BankDash.

# Application développée avec Spring Boot, MongoDB, Angular, JWT, Docker, Jenkins, et AWS

---

# Solution moderne pour la gestion et l'analyse des données bancaires

---

*Mené par :*
NEJIB Hamza

*Projet réalisé entre 14 Novembre 2024 et 30 Novembre 2024*

Village de l'Emploi

# Table des matières

# Table des figures

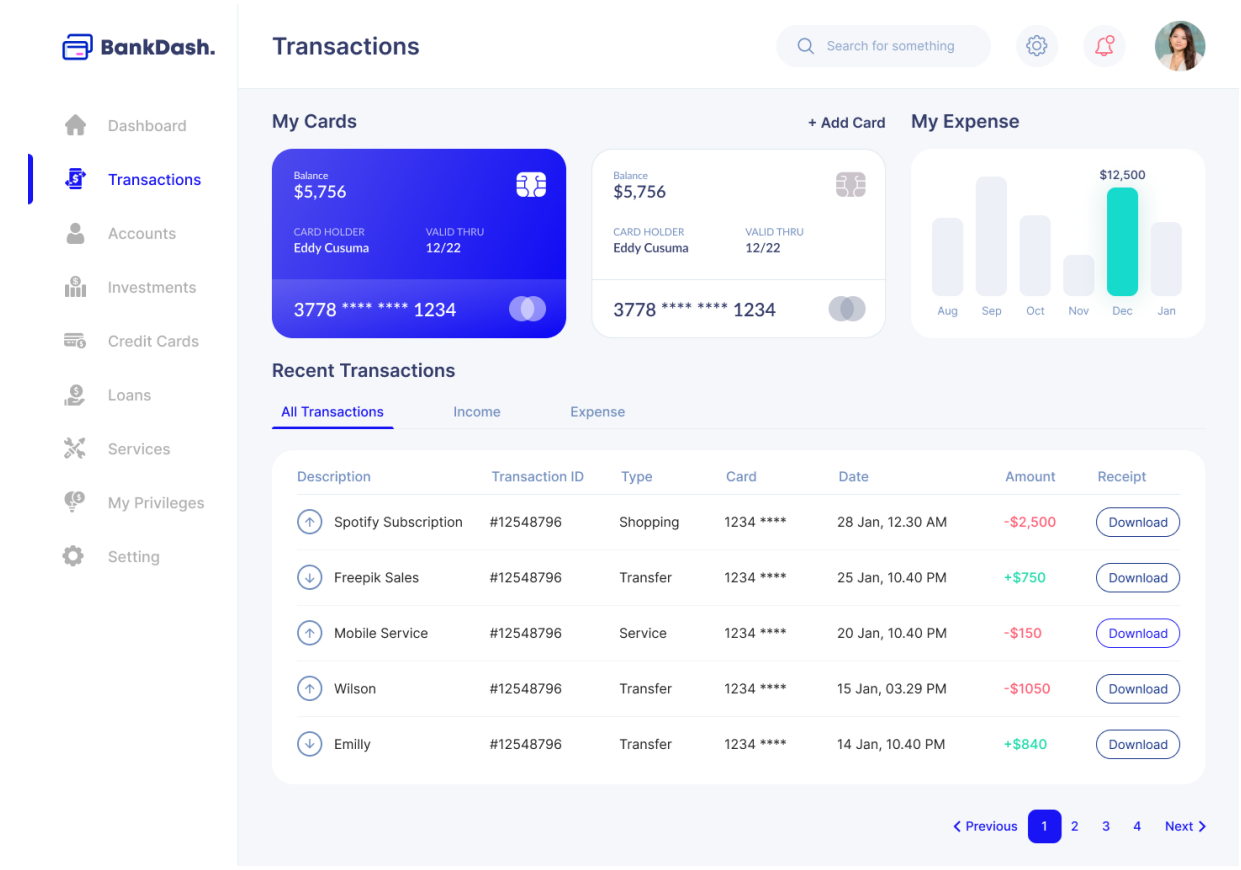
# 1 Résumé

BankDash est une application de gestion bancaire développée à partir d'une maquette récupérée sur Figma, l'application permet aux utilisateurs de visualiser leurs comptes, suivre leurs transactions, gérer leurs cartes de crédit, et effectuer des transferts rapides. L'application est divisée en une interface utilisateur (frontend) développée avec Angular, un backend en Spring Boot avec une base de données MongoDB, et des services de déploiement et CI/CD avec Docker, Jenkins, et AWS. J'ai récupéré la maquette sur Figma via le lien suivant : BankDash

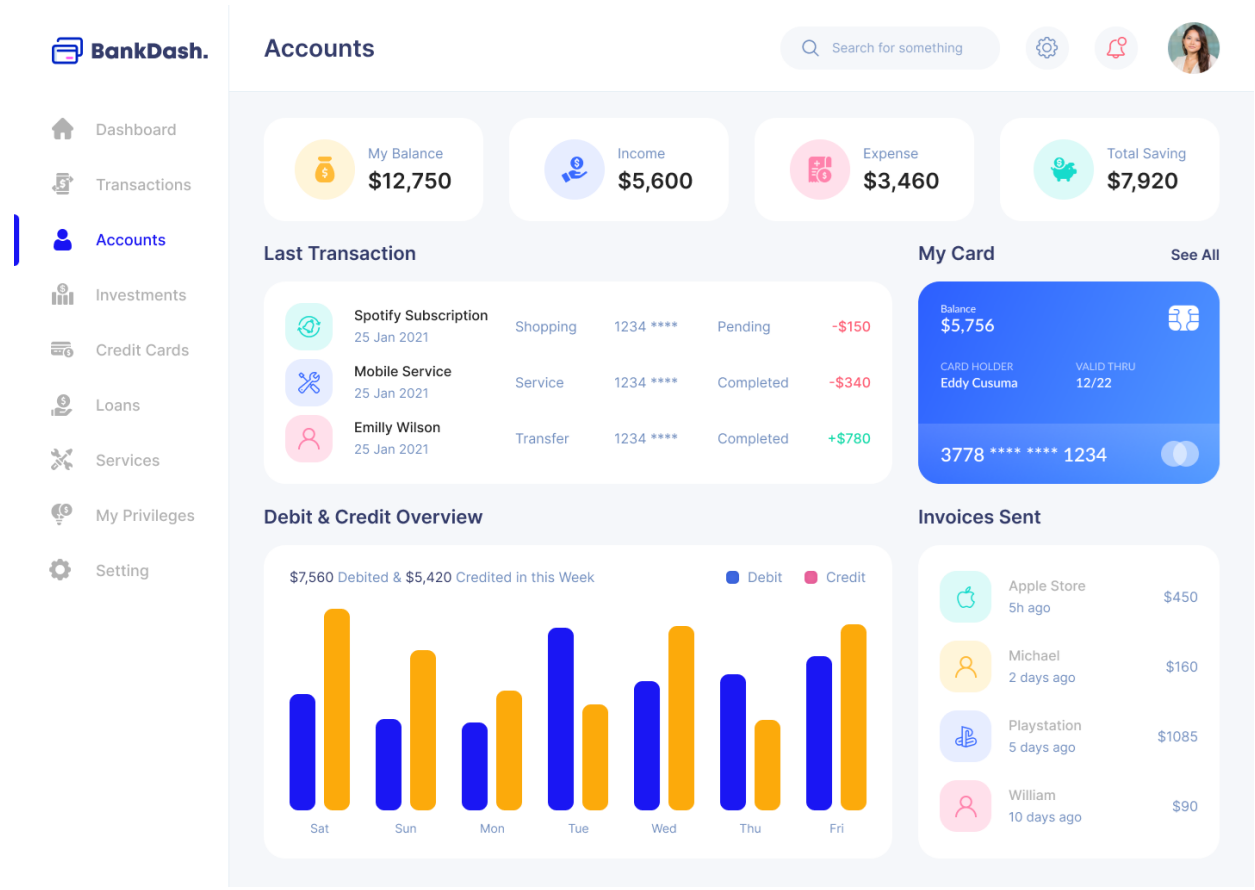
# 2 Pages de la maquette

### Page d'accueil (Dashboard)

## Transactions

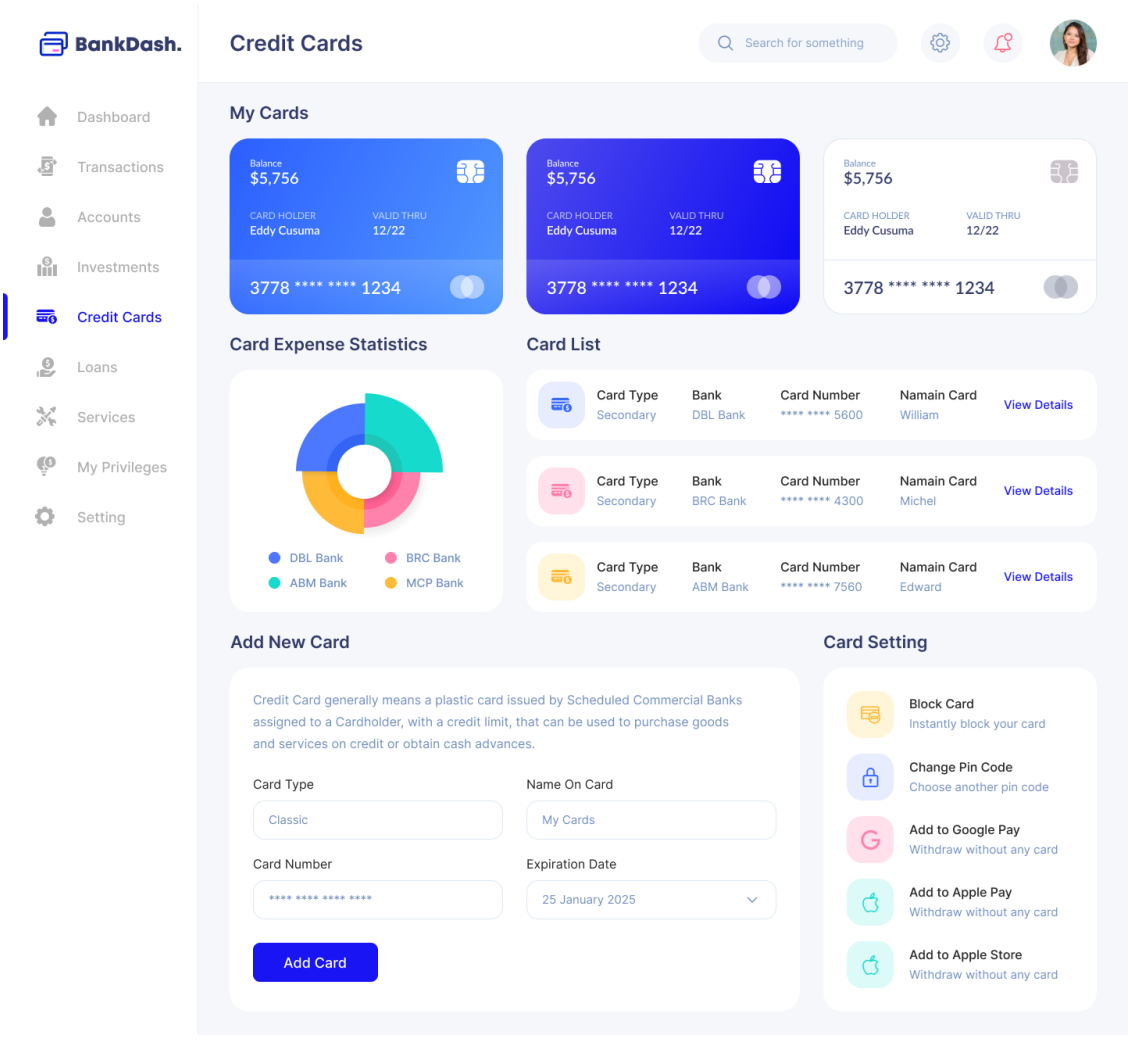



## Accounts

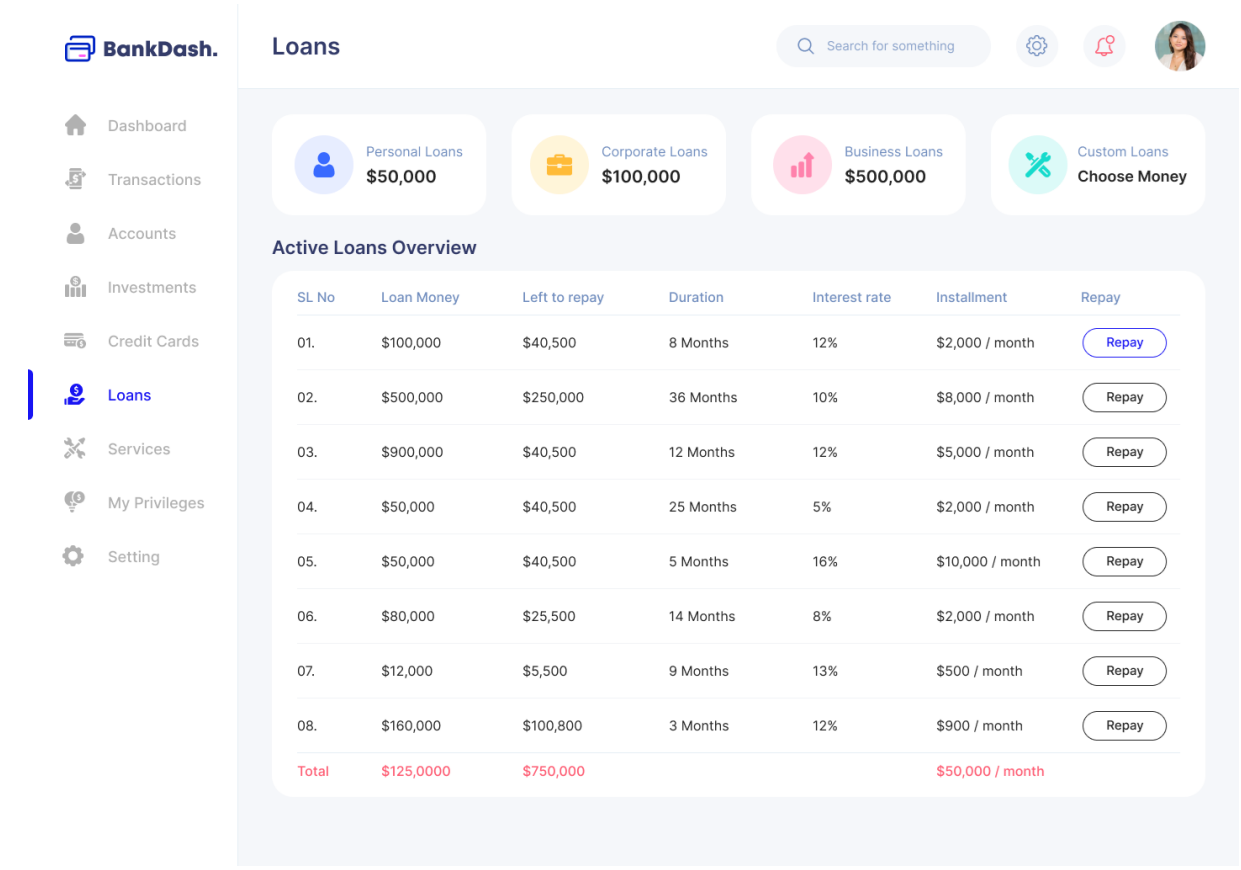


## Investments

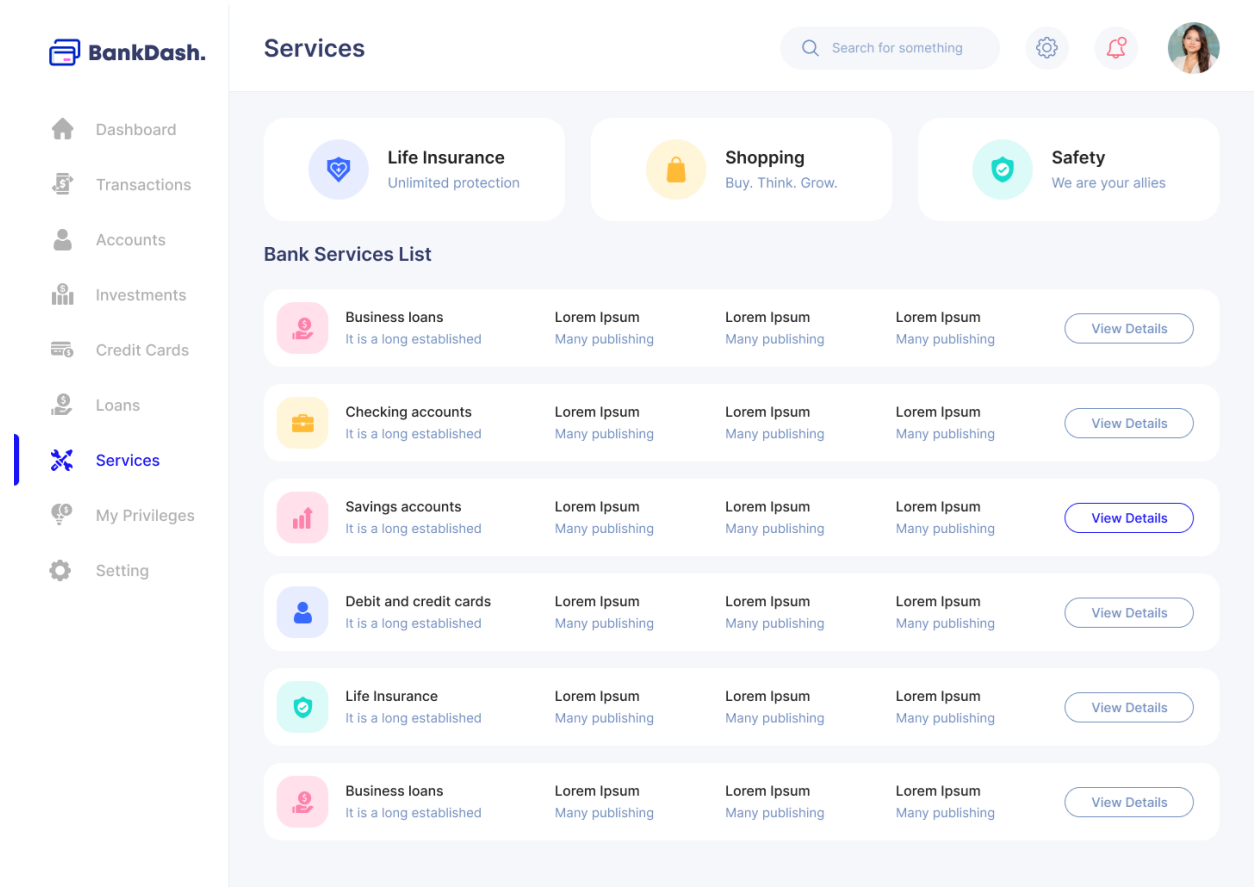

## Credit Cards



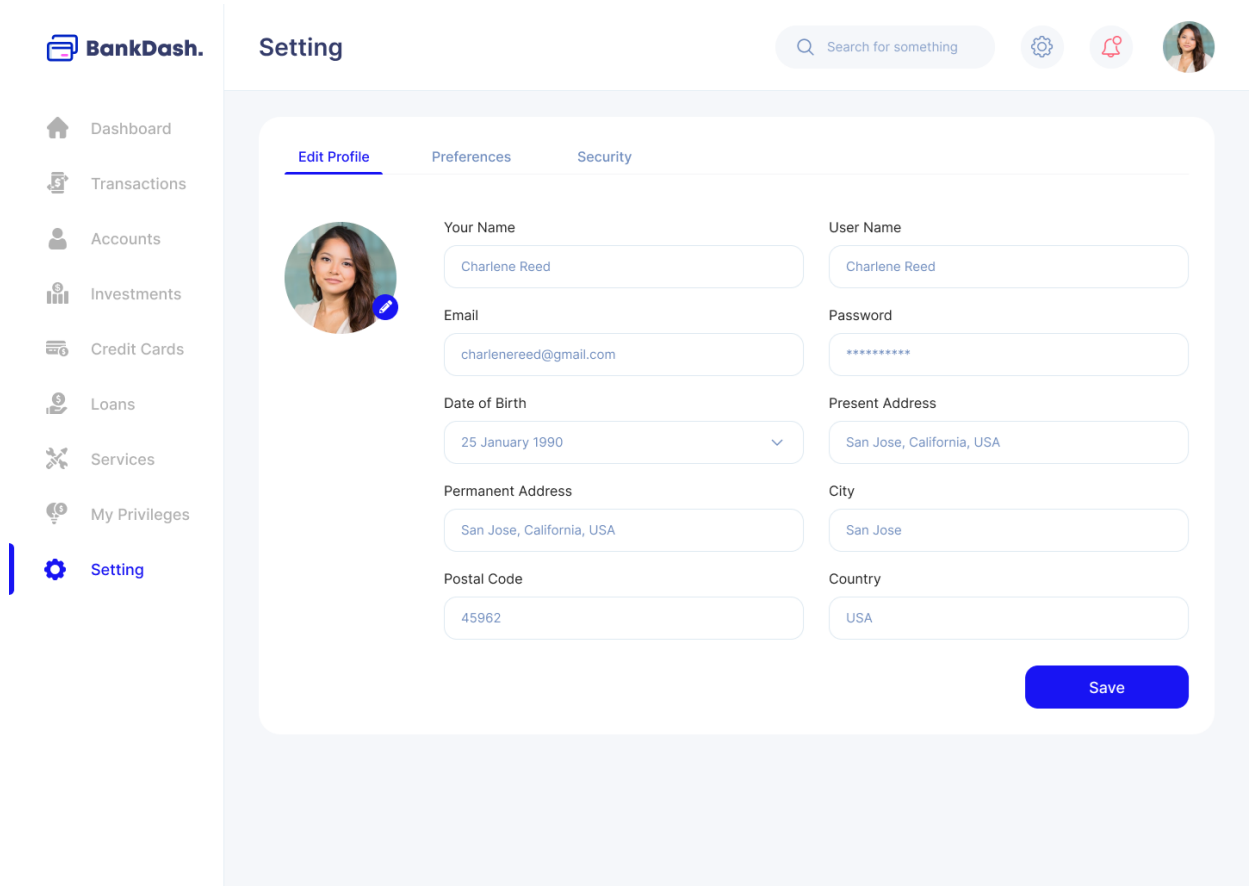
## Loans



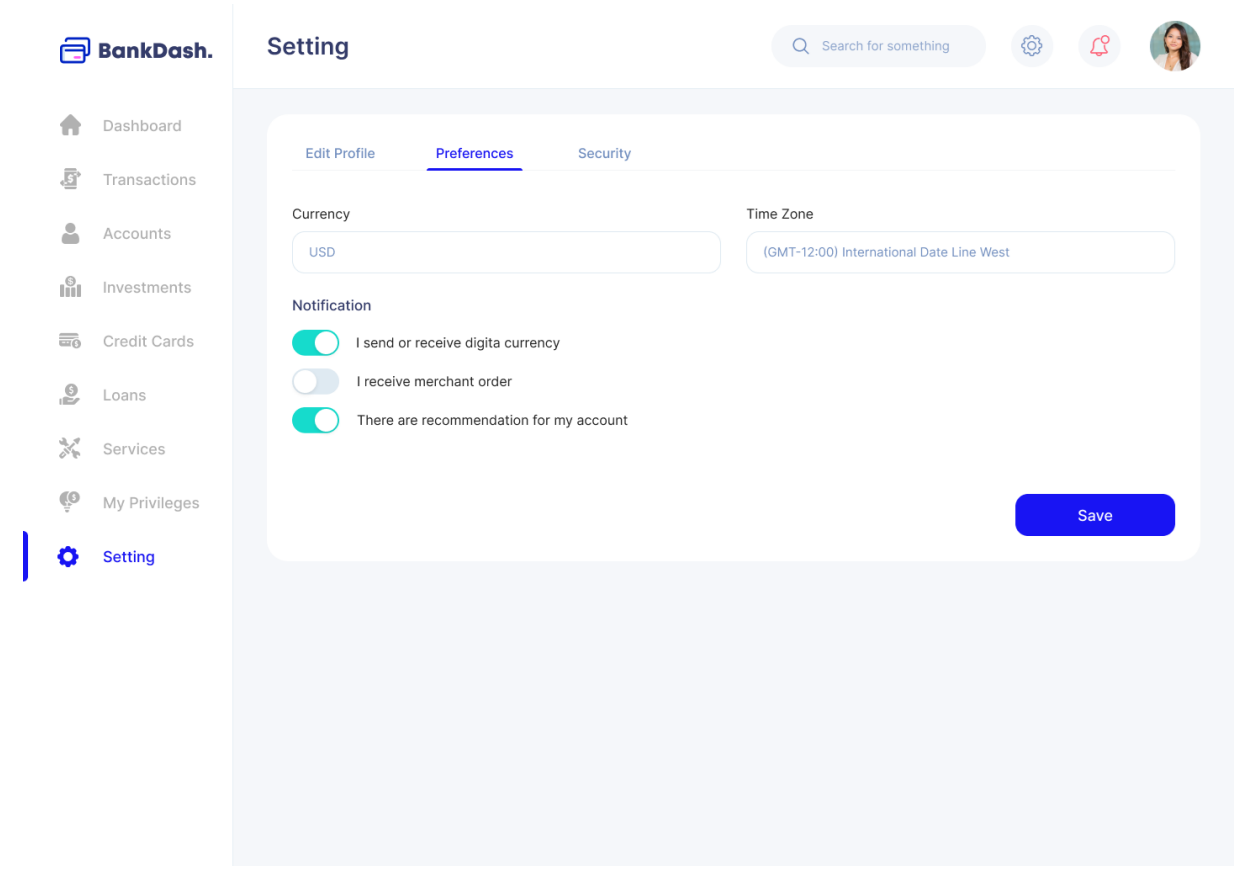

## Services



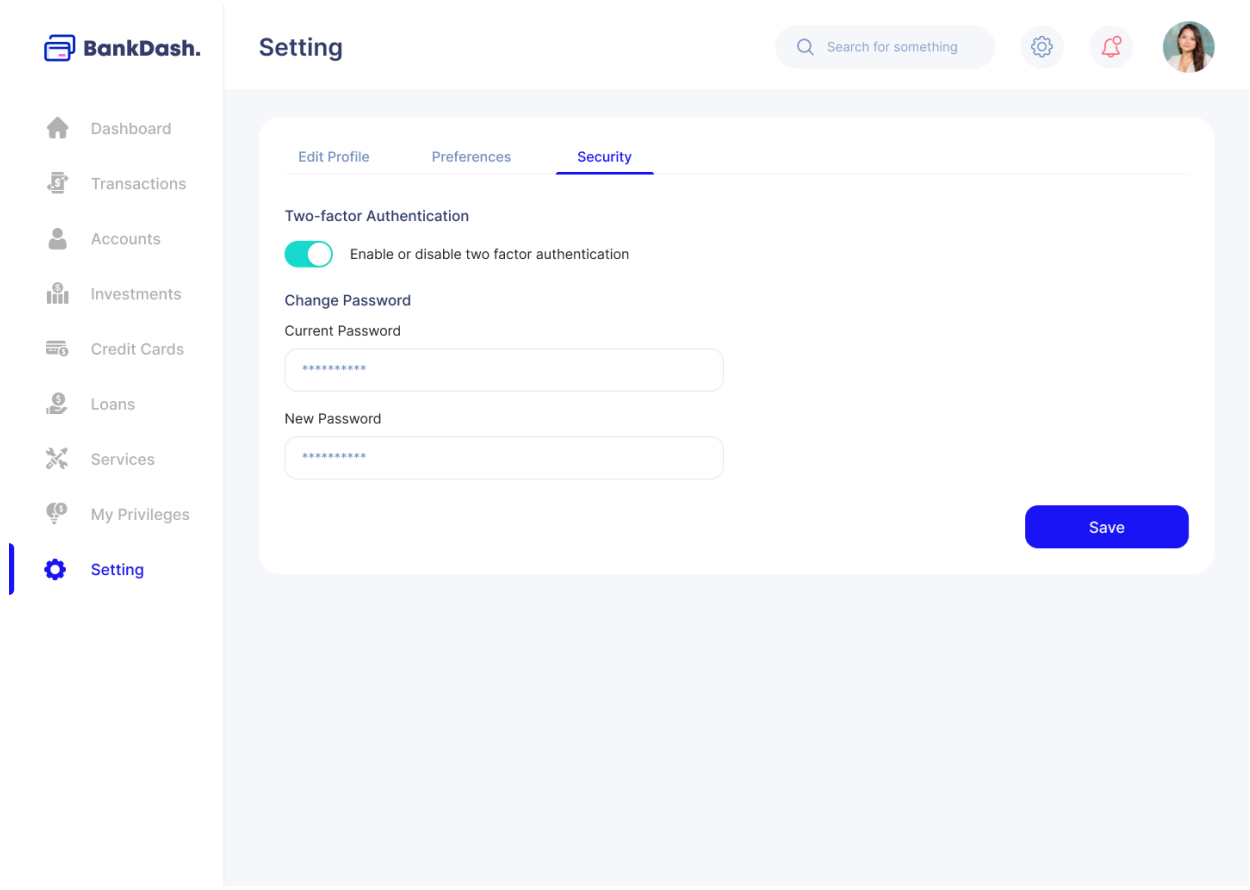
## Setting Page 1




## Setting Page 2



**Setting Page 3**




# 3 Fonctionnalités principales de l'application

BankDash, application bancaire moderne, doit répondre à ces attentes en offrant des fonctionnalités innovantes qui simplifient les opérations financières quotidiennes tout en garantissant une sécurité optimale.

## 3.1 Affichage des comptes bancaires et cartes de crédit

La gestion efficace des finances commence par une vision claire et organisée de ses comptes et cartes. Une application bancaire moderne doit fournir aux utilisateurs une interface intuitive qui centralise ces informations essentielles, facilitant ainsi le suivi et la prise de décisions financières. Voici les principales fonctionnalités liées à l'affichage des comptes et cartes :

### Liste claire et organisée des comptes et cartes associés à l'utilisateur

- Présentation intuitive et hiérarchisée pour une consultation rapide

### Informations détaillées pour chaque compte bancaire

- Numéro de compte (partiellement masqué pour la sécurité).
- Type de compte : épargne, courant, ou autre.
- Solde disponible pour les transactions et solde bloqué (le cas échéant).

### Détails des cartes de crédit ou débit

- Affichage du numéro masqué (**** **** **** 1234) pour une confidentialité renforcée.
- Type de carte : crédit, débit ou prépayée.

- Date d'expiration pour éviter les oublis.
- Statut de la carte : active, bloquée, ou en attente d'activation.

## 3.2 Visualisation des transactions récentes et du solde en temps réel

La gestion de ses finances personnelles nécessite une transparence totale et une réactivité immédiate. Pour répondre à ces attentes, l'application permet aux utilisateurs de suivre en temps réel toutes leurs transactions bancaires, offrant ainsi une visibilité complète sur leurs finances. Chaque mouvement est mis à jour instantanément, permettant une gestion fluide et un contrôle total des soldes. Voici les principales fonctionnalités liées à la visualisation des transactions et du solde en temps réel :

**Accès instantané à l'historique des transactions**

Les utilisateurs peuvent consulter rapidement leurs dernières transactions, triées par ordre chronologique ou selon des filtres personnalisés (par montant, date, bénéficiaire, etc.).

**Mise à jour en temps réel des soldes**

Dès qu'une transaction est effectuée, le solde des comptes est mis à jour immédiatement, garantissant ainsi une vision exacte et instantanée des finances.

**Détails complets des transactions**

Pour chaque opération, les utilisateurs peuvent accéder aux informations détaillées : montant, bénéficiaire ou payeur, date, statut de la transaction (en attente, validé, échoué), ainsi que des notes éventuelles associées.

**Notifications push ou par e-mail**

Des alertes en temps réel sont envoyées pour informer les utilisateurs des transactions importantes ou suspectes, renforçant ainsi la sécurité et la réactivité face à toute activité inhabituelle.

## 3.3 Graphiques de dépenses par catégorie et historique des soldes.

Dans un environnement financier en constante évolution, il est essentiel pour les utilisateurs de pouvoir visualiser l'impact de leurs dépenses et d'analyser leurs habitudes. Grâce aux graphiques interactifs, l'application offre une vue détaillée et compréhensible de la répartition des dépenses par catégorie. Cela permet de mieux gérer son budget, d'identifier les domaines d'optimisation et de suivre l'évolution des économies. Voici les principales fonctionnalités liées aux graphiques de dépenses par catégorie et à l'historique des soldes :

**Diagrammes et graphiques interactifs**

Des graphiques clairs et dynamiques permettant de visualiser les dépenses selon différentes catégories (alimentation, logement, loisirs, transport, etc.), offrant ainsi une vue d'ensemble facile à interpréter.

**Catégories de dépenses personnalisées**

Les utilisateurs peuvent consulter leurs dépenses classées dans des catégories spécifiques, comme les courses alimentaires, les loisirs, les factures de logement, etc., afin de comprendre rapidement où va leur argent.

**Comparaison mensuelle ou hebdomadaire des dépenses**

Possibilité de comparer les dépenses sur différentes périodes (hebdomadaires, mensuelles, etc.), afin d'évaluer les variations dans les habitudes de consommation.

**Affichage des tendances sur le long terme**

Visualisation de l'évolution des soldes au fil du temps, avec des graphiques montrant l'augmentation ou la diminution des économies, permettant ainsi aux utilisateurs d'anticiper et de planifier leur avenir financier.

**Export des graphiques et rapports**

Possibilité d'exporter les graphiques, les rapports de dépenses et l'historique des soldes au format PDF ou CSV, facilitant ainsi le suivi financier et l'archivage des informations importantes.

## 3.4 Transferts rapides d'argent entre utilisateurs

Les transferts d'argent rapides et sécurisés sont essentiels dans une application bancaire moderne. Pour répondre à cette exigence, l'application permet des transferts instantanés, tant entre comptes bancaires qu'entre utilisateurs de la plateforme. Ces transferts sont simples à réaliser, avec une expérience fluide et intuitive, offrant à l'utilisateur la possibilité de transférer de l'argent en quelques clics, tout en garantissant la sécurité des transactions. Voici les principales fonctionnalités liées aux transferts rapides dargent entre utilisateurs :

**Transferts instantanés entre comptes bancaires ou utilisateurs**

L'application permet de réaliser des transferts d'argent instantanés, que ce soit entre comptes bancaires externes ou au sein de la plateforme entre utilisateurs.

**Recherche de bénéficiaires par numéro de compte, adresse e-mail ou identifiant utilisateur**

Pour simplifier le processus, les utilisateurs peuvent rechercher des bénéficiaires par différents critères (numéro de compte bancaire, e-mail ou identifiant unique au sein de la plateforme).

**Confirmation en temps réel des transferts**

Après chaque transfert, l'utilisateur reçoit une confirmation immédiate avec un résumé détaillé de l'opération (montant, bénéficiaire, date, etc.), assurant une transparence totale sur la transaction.

**Option d'ajout de notes ou motifs pour chaque transfert**

Les utilisateurs ont la possibilité d'ajouter des commentaires ou des motifs pour chaque transfert, ce qui peut être utile pour clarifier l'objet de la transaction.

**Enregistrement des bénéficiaires pour des transferts fréquents**

Pour faciliter les transferts futurs, l'application permet d'enregistrer les bénéficiaires fréquemment utilisés, réduisant ainsi les étapes nécessaires lors de prochains transferts.

## 3.5 Authentification sécurisée avec JWT

Dans le cadre de la sécurité des informations bancaires sensibles, une authentification robuste est cruciale. L'application adopte une méthode d'authentification basée sur JSON Web Token (JWT), un standard moderne permettant de garantir l'intégrité des sessions utilisateur et de sécuriser les communications. Cette approche renforce la sécurité tout en offrant une expérience fluide et rapide pour l'utilisateur. Voici les principales fonctionnalités liées à l'authentification sécurisée avec JWT :

**Authentification basée sur JSON Web Token (JWT)**

Chaque utilisateur se connecte de manière sécurisée grâce à un JWT, qui assure une gestion fiable et sécurisée des sessions, permettant de vérifier l'identité de l'utilisateur tout au long de sa navigation.

**Support de l'authentification multi-facteurs (MFA)**

Pour ajouter une couche de sécurité supplémentaire, l'application prend en charge l'authentification multi-facteurs (MFA) via un code temporaire (OTP) envoyé par SMS ou e-mail, ou en utilisant une application d'authentification dédiée (ex. : Google Authenticator).

**Gestion automatique de l'expiration des tokens**

Les tokens JWT ont une durée de vie limitée, ce qui garantit que les sessions ne restent pas ouvertes indéfiniment. Lorsqu'un token expire, l'utilisateur doit se reconnecter, renforçant ainsi la sécurité.

**Protection contre les attaques CSRF (Cross-Site Request Forgery)**

L'application intègre des mécanismes de protection contre les attaques CSRF, empêchant toute tentative malveillante d'effectuer des actions non autorisées au nom de l'utilisateur, en s'assurant que les requêtes sont légitimes et sécurisées.

## 3.6 Gestion des privilèges utilisateur et accès aux paramètres

La gestion des privilèges utilisateur et des paramètres est essentielle pour garantir un contrôle granulaire sur l'accès aux informations et aux fonctionnalités de l'application. En permettant une hiérarchisation des rôles, ainsi qu'une personnalisation des paramètres, l'application assure à chaque utilisateur un niveau d'accès adapté à ses besoins, tout en garantissant la sécurité et la confidentialité des données. De plus, un suivi complet des activités permet de renforcer la transparence et la responsabilité des utilisateurs. Voici les principales fonctionnalités liées à la gestion des privilèges utilisateur et à l'accès aux paramètres :

**Hiérarchisation des rôles et permissions**

L'application offre différents niveaux d'accès, tels que utilisateurs standard, administrateurs, et super-utilisateurs. Chaque rôle bénéficie de permissions adaptées à son niveau de responsabilité, garantissant un contrôle précis des actions possibles au sein de l'application.

**Personnalisation des paramètres de l'application**

Les utilisateurs peuvent personnaliser divers aspects de leur expérience, tels que la langue de l'application, les préférences de notification (notifications push, e-mail, SMS), ainsi que les méthodes d'authentification (par exemple, choix entre mot de passe ou authentification multi-facteurs).

**Gestion des comptes liés**

Les utilisateurs peuvent ajouter ou supprimer des comptes liés à leur profil, comme ceux d'un conjoint, enfant, ou autre membre de la famille, permettant une gestion commune des finances au sein d'un même compte.

**Journal d'activité**

Un journal d'activité permet de suivre toutes les actions effectuées sur le compte, incluant les connexions récentes et les actions réalisées (modifications de paramètres, transferts d'argent, etc.), offrant ainsi un historique transparent et sécuritaire de l'utilisation de l'application.

# 4 Architecture de l'application

## 4.1 Structure des Composants

L'application est constituée de plusieurs composants clés. Voici les composants principaux de l'application, leur rôle et les technologies associées :

#### 4.1.1 Frontend (Angular)

**Rôle** Fournir une interface utilisateur réactive et dynamique.

**Technologie** Angular (un framework JavaScript pour la construction d'applications web à page unique).

**Interaction** Consomme les API REST du backend et fournit une interface riche pour l'utilisateur.

#### 4.1.2 Backend API (Spring Boot)

**Rôle** Fournir des endpoints RESTful pour le frontend. Gère la logique métier, les traitements des données, et les requêtes HTTP.

**Technologie** Spring Boot (un framework Java basé sur Spring pour créer des applications de backend facilement configurables).

**Interaction** Expose des API REST qui sont consommées par le frontend. Interagit avec la base de données pour gérer les données utilisateurs et autres ressources.

#### 4.1.3 Base de Données (MongoDB)

**Rôle** Stocker les informations essentielles comme les utilisateurs, les cartes, les transactions, etc.

**Technologie** MongoDB (une base de données NoSQL orientée documents, idéale pour les applications évolutives avec des données non structurées).

**Interaction** Le backend interroge et met à jour la base de données MongoDB selon les actions de l'utilisateur ou les processus métier.

#### 4.1.4 Authentification et Autorisation (JWT)

**Rôle** Gérer l'authentification des utilisateurs et leur autorisation à accéder à certaines ressources.

**Technologie** JWT (JSON Web Token, utilisé pour sécuriser les échanges entre le frontend et le backend).

**Interaction** Lors de la connexion de l'utilisateur, un token JWT est généré, et ce token est utilisé pour sécuriser les appels API.

#### 4.1.5 CI/CD (Jenkins, Docker)

**Rôle** Automatiser les processus de développement, tests, et déploiement.

**Technologie** Jenkins (outil d'intégration continue et de déploiement continu), Docker (pour conteneuriser les applications et faciliter leur déploiement).

**Interaction** Le code est automatiquement testé et déployé à chaque mise à jour via Jenkins, et Docker garantit que l'application fonctionne dans un environnement isolé et cohérent.

#### 4.1.6 Déploiement Cloud (AWS)

**Rôle** Héberger l'application et la base de données dans le cloud.

**Technologie** Amazon Web Services (AWS) fournit une infrastructure cloud pour déployer des applications évolutives.

**Interaction** L'application backend et la base de données MongoDB sont déployées sur AWS, offrant ainsi une scalabilité, une sécurité et une disponibilité accrues.

## 4.2 Schéma de l'architecture

Le schéma suivant illustre la manière dont les différents composants interagissent :

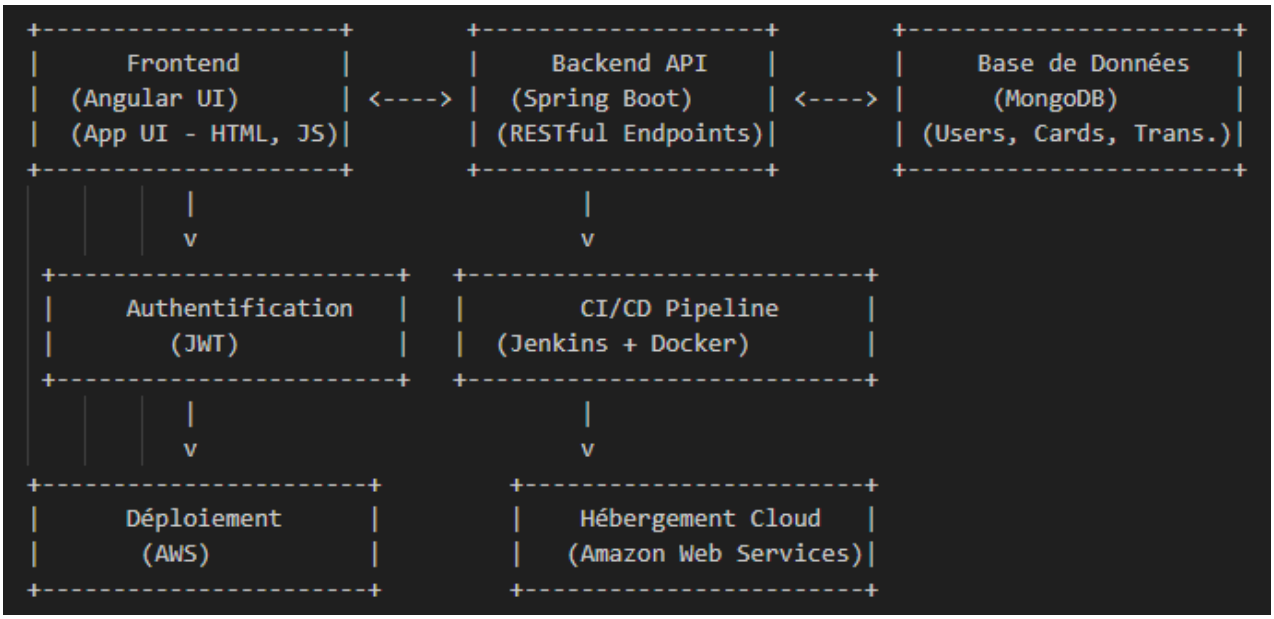



FIGURE 1 – Composants et interactions de l'application

L'architecture est basée sur une architecture client-serveur avec une séparation entre les composants :

1) **Client (Angular)** interface utilisateur qui consomme les API REST fournies par le backend.
2) **Serveur (Spring Boot)** gestion des requêtes, traitement des données, application de la logique métier.
3) **Base de Données (MongoDB)** stockage des données utilisateur, informations des cartes, historiques de transactions, etc.
4) **CI/CD (Jenkins et Docker)** pour le développement, le test et le déploiement continu.
5) **Hébergement (AWS)** déploiement du serveur et de la base de données.

```
BankDash/
├── Backend-Spring/          // Backend Spring Boot + MongoDB
├── Frontend-Angular/        // Frontend Angular
├── Docker/                  // Dockerfiles et Compose
├── Jenkins/                 // Scripts pour CI/CD avec Jenkins
├── AWS/                     // Scripts pour déploiement sur AWS
└── README.md                // Documentation
```

FIGURE 2 – Structure des dossiers de l'application

# 5 Technologies et outils requis

Pour assurer le bon développement, la configuration, et le déploiement de l'application, les outils et technologies suivants sont nécessaires :

• **Java 17** Langage de programmation requis pour exécuter le framework Spring Boot.

- **Maven** Outil de gestion de dépendances et d'automatisation des builds requis pour les projets Spring Boot.
- **Spring Boot** Framework permettant de créer des applications backend robustes, sécurisées, et évolutives.
- **Node.js** Environnement d'exécution JavaScript requis pour construire des projets Angular.
- **Angular CLI** Outil en ligne de commande pour générer et gérer des applications Angular efficacement.
- **MongoDB** Base de données NoSQL utilisée pour stocker les données de manière flexible.
- **Git** Système de contrôle de version distribué pour gérer efficacement le code source et suivre les modifications.
- **GitHub** Plateforme d'hébergement de référentiels Git pour collaborer, partager et versionner le code source.
- **VSCode** Éditeur de code léger et extensible adapté au développement front-end et back-end.
- **IntelliJ IDEA** Environnement de développement intégré (IDE) performant et complet pour travailler efficacement sur les projets Spring Boot et Java.
- **Docker** Outil permettant de conteneuriser l'application pour simplifier la gestion des environnements et le déploiement.
- **Jenkins** Serveur d'intégration continue utilisé pour automatiser les processus de build, test, et déploiement.
- **AWS (Amazon Web Services)**vPlateforme cloud utilisée pour déployer et héberger l'application, avec des options comme EC2, S3, Elastic Beanstalk ...

# 6 Modèle de Données et Schéma de la Base de Données

Le modèle de données pour cette application repose sur plusieurs collections MongoDB interconnectées, chacune représentant une entité ou un groupe d'entités spécifiques. Chaque collection est conçue pour stocker des informations pertinentes et permettre des requêtes efficaces. Ces collections sont définies avec des schémas flexibles qui facilitent la gestion des données complexes tout en optimisant les performances de l'application.

## 6.1 Entités principales de l'application

### 6.1.1 User

Cette entité représente l'utilisateur de l'application et contient les informations personnelles et de sécurité nécessaires pour l'authentification et l'autorisation. L'entité User contient les attributs suivants :

- **id** Identifiant unique de l'utilisateur.
- **username** Nom d'utilisateur pour la connexion.
- **password** Mot de passe sécurisé (hashé).
- **email** Adresse e-mail de l'utilisateur.
- **role** Rôle de l'utilisateur (e.g., "USER", "ADMIN").
- **profilePicture** URL de la photo de profil de l'utilisateur.
- **createdAt** Date de création du compte.

### 6.1.2 Card

Cette entité représente les cartes bancaires de l'utilisateur affichées dans la section "My Cards". L'entité Card contient les attributs suivants :

- **id** Identifiant unique de la carte.

- **cardNumber** Numéro de la carte (affiché partiellement pour la sécurité).
- **cardHolder** Nom du titulaire de la carte (affiché comme "CARD HOLDER").
- **balance** Solde actuel de la carte.
- **validThru** Date d'expiration de la carte.
- **type** Type de carte (débit, crédit, etc.).
- **createdAt** Date d'ajout de la carte.
- **userId** Référence à l'utilisateur propriétaire de la carte.

### 6.1.3 Transaction

Cette entité représente les transactions financières de l'utilisateur. Les transactions récentes sont affichées dans la section "Recent Transaction". L'entité Transaction contient les attributs suivants :

- **id** Identifiant unique de la transaction.
- **transactionType** Type de transaction (e.g., "Deposit", "Withdraw").
- **amount** Montant de la transaction.
- **date** Date de la transaction.
- **description** Brève description de la transaction (ex. "Deposit from my Card").
- **source** Source de la transaction (ex. "Paypal").
- **userId** Référence à l'utilisateur qui a effectué ou reçu la transaction.
- **cardId** Référence à la carte associée (si applicable).

### 6.1.4 Statistic

Cette entité représente les statistiques de dépenses par catégorie pour l'utilisateur, visibles dans le graphique circulaire "Expense Statistics". L'entité Statistic contient les attributs suivants :

- **id** Identifiant unique de la statistique.
- **category** Catégorie de dépense (e.g., "Entertainment", "Bill Expense", "Investment", "Others").
- **percentage** Pourcentage de chaque catégorie de dépense par rapport aux dépenses totales.
- **amount** Montant total des dépenses dans cette catégorie pour une période donnée (facultatif).
- **userId** Référence à l'utilisateur associé.

### 6.1.5 Transfer

Cette entité représente les transferts d'argent entre utilisateurs, visible dans la section "Quick Transfer". L'entité Transfer contient les attributs suivants :

- **id** Identifiant unique du transfert.
- **amount** Montant transféré.
- **date** Date du transfert.
- **status** Statut du transfert (e.g., "Completed", "Pending").
- **message** Message optionnel pour le transfert.
- **receiverId** Référence à l'utilisateur qui reçoit l'argent.
- **senderId** Référence à l'utilisateur qui envoie l'argent.

### 6.1.6 BalanceHistory

Cette entité représente l'historique du solde de l'utilisateur, utilisé pour générer le graphique de l'évolution du solde ("Balance History"). L'entité BalanceHistory contient les attributs suivants :

- **id** Identifiant unique de l'historique de solde.
- **date** Date du solde enregistré.
- **balance** Solde à cette date.

- **createdAt** Date de création de l'enregistrement.
- **userId** Référence à l'utilisateur associé.

#### 6.1.7 Notification

Bien que non visible dans la maquette, une entité Notification peut être ajoutée pour gérer les notifications de transaction ou d'autres événements importants. L'entité Notification contient les attributs suivants :

- **id** Identifiant unique de la notification.
- **message** Message de la notification.
- **type** Type de notification (e.g., "Transaction", "Alert").
- **date** Date de la notification.
- **readStatus** Indique si la notification a été lue.
- **userId** Référence à l'utilisateur concerné.

### 6.2 Relations entre les entités

- **User** a plusieurs Card, Transaction, Transfer (comme expéditeur et destinataire), Statistic, BalanceHistory, et Notification.
- **Card** appartient à un User et peut être associée à plusieurs Transaction.
- **Transaction** est associée à un User et à une Card.
- **Transfer** est associé à un User en tant qu'expéditeur et à un autre User en tant que destinataire.
- **Statistic** est associé à un User avec des informations sur les dépenses.
- **BalanceHistory** est associé à un User pour suivre les variations de son solde.
- **Notification** est associe à un User

## 7 Création du Backend avec Spring Boot et MongoDB

### 7.1 Initialiser le projet Spring Boot

#### 7.1.1 Accéder au site Spring Initializr

J'ai accédé sur le site Spring Initializr et j'ai rempli les informations suivantes :

**Project** Maven
**Language** Java
**Spring Boot Version** 3.3.5
**Group** com.bankdash
**Artifact** bankdash
**Name** BankDash
**Description** Application de gestion bancaire avec un tableau de bord
**Packaging** Jar
**Java Version** 17

#### 7.1.2 Ajouter les dépendances nécessaires

J'ai cliqué sur "Add Dependencies" et j'ai ajouté les dépendances suivantes :

- **Spring Web** pour la création de l'API REST.
- **Spring Data MongoDB** pour l'intégration avec MongoDB.
- **Spring Security** pour la sécurité et l'authentification.
- **Lombok** pour générer automatiquement les getters, setters, etc

### 7.1.3 Générer le projet

J'ai cliqué sur "Generate" pour télécharger le projet. Une fois généré, j'ai décompressé-le et le copié dans le dossier du projet de l'application BankDash. Enfin, j'ai ajouté, commité et poussé les fichiers sur GitHub.

### 7.1.4 Configurer MongoDB

Après avoir installer et exécuter MongoDB, j'ai ajouté la configuration de connexion à MongoD dans src/main/resources/application.properties :

```
spring.data.mongodb.uri=mongodb://localhost:27017/bankdash
```

### 7.1.5 Organisation des packages (Spring Boot + MongoDB)

J'ai organisé mon application en suivant une structure claire et modulaire pour faciliter sa gestion et sa maintenance. J'ai créé différents packages pour chaque fonction : les contrôleurs REST dans controller, les services métier dans service, les entités JPA/MongoDB dans entity, ainsi que la gestion des exceptions, la sécurité JWT et la configuration du projet dans leurs packages respectifs. Voici un schéma illustrant cette organisation des packages.

```
src/main/java/com/bankdash
├── controller        // Contient les classes REST Controllers
├── dto               // Contient les Data Transfer Objects (DTOs)
├── entity            // Contient les entités JPA/MongoDB
├── mapper            // Contient les classes de mappage entre entités et DTOs
├── exception         // Contient les classes de gestion des exceptions
├── repository        // Contient les interfaces de Repository
├── security          // Contient la configuration et les classes de sécurité JWT
├── service           // Contient les services métier
├── config            // Contient les fichiers de configuration (CORS, MongoDB, etc.)
└── BankDashApplication.java // Classe principale
```

FIGURE 3 – Organisation des packages (Spring Boot + MongoDB)

## 7.2 Créer les entités, les DTO et les Mappers

Pour créer des classes Java pour les entités comme User, Transaction, Card, Statistic, etc., j'ai utilisé Lombok qui permet de simplifier la gestion des getters, setters, constructeurs, etc, j'ai utilisé également les annotations Spring Data MongoDB pour les intégrer dans une base de données MongoDB. Pour chaque entité, j'ai créé un DTO et un mapper associés.
Exemple d'une entité User, de son DTO associé et du mapper pour effectuer la conversion entre les deux :

### 7.2.1 User

L'entité User représente un utilisateur dans le système avec ses informations de connexion.

```java
@Document(collection = "users")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class User {
    @Id
    private String id;

    @Indexed(unique = true)
    private String username;

    @Indexed(unique = true)
    private String email;

    private String password;
    private String role;
    private String profilePicture;
    private LocalDateTime createdAt;
}
```

### 7.2.2 UserDTO

La classe UserDTO représente un objet de transfert de données pour un utilisateur, exposant uniquement les informations essentielles, tout en excluant les données sensibles comme le mot de passe et les champs générés automatiquement comme id ou createdAt.

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class UserDTO {
    private String id;
    private String username;
    private String email;
    private String role;
    private String profilePicture;
    private LocalDateTime createdAt;
}
```

### 7.2.3 UserMapper

L'interface UserMapper est un mapper MapStruct pour convertir entre User et UserDTO

```java
@Mapper
public interface UserMapper {
    UserMapper INSTANCE = Mappers.getMapper(UserMapper.class);
    UserDTO toUserDTO(User user);
    User toUserEntity(UserDTO userDTO);
}
```

## 7.3 Créer les interfaces Repository

Pour chaque entité et à l'aide de Spring Data MongoDB, j'ai créé, dans le package repository, les interfaces (Repositories) qui permettent d'interagir avec les données MongoDB.

```java
public interface UserRepository extends MongoRepository<User, String> {

}
```

## 7.4 Implémenter les Service

Pour chaque entité, j'ai implémenté, dans le package service, les classes de service qui contiennent la logique métier nécessaire pour manipuler les données et orchestrer les différentes opérations de l'application (CRUD).

```java
@Service
@RequiredArgsConstructor
public class UserService {

    private final BCryptPasswordEncoder encoder = new BCryptPasswordEncoder( strength: 12);
    private final UserRepository userRepository;
    private int currentId;

    public List<UserDTO> getAllUsers() {
        return userRepository.findAll() List<User>
                .stream() Stream<User>
                .map(UserMapper.INSTANCE::toUserDTO) Stream<UserDTO>
                .collect(Collectors.toList());
    }

    public UserDTO getUserById(String id) {
        User user = userRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("User not found"));
        return UserMapper.INSTANCE.toUserDTO(user);
    }

    public UserDTO createUser(UserCreateDTO userCreateDTO) {
        User lastUser = userRepository.findTopByOrderByIdDesc();
        int newId = (lastUser == null) ? 1 : Integer.parseInt(lastUser.getId()) + 1;
        userCreateDTO.setId(String.valueOf(newId));
        String hashedPassword = encoder.encode(userCreateDTO.getPassword());

        User user = new User();
        user.setId(String.valueOf(newId));
        user.setUsername(userCreateDTO.getUsername());
        user.setPassword(hashedPassword);
        user.setEmail(userCreateDTO.getEmail());
        user.setRole(userCreateDTO.getRole());
        user.setProfilePicture(userCreateDTO.getProfilePicture());
        user.setCreatedAt(LocalDateTime.now());

        user = userRepository.save(user);
        return UserMapper.INSTANCE.toUserDTO(user);
    }
```

## 7.5 Développer les contrôleurs REST

J'ai développé, dans le package controller, les classes de contrôleur REST qui exposent les API nécessaires pour interagir avec les données de l'application. Ces contrôleurs gèrent les requêtes HTTP et orchestrent les appels aux services pour effectuer les opérations CRUD, tout en assurant la gestion des réponses et des exceptions.

```java
@RestController
@RequestMapping("/users")
@RequiredArgsConstructor
public class UserController {

    private final UserService userService;
    private final AuthenticationManager authenticationManager;
    private final JwtService jwtService;

    @GetMapping
    public List<UserDTO> getAllUsers() { return userService.getAllUsers(); }

    @GetMapping("/{id}")
    public UserDTO getUserById(@PathVariable String id) { return userService.getUserById(id); }

    @PostMapping
    public UserDTO createUser(@RequestBody UserCreateDTO userCreateDTO) {
        return userService.createUser(userCreateDTO);
    }
```

# 8 Implémentation de l'authentification JWT

Pour sécuriser l'application BankDash avec JWT (JSON Web Token), j'ai suivi plusieurs étapes. L'objectif est de permettre une authentification sécurisée basée sur des tokens JWT pour valider les utilisateurs sans nécessiter de sessions persistantes.

## 8.1 Ajouter les dépendances nécessaires

J'ai commencé par ajouter les dépendances nécessaires pour configurer Spring Security, JWT et la gestion des utilisateurs dans le fichier pom.xml :

```xml
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.12.5</version>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.12.5</version>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.12.5</version>
</dependency>
```

## 8.2 Créer le service JWT : JwtService

Le service JWT est un composant essentiel qui sera utilisé par le filtre, le contrôleur d'authentification et d'autres parties du système pour toutes les opérations liées aux jetons. Il doit être disponible avant de commencer à implémenter les interactions avec les tokens. Cette classe fournit des méthodes pour générer, valider, et extraire des informations des tokens JWT.

### 8.2.1 Gestion sécurisée de la clé secrète

Pour éviter de coder la clé secrète directement dans le code, j'ai déplacé la clé dans le fichier application.yml

```
jwt.secret-key=lTgd7rzjWzR9ozfpPguqhvPmrRUclT2dXh8N0lMEXIc=
```

Puis, j'ai injecté cette clé dans JwtService à l'aide de l'annotation @Value

```
@Value("${jwt.secret-key}")
private String secretKey;

private SecretKey getSignKey() {
    byte[] keyBytes = Decoders.BASE64.decode(secretKey);
    return Keys.hmacShaKeyFor(keyBytes);
}
```

Pour la production, j'ai utilisé un gestionnaire de secrets pour récupérer cette clé.

### 8.2.2 Génération des tokens

j'ai généré un token JWT signé, contenant les informations sur l'utilisateur et une expiration de 30 minutes :

- créer une méthode generateToken qui génère un token JWT pour un utilisateur.
- préparer des claims et passé le nom d'utilisateur à la méthode createToken.
- construire le token avec Jwts.builder().
- ajouté les claims, défini le sujet comme userName, et configuré une expiration de 30 minutes.
- signer le token avec une clé secrète via getSignKey() et retourné le token compacté.

```
public String generateToken(String userName) {
    Map<String, Object> claims = new HashMap<>();
    return createToken(claims, userName);
}

private String createToken(Map<String, Object> claims, String userName) {
    return Jwts.builder() JwtBuilder
            .claims() BuilderClaims
            .add(claims)
            .subject(userName)
            .issuedAt(new Date(System.currentTimeMillis()))
            .expiration(new Date(System.currentTimeMillis() + 1000 * 60 * 30))
            .and() JwtBuilder
            .signWith(getSignKey())
            .compact();
}
```

### 8.2.3 Extraction des données

J'ai ajouté des méthodes pour extraire le nom d'utilisateur et d'autres informations contenues dans le jeton (token), via des Claims.

```
public String extractUsername(String token) { return extractClaim(token, Claims::getSubject); }

private <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
    final Claims claims = extractAllClaims(token);
    return claimsResolver.apply(claims);
}

private Claims extractAllClaims(String token) {
    return Jwts.parser() JwtParserBuilder
            .verifyWith(getSignKey())
            .build() JwtParser
            .parseSignedClaims(token) Jws<Claims>
            .getPayload();
}
```

### 8.2.4 Validation des tokens

Validation des jetons : J'ai implémenté une méthode pour vérifier si un jeton est valide (signature et date d'expiration). La méthode validateToken vérifie :

- Si le jeton est signé correctement.
- Si le jeton n'est pas expiré.
- Si le nom d'utilisateur correspond bien à celui de l'utilisateur authentifié.

```
private Boolean isTokenExpired(String token) { return extractExpiration(token).before(new Date()); }

private Date extractExpiration(String token) { return extractClaim(token, Claims::getExpiration); }

public Boolean validateToken(String token, UserDetails userDetails) {
    final String username = extractUsername(token);
    return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
}
```

## 8.3 Créer les Services Utilisateur

## 8.4 Créer le filtre JWT : JwtAuthenticationFilter

## 8.5 Configurer Spring Security : SecurityConfiguration

## 8.6 Implémenter le contrôleur d'authentification

# 9 Développement du Frontend avec Angular

Pour la mise en place du Frontend Angular de l'application, j'ai effectué les étapes suivantes :

## 9.1 Initialisation du projet Angular

### 9.1.1 Créer le projet

Pour créer le projet, j'ai utilisé Angular CLI avec la commande :

```
ng new Frontend-Angular
```

Lors de l'exécution de cette commande, Angular CLI m'a demandé plusieurs options :

- Which stylesheet format would you like to use ? : J'ai choisi CSS pour la simplicité, mais d'autres options comme SCSS, SASS, ou LESS sont également disponibles.
- Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Pre-rendering) ? (y/N) : J'ai choisi "N" (Non), mon application n'a pas besoin de SSR ou SSG, car les interactions se passent principalement côté client et il n'y a pas de besoin particulier d'optimiser l'affichage initial ou d'améliorer le SEO. authentifié.

Angular CLI a alors créé une structure de projet complète dans le dossier "Frontend-Angular". Cette structure comprend des fichiers et des dossiers essentiels comme src/, qui contient le sous-dossier app/ où se trouvent les composants principaux de l'application.

### 9.1.2 Naviguer vers dossier "Frontend-Angular"

```
cd Frontend-Angular
```

### 9.1.3 Installer les dépendances

Ensuite, j'ai installé @angular/material pour les composants visuels, rxjs pour les flux de données, et @auth0/angular-jwt pour gérer le JWT.

```
ng add @angular/material
npm install @auth0/angular-jwt rxjs
```

### 9.1.4 Lancer le serveur Angular

Pour lancer l'application en mode développement, j'ai exécuté la commande suivante :

```
ng serve
```

Cela démarre un serveur de développement à l'adresse http ://localhost :4200

## 9.2 Assurer une communication sécurisée entre Angular et Spring

Après avoir initialisé la partie frontend de l'application BankDash, il est nécessaire d'apporter certaines modifications du côté backend avec Spring Boot afin de garantir une intégration fluide avec le frontend Angular. Voici les étapes importantes à suivre pour assurer cette connexion.

### 9.2.1 Ajouter la configuration CORS (Cross-Origin Resource Sharing)

Lorsque le frontend Angular interagit avec le backend Spring Boot, il peut se retrouver sur un domaine ou un port différent de celui du backend. Par défaut, les navigateurs bloquent les requêtes entre différentes origines (cross-origin requests). Pour contourner ce blocage, il est essentiel de configurer CORS dans le backend. Cela permet au serveur d'accepter les requêtes provenant de différentes origines, telles que celle du frontend Angular.

```java
@Configuration
public class CorsWebConfiguration implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping( pathPattern: "/**")
                .allowedOrigins("http://localhost:4200")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowedHeaders("*")
                .allowCredentials(true);
    }
}
```

Détails de la configuration CORS

- Annotation @Configuration indique une classe de configuration Spring.
- Implémentation de WebMvcConfigurer permet de personnaliser les règles MVC.
- addCorsMapping configure les règles CORS.
- registry.addMapping("/**") autorise tous les chemins.
- allowedOrigins("http ://localhost :4200") accepte les requêtes de cette origine.
- allowedMethods("GET", "POST", "PUT", "DELETE") limite les méthodes autorisées.
- allowedHeaders("*") autorise tous les en-têtes.
- allowCredentials(true) permet l'envoi de cookies ou identifiants.

### 9.2.2 Application de la configuration CORS dans SecurityConfiguration

Dans la classe SecurityConfiguration, j'ai ajouté la ligne **.cors(Customizer.withDefaults())** pour activer la gestion CORS (Cross-Origin Resource Sharing) en utilisant la configuration définie dans la classe CorsWebConfiguration. Cela indique à Spring Security d'appliquer automatiquement les règles CORS définies ailleurs dans l'application (comme les origines autorisées, les méthodes HTTP, etc.) sans avoir besoin de les redéfinir dans la configuration de sécurité.

### 9.2.3 Conversion du token JWT brut en token JWT au format JSON

Lorsqu'un utilisateur se connecte, le backend génère un token JWT brut (chaîne de caractères) après avoir validé les informations d'identification de l'utilisateur. Cependant, pour que le frontend Angular puisse l'utiliser correctement, il est nécessaire de renvoyer ce token dans un format structuré sous forme d'objet JSON. Cela permet au frontend de récupérer facilement le token pour l'utiliser dans les requêtes suivantes.

```java
return ResponseEntity.ok(Collections.singletonMap("token", jwtService.generateToken(authRequest.getUsername())));
```

Cette ligne génère un token JWT pour l'utilisateur et le renvoie dans une réponse HTTP avec le statut 200 (OK), sous forme de JSON contenant la clé "token" et sa valeur (le token).
En résumé, ces étapes sont essentielles pour assurer la communication sécurisée et fluide entre le frontend Angular et le backend Spring Boot de l'application BankDash. La gestion des CORS garantit que les requêtes inter-domaines sont autorisées, tandis que la conversion du token JWT permet au frontend de traiter l'authentification de manière transparente.

## 9.3 Structure des Composants

Cette étape consiste à transformer la maquette conçue sur Figma en une interface utilisateur fonctionnelle en Angular. Il s'agit de traduire les éléments visuels, les interactions et les

comportements définis dans la maquette en composants, directives et services Angular, en respectant les bonnes pratiques de développement. Cela inclut la création de la structure de l'application, l'intégration des composants graphiques, la gestion des états, ainsi que l'application des styles CSS. De plus, le projet Angular sera directement lié au backend Spring Boot déjà développé, utilisant JWT pour l'authentification et la gestion sécurisée des sessions utilisateurs, afin de permettre une communication fluide entre l'interface front-end et les services back-end.

### 9.3.1 Création des Composants

Pour chaque page de la maquette je vais créer des composants Angular qui implémentent l'interface visuelle et les comportements définis dans la maquette.

**Composant Login**

```
ng generate component components/login
```

Cela génère un composant LoginComponent dans le dossier components/login/ avec 4 fichiers :

- `login.component.ts`
- `login.component.html`
- `login.component.css`
- `login.component.spec.ts`

**Composant Dashboard**

```
ng generate component components/dashboard
```

Cela génère un composant DashboardComponent dans components/dashboard/ avec 4 fichiers :

- `dashboard.component.ts`
- `dashboard.component.html`
- `dashboard.component.css`
- `dashboard.component.spec.ts`

## 9.4 Configurer les services Angular

Dans le cadre du développement d'une application Angular, les services jouent un rôle crucial en permettant de gérer la logique métier et les interactions avec des ressources externes (API, stockage local, etc.). La configuration des services Angular est une étape importante pour assurer la bonne communication entre le front-end (l'interface utilisateur) et le back-end (serveur ou API) et pour organiser les différentes fonctionnalités de l'application.

**Service Auth**

```
ng generate service services/auth/auth
```

Cela génère un service AuthService dans services/auth/