnement sur mesure, permettant aux clients de bénéficier de solutions à la fois robustes, évolutives et parfaitement alignées avec leurs attentes.

### Organigramme de HPS

Le groupe HPS est dirigé par **Monsieur Mohamed HORANI**, et son organisation repose sur une structuration claire autour de quatre principales entités. Chacune de ces unités joue un rôle spécifique et complémentaire, permettant à HPS de répondre efficacement aux besoins de ses clients et de renforcer son positionnement sur le marché des paiements électroniques à l’échelle internationale.

🔹 HPS Group

Il s’agit de l’unité centrale qui assure la gouvernance stratégique et financière du groupe. Elle définit les orientations globales, pilote les investissements, et veille à la cohérence des actions menées par les différentes entités opérationnelles.

🔹 HPS Solutions

Cette entité est dédiée au **développement, à l'évolution et à la maintenance des solutions logicielles** proposées par HPS, notamment la plateforme PowerCARD. Elle se concentre également sur l’innovation technologique et l’adaptation des produits aux besoins du marché.

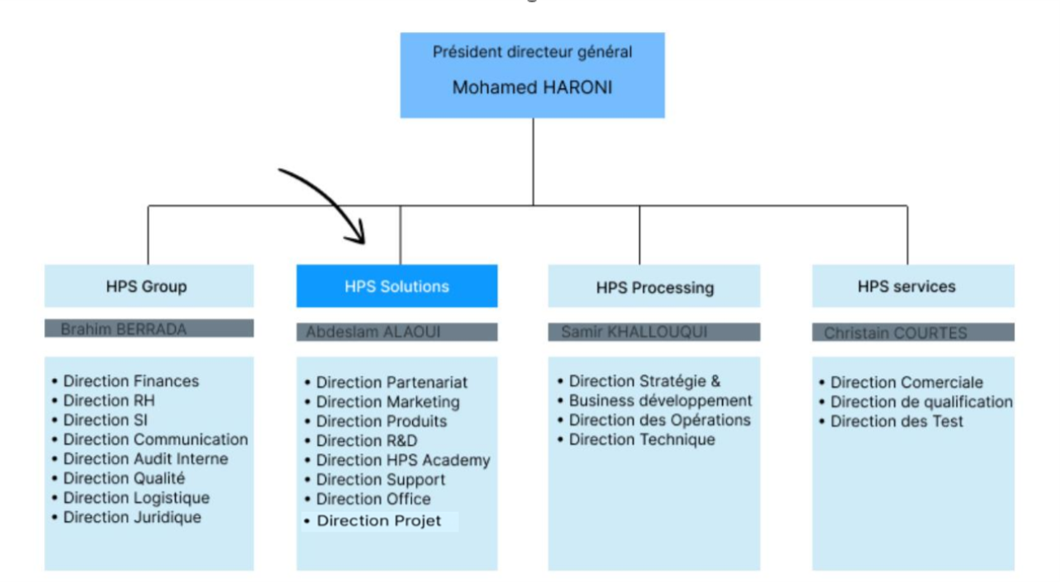
🔹 HPS Processing

HPS Processing fournit des services de traitement des transactions de paiement, en mode externalisé. Elle gère des infrastructures sécurisées pour le compte de banques et d’autres institutions financières, dans le respect des normes les plus strictes en matière de performance et de sécurité.

🔹 HPS Services

Cette unité regroupe les activités de **conseil, de support technique, d’intégration de solutions et de formation**. Elle accompagne les clients dans la mise en œuvre, la personnalisation et l’optimisation des solutions HPS à chaque étape de leur projet.

Une représentation schématique de cette organisation est présentée dans la figure suivante (Figure …).



Organigramme de HPS

### Présentation du département d’accueil

Mon stage de fin d’études a été réalisé au sein de l’unité **HPS Solutions**, rattachée à la **Direction Projet**. Plus précisément, j’ai intégré l’équipe en charge de l’équipe **Wallets**, spécialisée dans le développement et l’évolution des solutions de portefeuilles électroniques proposées par la plateforme PowerCARD

L’équipe **Wallets** au sein de HPS Solutions est responsable du développement, de l’évolution et du déploiement de la solution **PowerCARD-Wallet**, un portefeuille électronique en marque blanche. Cette solution mobile innovante permet la réalisation de paiements instantanés, transfert et réception d’argents, et la réception de notifications en temps réel. Elle repose sur un compte prépayé intégré.

PowerCARD-Wallet constitue une base technologique robuste pour la mise en œuvre de différents cas d’usage liés aux paiements mobiles tels que le **P2P (de personne à personne)**, **P2M (de personne à commerçant)**, **G2P (du gouvernement à une personne)** ou encore **P2B (de personne à entreprise)**.

La solution assure la gestion complète de l’écosystème d’acteurs : clients, commerçants, agents, distributeurs, etc. Une fois enrôlé sur la plateforme, l’utilisateur peut effectuer des transferts d’argent ou régler des achats en utilisant soit son compte prépayé intégré, soit ses cartes ou comptes bancaires existants comme sources de fonds.

Les opérations de **cash-in** et **cash-out** peuvent s’effectuer à différents points du réseau : guichets automatiques (ATM), agences, agents ou distributeurs agréés.

En outre, PowerCARD-Wallet propose une gestion hiérarchique avancée des clients, prenant en charge toute combinaison entre le client, son téléphone mobile, son portefeuille et les moyens de paiement associés. Grâce aux **API PowerCARD Connect’**, la solution facilite l’intégration avec des systèmes externes, offrant ainsi une flexibilité maximale pour les projets de digitalisation financière.

## Présentation du projet

Dans l’optique de poursuivre mes études dans le domaine des systèmes informatiques et logiciels à la **Faculté des Sciences et Techniques de Fès (FST de Fès)**, et en préparation à mon insertion dans la vie professionnelle, j’ai eu l’opportunité d’intégrer l’entreprise **HPS**.  
Le sujet qui m’a été confié dans le cadre de ce **projet de fin d’études** représente une opportunité passionnante de mettre en œuvre mes compétences techniques, tout en contribuant à la création d’une **solution innovante**.  
Plus précisément, il s’agit de **concevoir et développer une plateforme de gestion des affectations et du suivi des projets**, en adoptant une **architecture basée sur des microservices** pour plus de flexibilité, de scalabilité et de maintenabilité.

### Eude de l’existant

L’analyse de l’existant révèle que la gestion des affectations est actuellement réalisée de manière artisanale, avec des outils comme Excel ou des échanges par e-mail. Ce mode de fonctionnement présente plusieurs limites :

* Difficulté à avoir une vision globale et actualisée des projets en cours.
* Absence d’un outil centralisé pour croiser les données de disponibilité, de compétence et de charge de travail.
* Suivi manuel des tâches, entraînant un risque élevé de retards ou de surcharge.
* Collaboration inefficace entre les chefs de projets et les membres des équipes.

En résumé, le système actuel manque de **traçabilité, de réactivité et de transparence**, ce qui nuit à la performance globale de la gestion de projet.

### Problématique

**Comment centraliser et automatiser la gestion des affectations et le suivi des projets pour améliorer la répartition des ressources et le pilotage global ?**  
Le manque d’outils spécialisés engendre une perte de temps, un manque de visibilité pour les chefs de projet, et une difficulté à prendre des décisions fondées sur des données fiables.  
L’enjeu est donc de **concevoir une solution web centralisée**, intégrant des algorithmes intelligents et une interface ergonomique pour soutenir la coordination, la planification, et l’analyse.

### Solution proposée

La solution proposée est le développement d’une **plateforme web interne** dotée des fonctionnalités suivantes :

a) Gestion des affectations

* Affichage dynamique de tous les projets en cours et prévus.
* Algorithme intelligent d’affectation des collaborateurs selon les **compétences**, **disponibilités** et **charges**.
* Possibilité d’ajustement manuel par les chefs de projet.

b) Suivi des projets et des tâches

* Vue d’ensemble à travers des **tableaux de bord** interactifs.
* Gestion des tâches via un système **Kanban** ou **diagramme de Gantt**.
* Suivi des temps de travail par tâche pour une meilleure estimation.

c) Gestion des collaborateurs

* Fiches détaillées avec compétences, disponibilités, historique des projets.
* Alertes en cas de surcharge ou de deadline critique.

d) Rapports et analyses

* Génération de **rapports PDF/Excel** à destination du management.
* Tableaux de bord pour visualiser les **indicateurs clés de performance (KPI)**.
* Export de données pour analyses approfondies.

Cette plateforme permettra non seulement de **fiabiliser la planification**, mais également de **renforcer la collaboration inter-équipes** et de **faciliter la prise de décision** à tous les niveaux.

## Conduite du projet

### Méthodologie Agile/SCRUM

Dans le cadre de ce projet, nous avons adopté la méthodologie **Agile**, et plus spécifiquement le cadre de travail **SCRUM**, très utilisé dans les projets de développement logiciel. Cette approche se distingue par son caractère itératif, collaboratif et adaptatif. Elle permet une meilleure réactivité face aux changements et garantit une livraison progressive des fonctionnalités.

**Objectifs de SCRUM :**

* Fournir des incréments fonctionnels du produit à intervalles réguliers.
* Améliorer la communication entre les parties prenantes.
* Favoriser l’implication active de l’équipe tout au long du projet.

**Rôles SCRUM adaptés au projet :**

* **Product Owner** : représentant des besoins métiers et fonctionnels. Dans notre cas, il s’agit de l’encadrant chez HPS, responsable de la validation des priorités.
* **SCRUM Master** : facilitateur du processus, veille à l’application de la méthode et supprime les obstacles. Ce rôle a été assuré ponctuellement par un tuteur technique.
* **Développeur(s)** : exécution des tâches techniques, conception, développement, tests. Ce rôle a été assuré par moi-même.

**Cérémonies SCRUM pratiquées :**

* **Sprint Planning** : définition des objectifs à atteindre pour chaque sprint.
* **Daily Scrum** (adapté à une fréquence hebdomadaire) : point rapide sur l'avancement.
* **Sprint Review** : démonstration des fonctionnalités terminées.
* **Sprint Retrospective** : analyse des difficultés rencontrées et pistes d'amélioration.

**Durée des sprints :**

Chaque sprint a duré **2 semaines**, ce qui a permis un bon rythme d'évolution tout en gardant une certaine souplesse.

### Sprints exécutés

Le projet a été divisé en plusieurs sprints, chacun correspondant à une étape clé de la conception et du développement de la plateforme :

|  |  |  |
| --- | --- | --- |
| Sprint | Durée | Objectifs / Tâches principales réalisées |
| Sprint 1 | Semaine 1 à 2 | Analyse des besoins, étude de l’existant, rédaction du cahier des charges |
| Sprint 2 | Semaine 3 à 4 | Conception de l’architecture microservices, modélisation de la base de données |
| Sprint 3 | Semaine 5 à 6 | Développement du module de gestion des projets |
| Sprint 4 | Semaine 7 à 8 | Développement du module d'affectation + interface utilisateur |
| Sprint 5 | Semaine 9 à 10 | Intégration des tableaux de bord et visualisation des tâches |
| Sprint 6 | Semaine 11 à 12 | Tests finaux, débogage, documentation technique |

Sprints exécutés

### Planification (diagramme de Gant)

Un **diagramme de Gantt** a été établi afin de visualiser la planification du projet. Il présente l’enchaînement des tâches, leur durée estimée, ainsi que les chevauchements éventuels entre les différents modules. Ce diagramme a servi de référence tout au long du projet pour assurer un suivi précis de l’avancement.

Ici le diiagramme de Gant

Planification du projet via diagramme de Gant

Il a permis de garantir le respect des délais et d’anticiper les points critiques du projet, en facilitant la communication avec les encadrants et les parties prenantes

# Conclusion

# CHAPITRE II : Etat de l’art

## Introduction

## Architectures logicielles : Monolithique vs Microservices

Définir l’architecture d’un logiciel est une étape essentielle dans le cycle de développement. Elle intervient généralement lors de la phase de conception, connue sous le nom de **conception architecturale**. Si elle est bien pensée et bien adaptée, elle contribue fortement à la stabilité, à l'efficacité, à la maintenabilité et à la pérennité de l'application.

L’**architecture logicielle** décrit la manière dont les différents composants d’une application sont organisés ainsi que leurs interactions. Elle constitue une abstraction structurelle du logiciel et repose sur trois éléments clés :

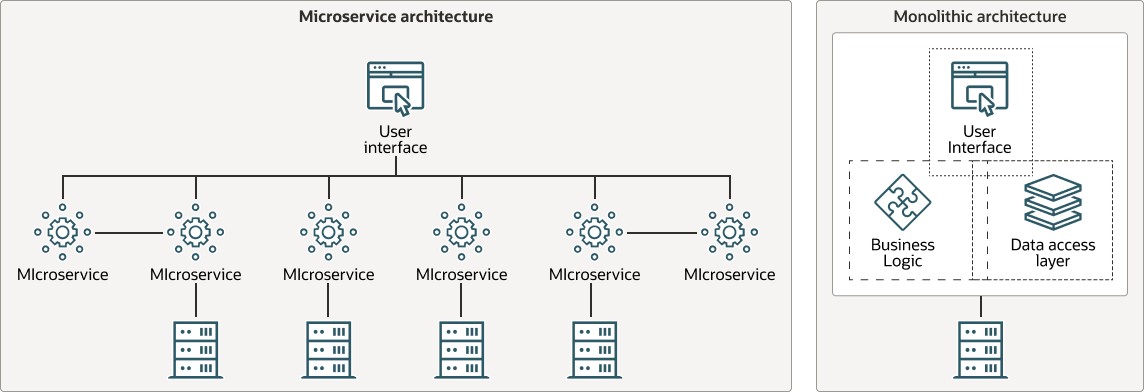
* Des **collections de composants** : clients, serveurs, middlewares, couches logicielles, vues, contrôleurs, classes, interfaces, etc.
* Des **topologies de composants** : relations hiérarchiques, associations, généralisations, agrégations, etc.
* Des **mécanismes d’interaction** : appels de services, envois de messages, partage de mémoire, synchronisation, etc.

Pour être efficace, une architecture doit satisfaire plusieurs **critères de qualité** : évolutivité, simplicité, compatibilité, maintenabilité et connectivité. Le **choix du type d’architecture** dépend donc des objectifs stratégiques de l'application et de ses exigences de déploiement.

Traditionnellement, de nombreuses applications sont conçues selon une **architecture monolithique**, où tous les modules (interface utilisateur, logique métier, accès aux données, etc.) sont regroupés dans une seule base de code, déployée et maintenue comme une seule unité.

En revanche, une **architecture microservices** divise l'application en plusieurs services indépendants, chacun correspondant à une fonctionnalité métier spécifique. Chaque service est autonome, peut être développé, déployé, mis à l’échelle et maintenu indépendamment des autres, ce qui offre une grande flexibilité en termes d’évolution et d’organisation du développement.

L'illustration suivante présente les architectures monolithiques et de microservices.



 Les architectures monolithiques et de microservices.

Le tableau suivant récapitule les différences entre les microservices et les architectures monolithiques.

|  |  |  |
| --- | --- | --- |
| **Caractéristique** | Architecture des microservices | Architecture monolithique |
| Conception d'unités | L'application se compose de services faiblement couplés. Chaque service prend en charge une seule tâche métier. | L'ensemble de l'application est conçu, développé et déployé en une seule unité. |
| Réutilisation des fonctionnalités | Les microservices définissent des API qui exposent leurs fonctionnalités à n'importe quel client. Les clients peuvent même être d'autres applications. | Les possibilités de réutilisation des fonctionnalités entre les applications sont limitées. |
| Communication dans l'application | Pour communiquer entre eux, les microservices d'une application utilisent le modèle de communication demande-réponse. L'implémentation standard utilise des appels d'API REST basés sur le protocole HTTP. | Les procédures internes (appels de fonction) facilitent la communication entre les composants de l'application. Il n'est pas nécessaire de limiter le nombre d'appels de procédure interne. |
| Flexibilité technologique | Chaque microservice peut être développé à l'aide d'un langage de programmation et d'un framework qui conviennent le mieux au problème que le microservice est conçu pour résoudre. | En général, l'application entière est écrite dans un seul langage de programmation. |
| Gestion des données | Décentralisé : chaque microservice peut utiliser sa propre base de données. | Centralisée : l'ensemble de l'application utilise une ou plusieurs bases de données. |
| Déploiement | Chaque microservice est déployé indépendamment, sans affecter les autres microservices de l'application. | Toute modification, aussi petite soit-elle, nécessite un redéploiement et un redémarrage de l'ensemble de l'application. |
| Facilité de maintenance | Les microservices sont simples, ciblés et indépendants. L'application est donc plus facile à maintenir. | Avec l'augmentation de la portée de l'application, la maintenance du code devient plus complexe. |
| Résilience | Les fonctionnalités de l'application sont réparties entre plusieurs services. En cas de défaillance d'un microservice, les fonctionnalités offertes par les autres microservices restent disponibles. | Une défaillance d'un composant peut affecter la disponibilité de l'ensemble de l'application. |
| Evolutivité | Chaque microservice peut être mis à l'échelle indépendamment des autres services. | L'ensemble de l'application doit être mis à l'échelle, même lorsque les besoins de l'entreprise ne concernent que certaines parties de l'application. |

Les différences entre les microservices et les architectures monolithiques.

## Justification du choix de l’architecture microservices

À la suite de l’étude comparative entre l’architecture monolithique et l’architecture microservices, nous avons opté pour une architecture **microservices** pour le développement de notre plateforme de gestion des affectations et du suivi des projets.

Ce choix est motivé par plusieurs facteurs techniques et organisationnels :

* 🔹 **Modularité et séparation des responsabilités** : chaque service représente une fonctionnalité métier bien définie (ex. : gestion des projets, gestion des affectations, suivi des tâches…). Cela facilite la compréhension, le développement et les tests de chaque module indépendamment.
* 🔹 **Facilité de maintenance et d’évolution** : en cas de mise à jour ou de correction, seul le microservice concerné est modifié et redéployé, sans impacter le reste du système.
* 🔹 **Scalabilité ciblée** : certains services plus sollicités (ex. : tableau de bord) peuvent être mis à l’échelle indépendamment, ce qui optimise les ressources et améliore les performances globales.
* 🔹 **Déploiement et intégration continue** : les microservices permettent une intégration progressive des fonctionnalités dans un pipeline CI/CD (Continuous Integration / Continuous Deployment), ce qui favorise les livraisons fréquentes et sécurisées.
* 🔹 **Alignement avec les pratiques modernes** : l’architecture microservices s’inscrit dans les standards actuels du développement logiciel, notamment dans les environnements cloud et DevOps.
* 🔹 **Adaptabilité aux besoins futurs** : elle offre une base robuste et évolutive qui pourra accueillir facilement de nouvelles fonctionnalités, voire intégrer des technologies ou services externes à l’avenir.

En résumé, l’architecture microservices répond aux besoins de flexibilité, d’évolutivité et de robustesse de la plateforme. Elle constitue un choix stratégique pour assurer un développement agile, une meilleure gestion des ressources, et une préparation à long terme aux évolutions fonctionnelles du projet.

## Architecture micro services

### Principes de l’architecture microservices

L’architecture microservices est une approche de conception logicielle qui consiste à décomposer une application en un ensemble de petits services indépendants, chacun réalisant une fonctionnalité métier spécifique. Contrairement à l’architecture monolithique, où tous les composants sont regroupés dans une seule base de code, les microservices sont faiblement couplés, développés et déployés de manière autonome.

Chaque microservice :

* Possède sa propre base de code,
* Est géré par une équipe dédiée,
* Communique avec les autres services via des API (souvent REST ou messaging),
* Peut être développé dans un langage ou un framework différent si nécessaire,
* Est déployé indépendamment.

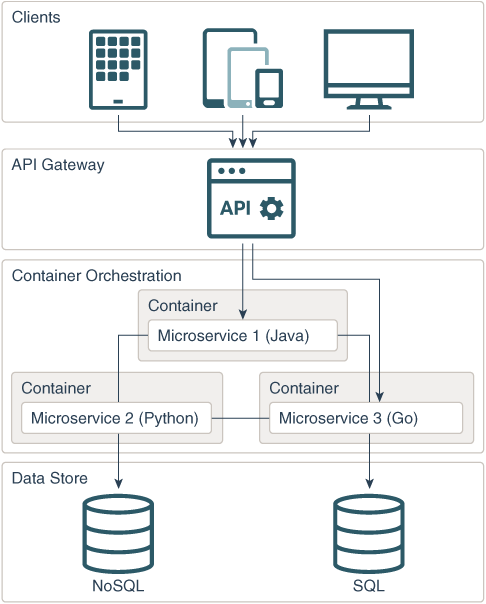
Cette approche favorise :

* **La scalabilité** (chaque service peut être mis à l’échelle indépendamment),
* **La résilience** (en cas de panne, seul un service est impacté),
* **L’agilité** (les équipes peuvent livrer plus fréquemment et indépendamment les unes des autres).

### Schéma de l’architecture microservices

Les microservices vous permettent de concevoir votre application en tant qu'ensemble de services faiblement interdépendants. Les microservices suivent le modèle de partage sans partage et s'exécutent en tant que processus sans état. Cette approche facilite le redimensionnement et la maintenance de l'application.

Pour illustrer ces principes, on peut présenter un **schéma type** d’une architecture microservices :



Le schéma de l'architecture d'une application composée de plusieurs microservices

Le schéma ci-dessus illustre une architecture microservices typique, décomposée en plusieurs couches fonctionnelles :

 **Clients** : Représentent les différents types de terminaux pouvant accéder à l'application (ordinateurs, smartphones, tablettes, etc.). Ces clients envoient des requêtes pour consommer les services fournis par l’application.

 **API Gateway (couche API)** : Il s’agit du point d’entrée unique de toutes les requêtes. L’API Gateway joue un rôle de répartiteur intelligent, redirigeant chaque requête vers le microservice approprié. Elle peut également intégrer des fonctions transversales telles que l’authentification, la journalisation, ou encore la gestion de la charge.

 **Couche logique**: Elle regroupe l’ensemble des microservices, chacun étant responsable d’une tâche métier bien définie. Cette couche est conçue pour assurer une faible dépendance entre les services, facilitant la maintenance, le développement et le déploiement continu.

* **Microservices** : Chaque microservice correspond à une fonctionnalité métier précise. Il s'exécute de manière autonome, peut être développé dans un langage spécifique (ex. : Java, Python, Go) et est indépendant des autres services. Cela garantit une plus grande flexibilité technologique et organisationnelle.
* **Orchestration des conteneurs** : Les microservices sont déployés dans des conteneurs légers et isolés. L’orchestration de ces conteneurs (via des outils comme Kubernetes) permet d’automatiser leur déploiement, de gérer leur montée en charge, leur mise à jour, et leur tolérance aux pannes. Elle joue un rôle clé dans la stabilité et la scalabilité de la couche logique.

 **Couche de persistance (Data Stores)**: fournit un mécanisme de persistance, tel qu'un moteur de stockage de base de données, des fichiers journaux, etc. Chaque microservice dispose de sa propre base de données, ce qui permet d’éviter les dépendances interservices sur les données. Le choix entre base de données SQL ou NoSQL dépend des besoins spécifiques du service. Cette séparation garantit la cohérence, la sécurité et la performance des accès aux données.

**Remarques clés** :

* Chaque service est **sans état** (stateless), ce qui facilite la scalabilité.
* L’indépendance des microservices permet un **déploiement autonome** de chaque module.
* Cette architecture favorise une **meilleure tolérance aux pannes** et une **agilité accrue** en développement.

## Spring boot dans le contexte micro services

**Spring Boot** est un framework open-source basé sur le framework Spring, conçu pour simplifier le développement d’applications Java, notamment dans un environnement microservices. Il permet de créer des applications **autonomes**, **facilement configurables** et **prêtes à être exécutées**, en éliminant la complexité liée à la configuration manuelle.

Spring Boot repose sur trois principes fondamentaux :

* **Convention plutôt que configuration** : des valeurs par défaut intelligentes pour éviter la surcharge de configuration.
* **Démarrage rapide** : grâce aux starters (dépendances préconfigurées pour chaque besoin).
* **Exécution autonome** : chaque service peut être exécuté comme une application indépendante via un simple java -jar

### Avantage de Spring Boot pour les microservices

**Développement rapide:**

Spring Boot simplifie la configuration et permet aux développeurs de démarrer rapidement des projets de microservices.

**Applications autonomes :**

Il permet de créer des applications autonomes, prêtes à être exécutées sans nécessiter de serveur d'applications externe (Tomcat embarqué, par exemple).

**Configurations par défaut:**

Spring Boot offre des configurations par défaut qui réduisent le besoin de configuration manuelle, rendant le développement plus efficace.

**Démarreurs (starters):**

Les démarreurs Spring Boot préconfigurent les dépendances, facilitant l'ajout de nouvelles fonctionnalités.

**Microservices légers:**

Spring Boot permet de créer des microservices avec un faible encombrement, ce qui est idéal pour les architectures basées sur des microservices.

**Prise en charge de REST :**

Spring Boot est bien adapté pour le développement d'API REST, qui est un composant clé des architectures microservices.

**Spring Cloud pour compléter Spring Boot:**

Bien que Spring Boot permette le développement de microservices, il est souvent utilisé avec Spring Cloud pour fournir des fonctionnalités supplémentaires telles que la gestion de la configuration, la découverte de services et la résilience, le routage dynamique, la configuration centralisée, le monitoring, etc. Spring Cloud étend les capacités de Spring Boot pour les environnements de microservices distribués.

**Support de la production** : avec des outils comme **Spring Boot Actuator**, on peut surveiller l’état des microservices (santé, métriques, logs…) via des endpoints REST sécurisés

En résumé : Spring Boot est un excellent choix pour le développement de microservices grâce à sa simplicité, sa rapidité et sa capacité à créer des applications autonomes, tandis que Spring Cloud complète ces fonctionnalités dans un contexte de microservices distribués.

### Structure typique d’un microservice spring boot

On peut architecturer une application **Spring Boot** de différentes manières, le framework n'impose pas de façon particulière d'organiser notre code. Cependant, il existe une architecture très utilisée dans le monde des applications web : **l'architecture n-tiers**.

Un microservice Spring Boot est généralement structuré en différentes couches, selon l’architecture logicielle en couches :

**1. Couche de présentation (Controller) :**

* Gère les requêtes HTTP, reçoit les données d'entrée, les valide, et renvoie les réponses.
* Utilise des annotations comme @RestController, @RequestMapping, etc. pour définir les points de terminaison du microservice.

**2. Couche métier (Service):**

* Contient la logique métier principale du microservice.
* Implémente les opérations spécifiques au service, comme la création, la lecture, la mise à jour ou la suppression de données.

**3. Couche de persistance (Repository) :**

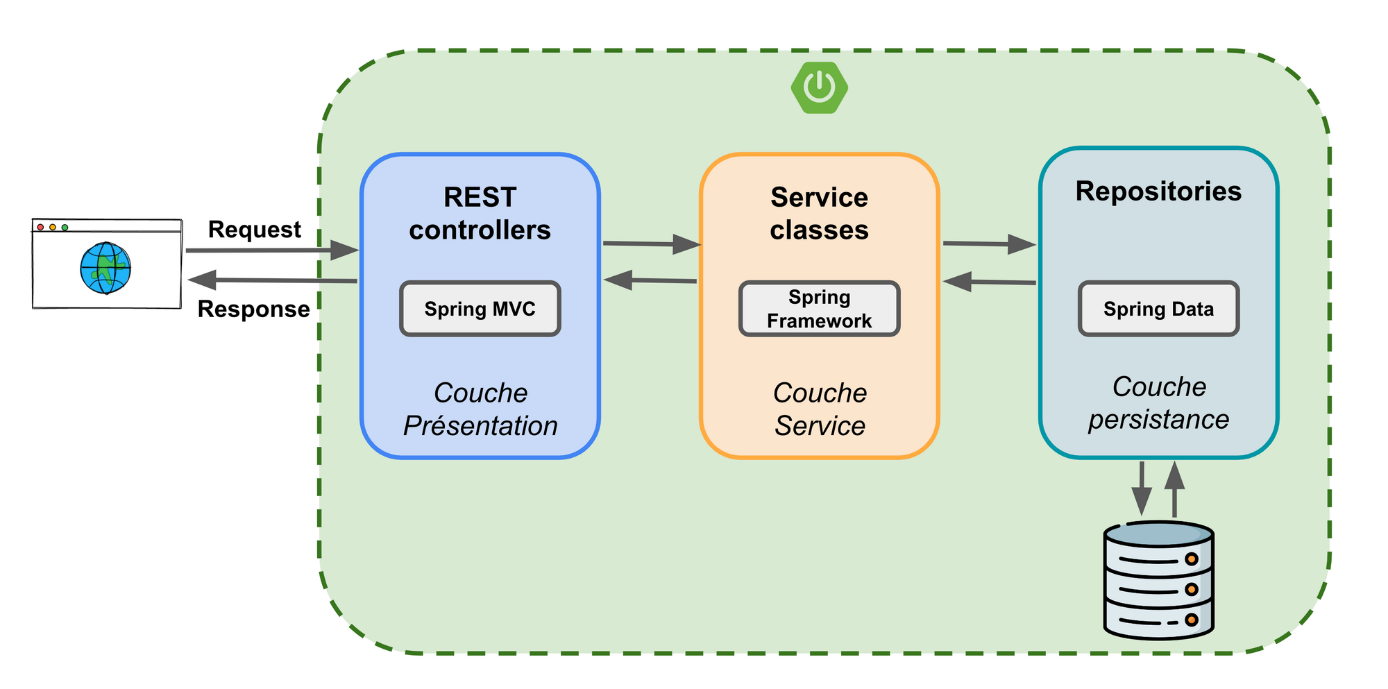
* Interface avec la base de données ou à d'autres sources de données.
* Peut utiliser JPA (Java Persistence API) pour interagir avec la base de données, et des annotations comme @Repository pour définir les opérations de persistance.

**4. Entities** : représentent les modèles de données mappés à la base via JPA (@Entity).

**5. Configuration:**

* Spring Boot permet une configuration automatique basée sur les dépendances présentes dans le classpath.
* Utilisation de fichiers de propriétés (application.properties ou application.yml) pour configurer le microservice, y compris la connexion à la base de données, les paramètres du serveur HTTP, etc.

Cette organisation modulaire permet une meilleure lisibilité du code, une séparation claire des responsabilités, et facilite les tests unitaires.



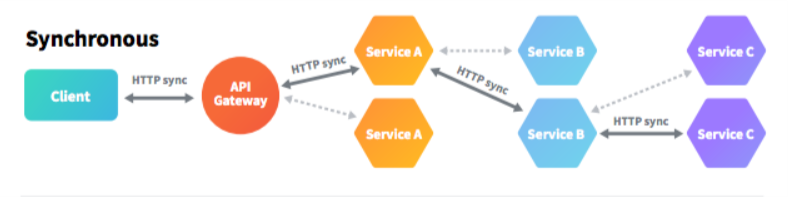
Architecture en couches d’un microservice avec Spring Boot

## Communication entre microservices

Dans une architecture microservices, les différents services doivent collaborer pour fournir des fonctionnalités cohérentes à l’utilisateur. La **communication interservices** devient donc un enjeu fondamental. Deux grandes approches sont généralement utilisées : **la communication synchrone** (ex. : via REST) et **la communication asynchrone** (ex. : via des messages).

### **Communication synchrone**

La communication synchrone repose sur un modèle de type demande-réponse immédiate. Concrètement, cela signifie qu’un service en appelle un autre et attend sa réponse avant de poursuivre son exécution. Ce mode de communication est souvent mis en œuvre à l’aide du protocole HTTP, en utilisant des API REST et des formats standards comme JSON. Ce modèle présente plusieurs avantages : il est simple à implémenter, permet une intégration rapide à l’aide d’outils comme Swagger ou Postman, et repose sur des standards largement adoptés dans le développement web. Cependant, il comporte également certaines limites. En effet, il crée une forte dépendance vis-à-vis de la disponibilité du service distant. Si ce dernier met du temps à répondre ou devient indisponible, cela peut entraîner des problèmes de latence ou de blocage dans l’ensemble du système. Un exemple typique de communication synchrone est un service "Projet" qui interroge en temps réel un service "Collaborateurs" afin de vérifier la disponibilité d’un membre avant de lui affecter une tâche.



Modèle de communication synchrone via REST.

La figure ci-dessus illustre un modèle de communication synchrone dans une architecture microservices. Dans ce cas, un client envoie une requête via une API Gateway. Cette requête est ensuite transmise successivement entre les services A, B, puis C, à l’aide d’appels HTTP REST. Chaque service attend la réponse du suivant avant de continuer son traitement. Cette chaîne séquentielle implique une forte dépendance temporelle : si l’un des services est lent à répondre ou indisponible, c’est l’ensemble de la chaîne qui est impacté, bloquant ainsi la réponse au client. Ce modèle présente l’avantage d’être simple à mettre en œuvre, notamment grâce à l’utilisation de protocoles standardisés. Toutefois, cette simplicité a un coût en termes de robustesse et de résilience du système.

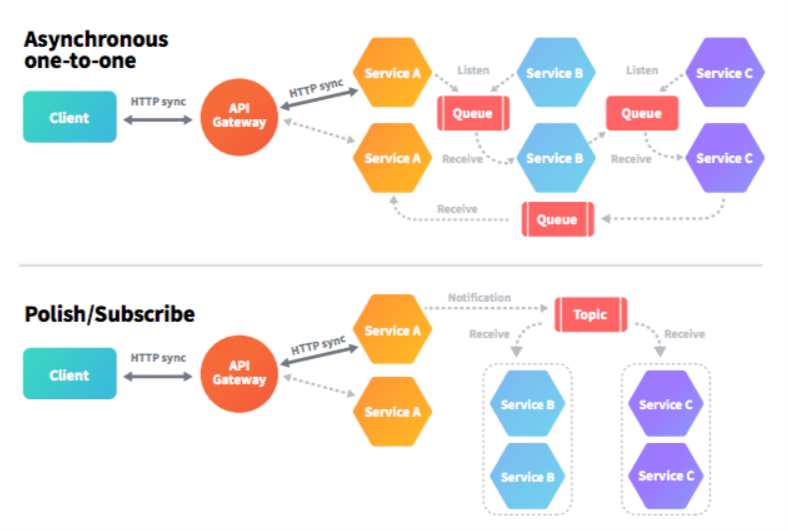
### **Communication asynchrone**

À l’inverse, la communication asynchrone repose sur un échange indirect de messages via une file d’attente ou un système de messagerie, tel que Kafka ou RabbitMQ. Dans ce modèle, un service producteur publie un événement dans un canal de communication, et un ou plusieurs services consommateurs réagissent à cet événement de manière indépendante, sans bloquer le producteur. Ce découplage permet d’obtenir une architecture plus résiliente, car les services ne dépendent pas directement les uns des autres pour fonctionner. Ce type de communication est particulièrement adapté au traitement d’événements, à la gestion de logs, ou à l’exécution de tâches en arrière-plan. Néanmoins, l’implémentation de la communication asynchrone peut s’avérer plus complexe. Elle nécessite la gestion fine des files de messages, l’assurance de l’idempotence, ainsi qu’un contrôle rigoureux de l’ordre des messages. De plus, les tests et le débogage peuvent être plus délicats à réaliser en raison de la nature non linéaire des échanges.

La figure suivante illustre **deux modèles de communication asynchrone** entre microservices.

Dans le **premier modèle**, basé sur une **file de messages (one-to-one)**, le **service A** envoie un message dans une **queue** dédiée. Un seul **service B** est chargé de consommer ce message lorsqu’il est prêt. Ce modèle favorise un **découplage fort** : le producteur n’a pas besoin d’attendre une réponse immédiate, ce qui améliore la **résilience** du système. Même si le service consommateur est temporairement indisponible, le message est conservé jusqu’à son traitement. Des solutions comme **RabbitMQ**, **ActiveMQ** ou **Amazon SQS** sont souvent utilisées dans ce cas. Toutefois, cette approche peut introduire une **complexité supplémentaire** pour le suivi des messages, notamment en cas d’échec, de perte ou d’ordre de traitement non garanti.

Le **second modèle**, appelé **publish/subscribe**, repose sur la diffusion d’un message publié par le **service A** dans un **topic**, qui agit comme un canal de diffusion. Tous les services abonnés à ce topic (par exemple les services B et C) **reçoivent simultanément** le même message et peuvent le traiter indépendamment. Ce modèle est particulièrement adapté aux cas d’usage comme les **notifications**, les **alertes**, le **logging** ou encore la **diffusion d’événements métiers** à plusieurs modules. Il permet une **extensibilité accrue** de l’architecture, mais nécessite une **gestion rigoureuse des abonnements** et une organisation efficace du traitement parallèle. Des outils comme **Apache Kafka**, **MQTT** ou **Google Pub/Sub** sont couramment utilisés dans ce type de communication.



Modèles de communication asynchrone (file vs topic).

### **Choix du mode de communication dans notre projet**

Dans le cadre de notre projet, nous avons choisi d’utiliser **une communication synchrone basée sur REST**, facilitée par **Spring Cloud OpenFeign**, pour les raisons suivantes :

🔹 **Cohérence avec la logique métier** : certaines fonctionnalités, comme l’affectation automatique ou le suivi en temps réel, nécessitent une réponse immédiate d’un autre service.

🔹 **Simplicité d’implémentation** : Feign Client permet de consommer des API REST de manière déclarative, avec peu de configuration, ce qui accélère le développement.

🔹 **Facilité de débogage et de tests** : les appels REST peuvent être testés individuellement avec des outils simples, ce qui est idéal pendant la phase de prototypage.

🔹 **Tolérance aux pannes** : bien que Feign repose sur une communication synchrone classique (appel et attente de réponse), nous avons intégré Resilience4j avec un circuit breaker. Cela nous permet, en cas d’indisponibilité du service distant, de basculer automatiquement vers une méthode de **secours (fallback)**. Ainsi, la résilience est améliorée sans changer le mode de communication.

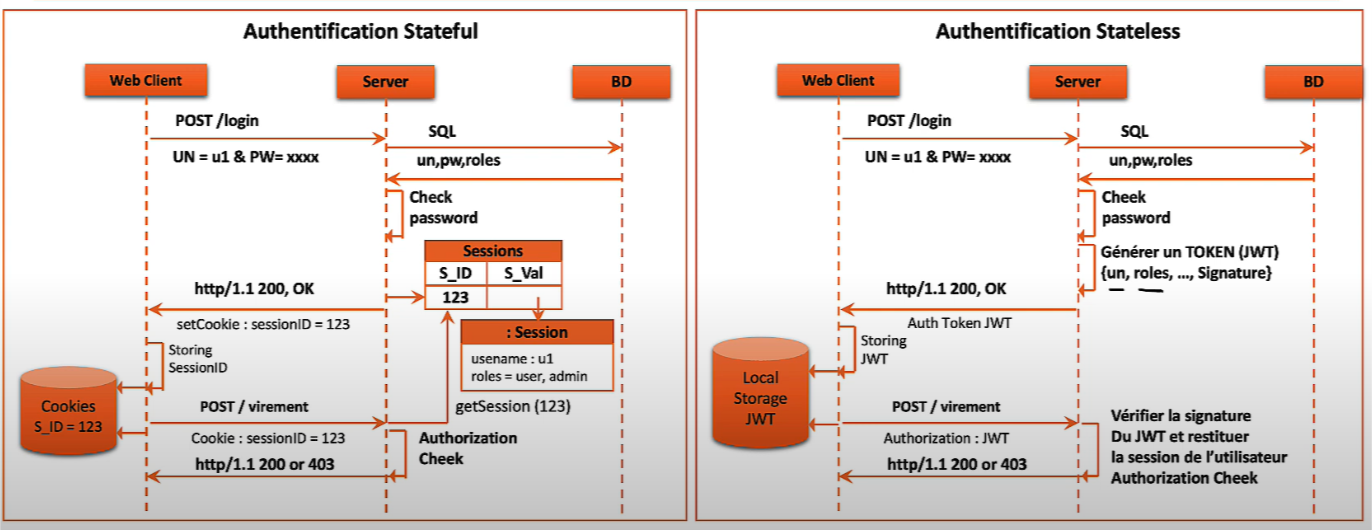
🔹 **Pas de besoin immédiat d’asynchrone** : notre projet ne manipule pas de volumes massifs de messages ni de traitements différés, ce qui rend l’approche synchrone suffisante pour cette première version.

Cependant, l’architecture reste **ouverte à une évolution vers une communication asynchrone** (via Kafka ou RabbitMQ), notamment si des modules de notifications, d’alertes ou de traitements différés sont ajoutés ultérieurement.

## Sécurité des Systèmes Distribués

### Système d’authentification : Stateful vs Stateless

Dans le cadre des systèmes distribués, la gestion de l’authentification représente une composante essentielle de l’architecture de sécurité. Elle permet de garantir que seules les entités légitimes peuvent accéder aux ressources ou interagir avec les services. Deux approches dominantes coexistent pour la gestion des sessions d’authentification : le modèle stateful et le modèle stateless. Le choix entre ces deux paradigmes influe considérablement sur la scalabilité, la robustesse et la facilité d’intégration du système.



Authentification Stateful vs Stateless

**a) Authentification Stateful**

Dans un modèle d’authentification stateful, les informations relatives à la session d’un utilisateur sont conservées côté serveur. Lorsqu’un utilisateur s’authentifie, le serveur crée une session unique identifiée par un identifiant de session (Session ID), généralement transmis au client via un cookie. À chaque requête ultérieure, le client renvoie ce Session ID, permettant au serveur de retrouver le contexte de session correspondant.

Ce modèle repose donc sur la persistance de l’état côté serveur, que ce soit en mémoire ou dans une base de données. Le serveur est chargé de valider la session à chaque interaction, ce qui implique une gestion centralisée des sessions et peut engendrer une charge importante en cas de forte affluence. Cette approche est relativement simple à implémenter, notamment dans les architectures monolithiques. Toutefois, elle présente des limitations dans un environnement distribué, où la gestion de sessions partagées entre plusieurs instances de serveurs devient complexe et coûteuse en termes de coordination et de cohérence.

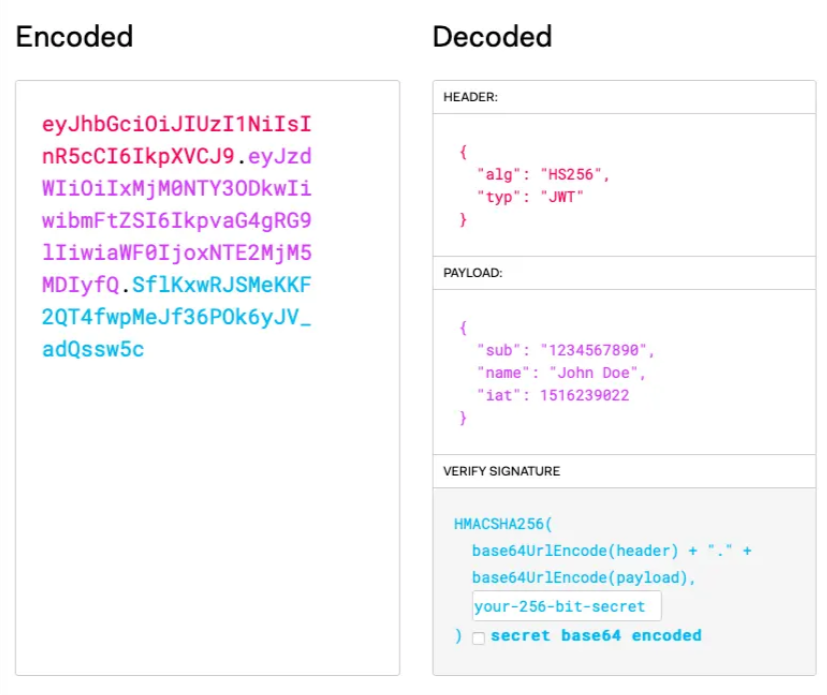
**b) Authentification Stateless**

Le modèle stateless, en revanche, repose sur l’absence de stockage d’état côté serveur. Lors de l’authentification, un jeton signé numériquement – le plus souvent un JSON Web Token (JWT) – est généré et transmis au client. Ce jeton encapsule l’ensemble des informations nécessaires à l’identification de l’utilisateur, telles que son identité, ses rôles et la date d’expiration du jeton. À chaque requête, le client joint ce jeton, que le serveur peut valider sans interroger une session persistante.

L’approche stateless est particulièrement bien adaptée aux architectures modernes basées sur les microservices, car elle simplifie la montée en charge et l’indépendance des services. En supprimant la dépendance à un stockage centralisé des sessions, elle permet une meilleure scalabilité horizontale. Toutefois, ce modèle présente certaines contraintes, notamment en ce qui concerne la révocation immédiate des jetons, qui nécessite des mécanismes additionnels pour assurer une sécurité fine (listes de révocation, durées de vie courtes, etc.).

### JSON Web Token

Dans les environnements distribués, en particulier ceux reposant sur une architecture à base de microservices, le besoin d’un mécanisme d’authentification robuste, scalable et indépendant de l’état (Stateless) devient essentiel. Le **JSON Web Token (JWT)** s’impose comme une solution standardisée, permettant non seulement d’assurer l’authentification des utilisateurs, mais également de garantir l’intégrité et l’authenticité des données échangées entre services. L’un des atouts majeurs de cette approche réside dans sa capacité à instaurer une **confiance numérique** entre les composants sans nécessiter de persistance côté serveur.



Exemple d’un JSON Web Token (JWT)

#### a) Structure d’un JWT

Un token JWT est constitué de trois segments encodés en Base64URL, séparés par des points. Ces segments sont les suivants :

**Header** : il contient des métadonnées sur le token, notamment le type de jeton (JWT) et l’algorithme de signature utilisé (alg), tel que HS256 (HMAC avec SHA-256) ou RS256 (RSA avec SHA-256).

**Payload** : il renferme les claims (déclarations) qui peuvent être :

* Registered claims : champs standardisés tels que iss (issuer), sub (subject), exp (expiration), aud (audience).
* Public claims : définitions ouvertes selon des conventions d’usage.
* Private claims : informations métier spécifiques, comme user\_id, roles, etc.

**Signature** : générée par l’application d’un algorithme de hachage spécifié dans le header et d’une clé (secrète ou privée) sur les deux premières parties. Elle permet de garantir que le contenu du token n’a pas été altéré.

Cette structuration assure une lecture simple, mais surtout une vérification fiable de l’authenticité du jeton.

#### b) Encodage, hachage et signature

Bien que le JWT soit encodé en Base64URL pour faciliter son transport dans les entêtes HTTP, il ne s’agit pas d’un mécanisme de sécurité à proprement parler : les données peuvent être décodées aisément par toute personne ayant accès au jeton.

La **sécurité repose principalement sur la signature numérique**, obtenue par une fonction de hachage (par exemple SHA-256) appliquée au header et au payload, combinée à une clé.

Deux modèles cryptographiques sont couramment utilisés :

* **HMAC (symétrique)** : repose sur une clé secrète partagée entre les services pour signer et vérifier le token. Bien que simple, ce modèle est moins adapté aux architectures distribuées, en raison de la difficulté de gestion du secret partagé à grande échelle.
* **RSA (asymétrique)** : utilise une clé privée pour signer le token (côté serveur d’authentification), et une clé publique pour le vérifier (côté consommateurs). Ce modèle est bien plus adapté aux environnements multi-services, car seule la clé publique doit être diffusée.

#### c) Vérification du JWT et confiance numérique

Lorsqu’un service reçoit un JWT, il procède à une vérification en recalculant la signature à partir du header et du payload reçus, selon l’algorithme indiqué, et en la comparant à la signature fournie dans le jeton.

* Si la signature est valide, le token est considéré comme **authentique** et non modifié.
* Si la signature est invalide, cela signifie que le contenu du token a été altéré, volontairement ou non, et il ne doit donc pas être accepté.

Ce mécanisme cryptographique permet de garantir une **chaîne de confiance numérique** sans avoir à interroger un serveur central à chaque requête. Il permet également de déléguer l’authentification à un service dédié tout en autorisant la vérification localement au sein des microservices.

#### d) Limites du modèle HMAC et avantages du modèle RSA

L’utilisation de HMAC dans un contexte distribué pose plusieurs difficultés :

* Le partage du secret entre plusieurs services constitue un **point de vulnérabilité majeur**, augmentant le risque de compromission.
* La gestion de ce secret à grande échelle complexifie la maintenance du système.

À l’inverse, l’utilisation d’algorithmes **asymétriques** comme RSA permet de centraliser la signature (par le serveur d’authentification) tout en **déléguant la vérification** aux services consommateurs via une clé publique, ce qui renforce la sécurité globale et facilite l’évolutivité du système.

#### e) Gestion des secrets dans une architecture distribuée

Que ce soit pour des clés symétriques (HMAC) ou asymétriques (RSA), la **gestion sécurisée des secrets** est un élément critique. Des outils spécialisés, tels que **Vault** de HashiCorp, permettent de :

* Stocker les secrets et clés cryptographiques de manière sécurisée.
* Définir des politiques d’accès granulaires basées sur les rôles et services.
* Automatiser la **rotation des clés** afin de réduire la surface d’exposition.

Dans le cas d’un modèle asymétrique, Vault peut également être utilisé pour distribuer dynamiquement les **clés publiques** aux microservices tout en conservant la **clé privée isolée** et protégée.

### Le protocole OAuth2

Le protocole **OAuth 2.0** est un standard d’autorisation largement adopté dans les architectures modernes, notamment pour permettre à des applications tierces d’accéder de manière limitée et sécurisée aux ressources d’un utilisateur, sans avoir à transmettre ni stocker ses identifiants. Il repose sur le principe de delegated access, dans lequel un **serveur d’autorisation** émet un **jeton d’accès** (access token) après que l’utilisateur a consenti à l’accès à ses données. Ce jeton est ensuite utilisé par l’application cliente pour effectuer des requêtes auprès des services protégés. OAuth 2.0 prend en charge plusieurs **flux d’authentification** (authorization code, implicit, client credentials, etc.) adaptés à différents cas d’usage (applications web, mobiles, serveurs backend, etc.). Il introduit également des rôles distincts : resource owner, client, authorization server, et resource server. Bien qu’il ne gère pas directement l’authentification de l’utilisateur, OAuth 2.0 constitue la base sur laquelle des protocoles plus complets, comme OpenID Connect, sont construits.

### Le protocole OpenID Connect

**OpenID Connect** est une extension du protocole OAuth 2.0 qui introduit une couche d’**authentification** au-dessus du mécanisme d’autorisation. Contrairement à OAuth 2.0 qui ne spécifie pas comment identifier un utilisateur, OIDC permet à une application cliente de vérifier l’identité d’un utilisateur final et d’obtenir des informations de profil de manière sécurisée. Pour ce faire, il s’appuie sur l’émission d’un **ID Token** au format JWT, signé par le **serveur d’identités (Identity Provider)**. Ce jeton contient des claims relatifs à l’utilisateur (nom, email, identifiant, etc.) ainsi que des métadonnées d’authentification (durée de session, audience, etc.). OIDC permet ainsi une **authentification fédérée**, idéale pour les systèmes distribués ou les applications souhaitant déléguer l’identification à un fournisseur tiers tel que Google, Microsoft, ou un serveur d’identité interne. Grâce à sa simplicité et sa compatibilité avec OAuth 2.0, OpenID Connect est devenu un standard incontournable pour l’authentification sécurisée dans les environnements modernes.

### Sécurité avec spring Security

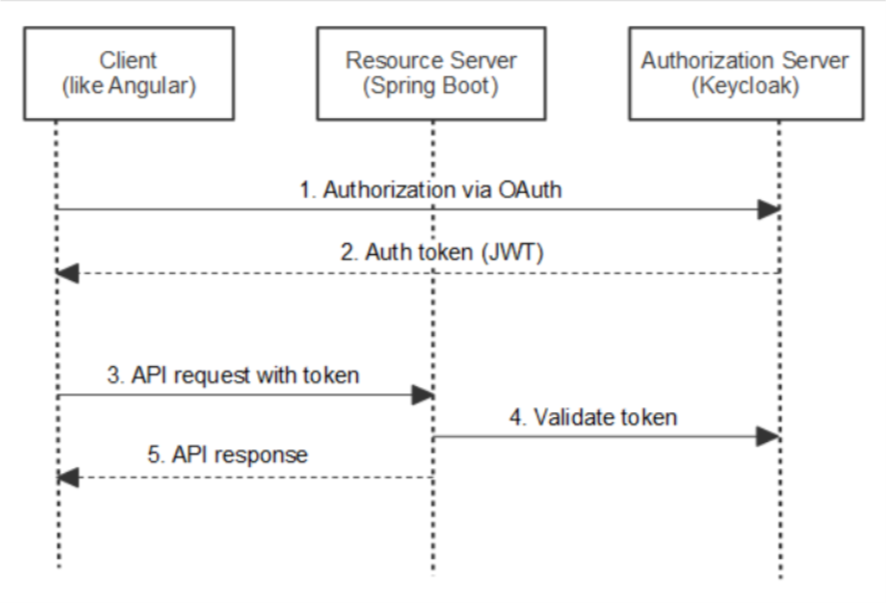
**Spring Security** est un framework robuste et largement utilisé dans l’écosystème Java pour la mise en œuvre de mécanismes de sécurité au sein des applications. Il fournit un ensemble complet de fonctionnalités permettant de garantir à la fois **l’authentification** des utilisateurs ou des services (vérification de leur identité) et **l’autorisation** (contrôle des accès selon des rôles définis, tels que ADMIN, USER, etc.). Il permet également de sécuriser les points d’accès REST à travers des annotations déclaratives telles que @PreAuthorize ou @Secured, offrant ainsi un contrôle granulaire des permissions.

Dans les architectures modernes orientées microservices, Spring Security s’intègre aisément avec des technologies comme **JWT (JSON Web Token)**, facilitant une authentification **stateless**, adaptée aux environnements distribués. Par ailleurs, le framework intègre des mécanismes essentiels tels que le **hachage sécurisé des mots de passe**, la **protection contre les attaques CSRF**, et bien d’autres vulnérabilités courantes.

**Toutefois, dans une architecture distribuée, Spring Security seul peut vite devenir complexe à gérer, notamment pour la mise en place du Single Sign-On (SSO) ou la gestion centralisée des utilisateurs. Ce niveau de configuration nécessite une solide expertise, et en l’absence d’une maîtrise approfondie, on risque de laisser des failles de sécurité critiques.** C’est pourquoi il est fortement recommandé de **déléguer cette responsabilité à un Identity Provider robuste et éprouvé**, comme **Keycloak**, qui offre des mécanismes d’authentification, d’autorisation et de gestion des rôles déjà sécurisés, centralisés et facilement intégrables.

### Keycloak

Dans le cadre des systèmes distribués modernes, la gestion centralisée de l’authentification et de l’autorisation devient indispensable pour garantir la cohérence, la scalabilité et la sécurité. **Keycloak** est une solution open source de gestion des identités et des accès (IAM) qui permet de mettre en place facilement des mécanismes tels que le **Single Sign-On (SSO)**, la gestion des utilisateurs, des rôles, et des politiques d’accès. Il s’intègre nativement avec les protocoles standards comme **OAuth2** et **OpenID Connect**, offrant ainsi une interopérabilité fluide avec les applications front-end (ex. Angular) et les API backend (ex. Spring Boot).



Processus d’authentification OAuth2 avec Keycloak dans une architecture distribuée

La figure ci-dessus illustre le processus d’authentification basé sur **OAuth2** avec **Keycloak** en tant que serveur d’autorisation, dans une architecture où le client est une application front-end (ex. Angular) et le serveur de ressources est développé avec **Spring Boot**.

1. **Authorization via OAuth** : L’utilisateur initie le processus d’authentification via le client. Celui-ci redirige la demande vers Keycloak, qui agit en tant que serveur d’autorisation.
2. **Auth token (JWT)** : Une fois l'utilisateur authentifié, Keycloak émet un **token d’accès** sous forme de **JWT** (JSON Web Token) que le client stocke et utilise pour accéder aux ressources.
3. **API request with token** : Le client effectue une requête vers le backend (Spring Boot) en incluant le token dans les en-têtes HTTP (souvent dans le champ Authorization).
4. **Validate token** : Le serveur de ressources vérifie la validité du JWT. Cette validation peut être faite localement (en vérifiant la signature via une clé publique) ou via une requête vers le serveur Keycloak.
5. **API response** : Si le token est valide et les autorisations suffisantes, la ressource demandée est renvoyée au client.

Ce fonctionnement repose sur une **authentification stateless**, ce qui le rend parfaitement adapté aux **architectures microservices**, tout en assurant une **séparation des responsabilités** entre l’authentification (gérée par Keycloak) et la logique métier (gérée par le backend).

## Gestion des données (spring data JPA)

## Infrastructure et déploiement

### Containerisation

### Orchestration

# Conclusion

# CHAPITRE III : ANALYSE ET CONCEPTION

## Introduction

## Analyse des besoins

### Besoins fonctionnels

### Besoins non fonctionnels

## Modélisation des besoins fonctionnels

### Diagramme de cas d’utilisation

### Diagramme de séquences

### Diagramme de classes

# Conclusion

# CHAPITRE IV : MISE EN ŒUVRE

## Introduction

## Outils de développement

### Front end

### Back end

## Réalisation

# Conclusion

# CONCLUSION GENERALE

## Synthèse

## Perspective

## Références

<https://dzone.com/articles/communicating-between-microservices>

<https://medium.com/@martinezmendietagerman/scaling-web-apps-stateful-vs-stateless-87ce157b8dcf>