POLYTECH NICE SOPHIA | UNIVERSITÉ CÔTE D'AZUR

Rapport Final ISA

Equipe M

Farineau Thomas

Kitabdjian Léo

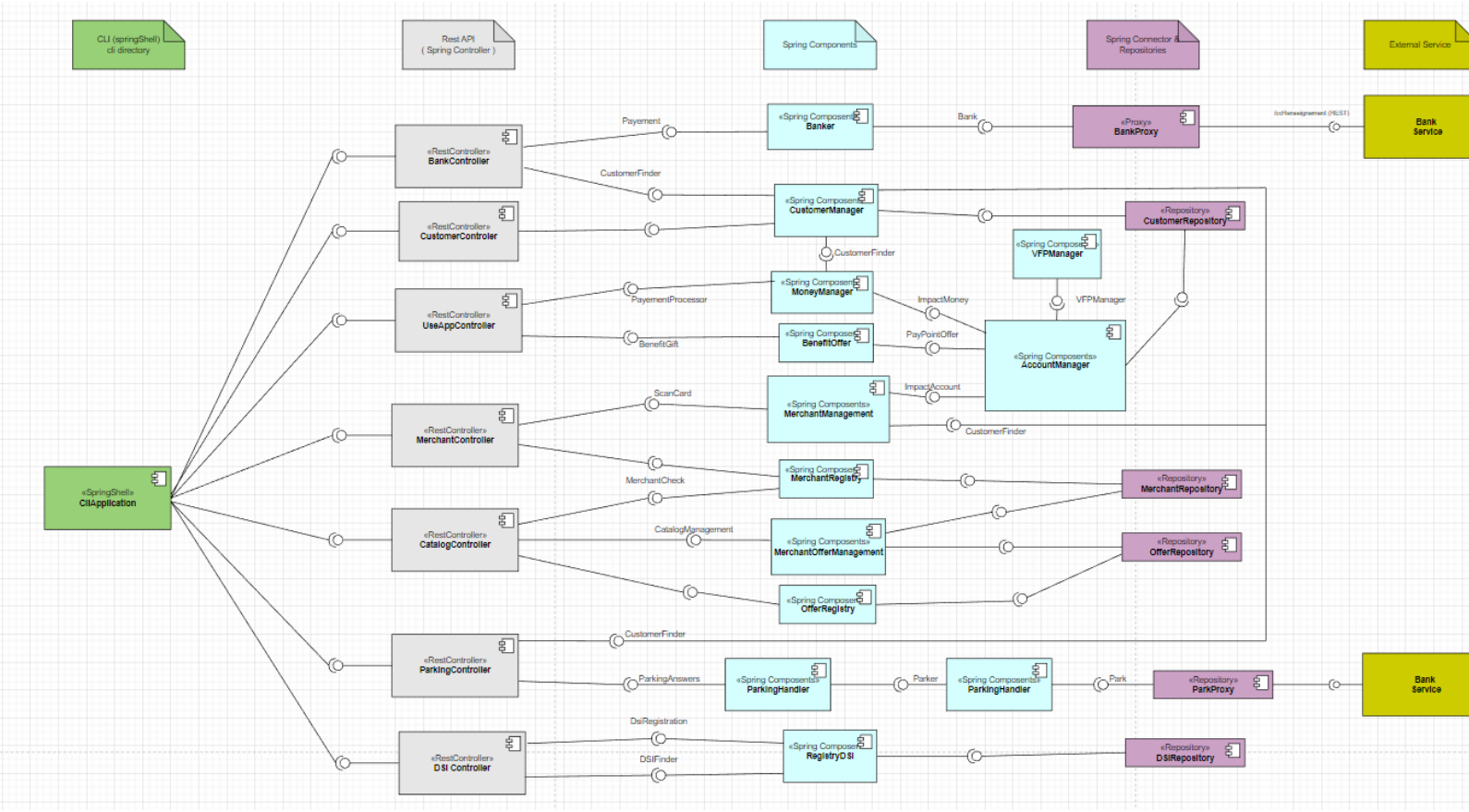
Chelgham Zinedine

Bassy Ayman

ELFIGUIGUI Aymane

SOMMAIRE

Diagramme de Composants et Interfaces



Nous présentons dans ce document l'architecture logicielle Spring actuelle de notre système, qui a connu plusieurs modifications depuis sa première version. Dans la version initiale, par exemple, les marchands déjà inscrits étaient considérés comme des clients, alors qu'ils sont maintenant considérés comme des marchands partenaires et peuvent s'inscrire directement via le composant "MarchantRegistry". Les composants liés à la DSI ont été abandonnés, car les fonctionnalités n'ont pas pu être implémentées avant la fin du temps imparti.

Malgré les efforts fournis, notre équipe a rencontré d'importantes difficultés lors de la mise en place de l'architecture initiale, notamment en raison d'un manque de communication - en partie lié à la grève - et d'un travail en parallèle des membres de l'équipe. Cela a entraîné un certain retard dans le développement des fonctionnalités, ce qui fait que nous avons priorisé les fonctionnalités nécessaires au développement d'un MVP viable en ayant le plus de valeur possible en passant par les deux services externes.

Notre diagramme de composants actuel représente une architecture plus simple, avec moins de fonctionnalités que la version initiale, mais elle est mieux construite et plus adaptée à nos besoins.

Justification de découpage Composants/Interfaces

Banker : Pour l'interaction avec la banque, nous avons besoin d'un composant qui implémente la logique de gain d'argent en cas de recharge de la carte multi-fidélité. Ce composant est également lié au proxy de liaison avec le service externe de la banque pour savoir si le paiement a été validé pour le montant donné.
Interface : Payment, elle permet la recharge d'un compte client via l'utilisation de sa carte bancaire.

ParkingHandler : Ce composant a la responsabilité de capter les potentielles erreurs identifiées par le composant proxy du dessous (ParkingGuard) en attribut et de renvoyer au client la réponse à sa requête, que cela soit un succès ou un message d'erreur clair.
Interface : ParkingAnswerer, contient les opérations de renvoi de message selon la requête du client

ParkingGuard : Composant possédant le proxy du parking en attribut qui va s'occuper de lancer l'appel et de l'interpréter en erreur ou succès pour le composant du dessus (ParkingHandler)
Interface : Parker, contient les opérations pour communiquer avec le service externe, à savoir entamer un stationnement et récupérer le temps de stationnement restant.

CustomerManager : Afin de pouvoir enregistrer des clients et récupérer des informations relatives à un client dans le système.
Interface : CustomerRegistration, pour enregistrer un client, et CustomerFinder, dans le but de pouvoir récupérer un client selon un critère depuis la base de données, dans l'objectif de vérification et non de modification.

MoneyManager : Dans le cas où un client souhaite payer avec sa carte multi-fidélité, il faut que cela soit possible. MoneyManager implémente la logique de paiement de transaction émise en magasin impliquant le système. Ces paiements peuvent être effectués grâce à la carte multi-fidélité, préalablement rechargée. Ou dans le cadre d'une amélioration du projet, ils peuvent être effectués directement en ligne.
Interface : PaymentProcessor est l'interface qui implémente l'action de payer avec sa carte de fidélité directement en magasin.

BenefitOffer : L'objectif principal de l'offre de multi-fidélité est de permettre au client de dépenser les points acquis dans des offres émises par les marchands ou les membres de la DSI. BenefitOffer implémente cette logique et permet au client, via la dépense de ces points, de pouvoir percevoir une offre, pouvant être un cadeau en magasin ou un avantage ultérieur.

Interface : BenefitGift implémente l'achat d'un cadeau en magasin par un client. Après l'achat, le nombre de points du client est déduit et il reçoit un code associé à l'offre qui lui permettra de récupérer son cadeau après scan chez un marchand.

VFPManager : Après un certain nombre d'achats, espacés par une limite de temps, le client peut obtenir un statut supérieur qui lui permet d'accéder à des offres plus avantageuses.
Interface : VFPManager est l'interface qui implémente la vérification d'un achat effectué par un client, lui permettant potentiellement de recevoir le statut de VFP ou de le perdre.

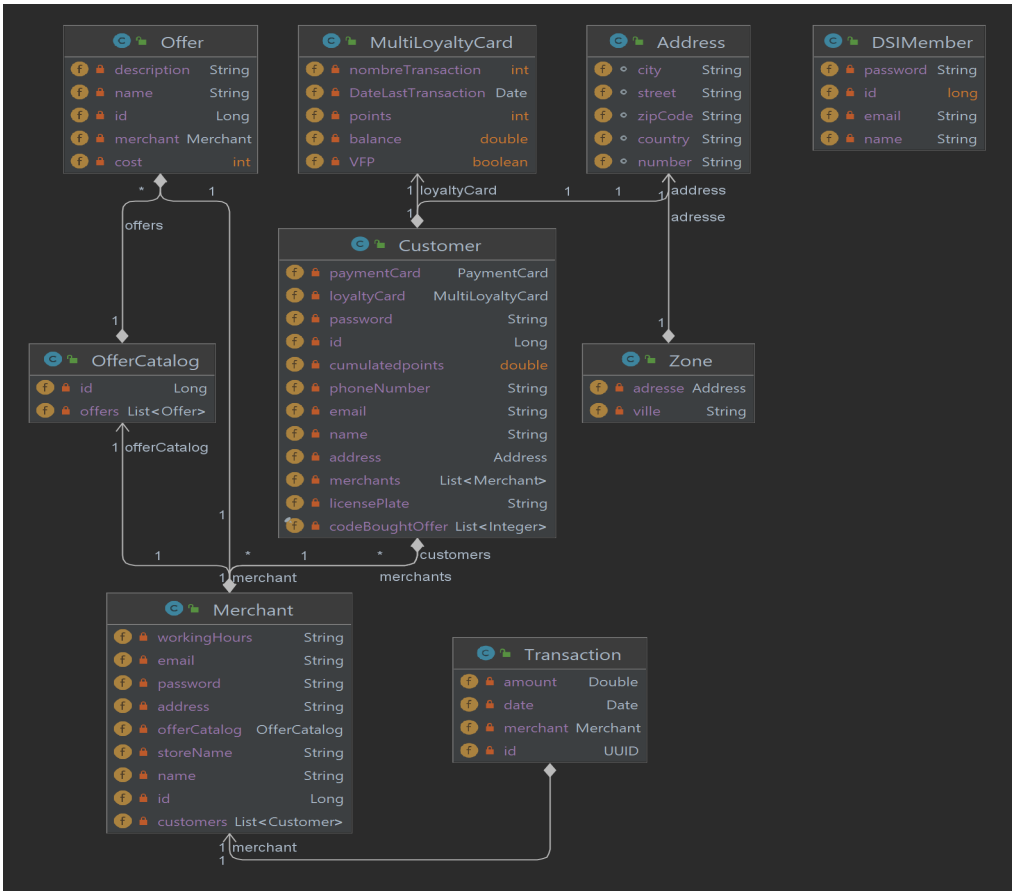
MerchantManagement : Le marchand joue un rôle important dans l'application. Il permet au client de recevoir des points, mais aussi des offres. En effet, le client peut scanner des codes de transaction pour gagner des points de fidélité, ou des codes d'offre pour confirmer la récupération en magasin de l'offre.
Interface : ScanCard est l'interface représentative de la responsabilité métier du marchand pour le scan de transaction ou de code d'offre. Cette démarche s'inscrit dans le cadre de la responsabilité du marchand. Dans une future implémentation, l'inscription du marchand sera soumise à des vérifications, via la responsabilité de la DSI, par exemple.

AccountManager : En interne, les différentes actions effectuées ont un impact direct sur l'état du compte du client. En effet, en fonction du client impliqué, il faudra retenir les changements engendrés par celles-ci, via la persistance dans le cas présent. AccountManager est directement lié au CustomerRepository et est chargé d'implémenter directement sur le compte client les modifications qui auront été effectuées.
Interface : ImpactMoney implémente le métier lié au paiement avec une carte de fidélité, vérifie si le paiement est possible, retire le montant payé au client tout en vérifiant la possibilité de devenir VFP via VFPManager, et lui faisant gagner des points de fidélité. PayPointOffer fait payer le client un montant de point et lui génère un code d'offre à présenter au marchand. ImpactAccount permet de faire gagner des points au client grâce au montant de la transaction.

MerchantRegistry : Les marchands doivent pouvoir s'enregistrer dans le système et ce dernier peut être modifié sans affecter ce qui est déjà implémenté.
Interface : MerchantRegistry pour enregistrer les marchands et MerchantCheck pour vérifier l'existence d'un marchand.

MerchantOfferManager : Les marchands doivent pouvoir accéder à leur catalogue d'offres et le modifier.
Interface : CatalogManagement implémente un ensemble d'actions CRUD liées aux offres par un marchand, en vérifiant qu'un marchand ne puisse pas modifier celles d'un autre. À l'état actuel, une offre est une offre, mais pas encore un avantage ou un cadeau en magasin. Certaines offres peuvent cependant avoir le statut de VFP.

Modèle métier



L'entité centrale de notre modèle comme on peut le voir est la classe Customer qui est assez chargée. Pour chaque Customer créé, il lui sera attribué une MultiLoyalityCard, d'où la liaison 1 à 1. Le Customer est liée à aucun ou plusieurs marchands (favoris) pour gérer les notifications en cas de changements d'horaires.

"PayementCard" représente la carte banquaire classique du client, qui a été renseignée à l'inscription. Un achat chez un commerçant partenaire incrémente la valeur de "cumulatedPoints" modulo 20 pour obtenir après des "points" visibles et utilisables par le Customer plus tard.

"LicensePlate" est un atribut optionnel à la création mais obligatoire d'avoir renseigné au moment d'utiliser l'avantage du parking.

En résumé on peut dire que cet objet métier a été rendu assez lourd, pour des soucis facilité durant le développement et pour garder une certaine base commune. Elle pourrait effectivement être allégé, en repérant zones qui ne dépendent pas directement d'un client et qui se répète aussi pour le marchant, afin d'en faire des objets ayant une responsabilité métier singulière, en suivant l'exemple de la "MultiLoyalityCard".

Forces et Faiblesses

Forces

L'architecture actuelle a été conçue pour obtenir un rendu minimal viable et pour répondre aux fonctionnalités nécessaires pour faire fonctionner l'activité commerciale originale du projet, qui consistait à développer la fidélité des clients envers un commerçant grâce à un système de points. Il est important de noter que le modèle actuel n'est pas complet. Toutefois, l'avantage est que chacun des composants implémente une logique métier propre et indépendante des autres composants du même niveau. Ainsi, même en cas de changement dans la logique d'enregistrement des marchands par exemple, seul un petit pourcentage du code sera réellement impacté, ne nécessitant pas une refonte de code conséquente, ce qui permet aux autres parties du système de rester fonctionnelles. Par conséquent on peut dire que le code actuel est scalable et modifiable.

Faiblesses

Cependant, notre travail comporte de nombreuses faiblesses. Tout d'abord, nous avons négligé l'interaction des objets métiers avec les composants. En effet, depuis la CLI, seul un identifiant représentant un objet Customer traverse les composants, ce n'est qu'à la fin que nous pouvons récupérer un client complet avec toutes les informations à jour pour pouvoir lui appliquer une éventuelle logique associée. Cette mauvaise pratique a été mise en place, car nous avons supposé que dans une session ouverte, nous aurions toujours accès à l'identifiant du client qui émet la requête voulue.

Certains éléments de notre architecture, notamment les composants de Registry en général, implémentent une approche légèrement CRUD en fonction du composant en question. Cette mauvaise pratique était toutefois nécessaire pour pouvoir récupérer, modifier et ajouter des objets nécessaires à la réalisation des fonctionnalités les plus importantes du MVP. Cependant, cela ne justifie pas cette approche. En effet, pour les composants d'enregistrement de marchands et d'agents de la DSI, une autre approche devrait être mise en place, impliquant une logique métier mise en place en amont de la simple communication avec les JPARepository respectifs. Ainsi, pour une implémentation future, en tant qu'axe d'amélioration, dans le cas des marchands, la responsabilité de leur enregistrement devrait être confiée aux composants de gestion de la DSI, afin d'ajouter une logique de vérification du client qui souhaite devenir marchand partenaire. Pour la DSI, il faudrait renvoyer vers un portail de connexion du secteur de la mairie associée.

Certains contrôleurs sont assez chargés et risquent de continuer à l'être, comme c'est le cas pour "AccountManager". En effet, il implémente les logiques de modification sur un compte client, mais celles-ci ont tendance à être de plus en plus sollicités. Il est donc nécessaire de décomposer ce composant en plusieurs sous-composants pour ne pas garder une "God classe".

Enfin, concernant l'utilisation des services externes, ceux-ci sont fonctionnels, mais ne sont pas intégrés de la bonne manière à l'application en général. En effet, ils sont implémentés en parallèle des autres fonctionnalités mises en place, ce qui est fortement visible sur le diagramme de composant. Le contrôleur communique directement avec un composant responsable de la logique du service externe, alors qu'il devrait y avoir entre eux un composant qui implémente la logique générale dans laquelle s'insère celle du service en question. Par exemple, dans le cas du parking, il faudrait un composant de gestion des offres qui reconnaît l'offre demandée, et, en fonction de cette offre, vérifie certaines contraintes ou délègue cette responsabilité à un autre composant dédié avant de faire passer l'objet métier "offre" au composant suivant qui confirmera l'achat, et ce, jusqu'à ce que celle-ci soit transmise au service externe.

Persistance

Pour cette partie, nous allons examiner la persistance dans le contexte de notre projet. Nous allons détailler les différentes entités de notre application et comment elles sont stockées dans la base de données.

Entités et liens

Notre application comprend plusieurs entités qui sont stockées dans une base de données relationnelle. Les entités sont les suivantes : **Customer**, **Address**, **Merchant**, **MultiLoyaltyCard**, **Offer** et **OfferCatalog** . Les liens entre ces entités sont définis par les annotations @OneToOne, @OneToMany et @ManyToMany.

La première entité est **Customer**. Cette entité est liée à **Merchant** par une relation @ManyToMany, car un client peut acheter chez plusieurs commerçants et un commerçant peut avoir plusieurs clients. Customer est également lié à **MultiLoyaltyCard** par une relation @Embedded, car la carte de fidélité est intégrée dans les informations du client.

L'entité **Customer** possède une relation un-à-un avec l'entité Address, représentée par l'annotation @Embedded. Cela signifie que l'objet Address est stocké dans la même table que l'objet Customer.

La deuxième entité est **Merchant**. Cette entité est liée à **Customer** par une relation @ManyToMany, car un commerçant peut avoir plusieurs clients et un client peut acheter chez plusieurs commerçants. **Merchant** est également lié à **OfferCatalog** par une relation @OneToOne, car chaque commerçant peut avoir un catalogue d'offres.

La troisième entité est **MultiLoyaltyCard**. Cette entité est incorporée dans l'entité Customer à l'aide de l'annotation @Embedded.

La quatrième entité est **Offer**. Cette entité est liée à **Merchant** par une relation @OneToOne, car chaque offre est liée à un seul commerçant.

La cinquième entité est **OfferCatalog**. Cette entité est liée à Offer par une relation @OneToMany, car chaque catalogue d'offres peut avoir plusieurs offres.

Stockage en base de données :
Notre application utilise une base de données relationnelle pour stocker les données. Les tables sont créées à partir des entités définies dans notre application. Les relations entre les tables sont par ailleurs définies en utilisant les annotations @OneToOne, @OneToMany et @ManyToMany.

Lorsqu'un objet est créé dans notre application, il est persisté dans la base de données. Pour cela, nous utilisons l'API JPA (Java Persistence API). JPA permet de stocker des objets Java dans une base de données relationnelle.

Répartition des points

100 pour tous les membres.