Rapport ISA :

Système de carte multi-fidélités

Equipe C

BEN AISSA Nadim
BOUBIA Marouane
ZOUBAIR Hamza
SAISSI Omar
EL GARMIT Youssef

SOMMAIRE

1. Introduction

Dans le cadre du projet ISA-DEVOPS du semestre 8, nous avions pour but de développer une architecture logicielle pour un système de carte multi-fidélité destiné aux clients, qui pourrait être utilisée dans une certaine zone géographique. Notre système vise à inciter les clients à acheter chez les commerces partenaires, en leur offrant des avantages tels que des offres gratuites ou des avantages de la collectivité territoriale associée en échange de points accumulés. Il permet également de charger de petits montants sur la carte pour des achats chez les partenaires. Les clients les plus fidèles peuvent également obtenir le statut de Very Faithful Person (VFP), ce qui leur permet de débloquer des avantages institutionnels supplémentaires.

L'objectif global de ce système est d'encourager les clients à dépenser davantage chez les commerces partenaires, tout en offrant des avantages attrayants aux clients fidèles.

2. Objets métiers :

La figure ci-dessous montre le diagramme de classes d'objets métiers de notre système de carte multi-fidélité: *(https://github.com/pns-isa-devops/isa-devops-22-23-team-c-23/blob/main/doc/diagrams/class-diagram.png)*

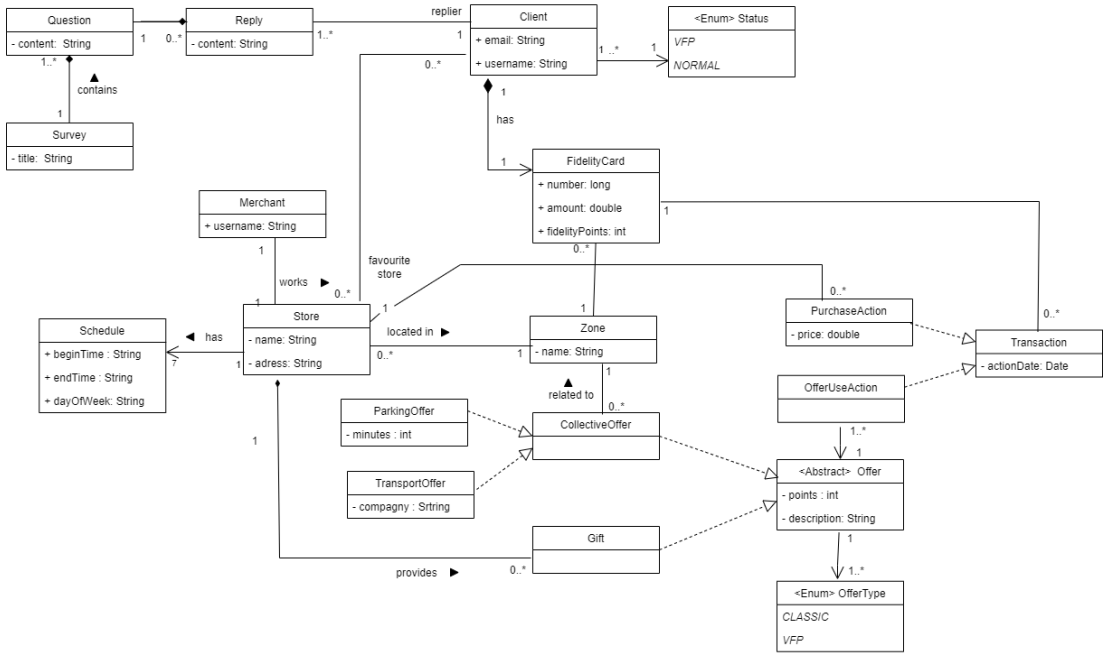


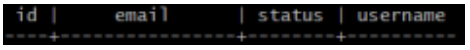
Diagramme d'objets métiers du "Multi Fidelity Card System"

- **Explication :**

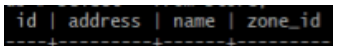
Notre système de carte multi-fidélité est organisé en zones géographiques, chacune est représentée par la classe Zone, à laquelle est associée une carte de fidélité. Chaque client possède une carte avec un statut ***NORMAL*** ou ***VFP***, permettant de savoir si ce dernier est éligible aux offres spéciales VFP. Nous retrouvons deux types d'offres : les collectives liées à une zone géographique (***TransportOffer*** ou ***ParkingOffer***), ou des cadeaux (***Gift***) liés à un magasin. Afin de suivre le nombre d'achats hebdomadaires des clients et leur éligibilité à un statut VFP, nous avons ajouté la classe Transaction, qui enregistre la date de chaque transaction effectuée par la carte de fidélité dans un magasin. Deux types de transactions ont été créés via l'héritage : ***PurchaseAction*** pour les achats bruts, et ***OfferUseAction*** pour l'utilisation des offres. Ces transactions permettent également d'obtenir des informations sur les habitudes de consommation et d'offrir des indicateurs sur l'utilisation du programme auprès des différents magasins. Enfin, nous avons ajouté la classe ***Survey*** pour les sondages créés par les administrateurs, comprenant plusieurs questions et les réponses des clients.

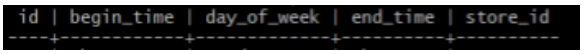
- **Migration vers la persistance :**

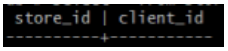
Les objets présentés dans le diagramme d'objets ont été migrés vers la persistance en base de données. Cependant, certaines classes, telles que les KPIs, n'ont pas été migrées, étant donné qu'elles sont créées à partir des transactions déjà stockées. Par conséquent, leur migration serait superflue et ne ferait qu'encombrer la base de données. Suite à la migration vers la persistance, nous avons obtenu les tables suivantes :

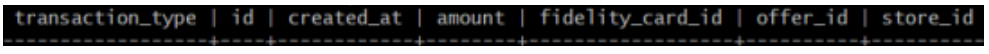
- Client : 

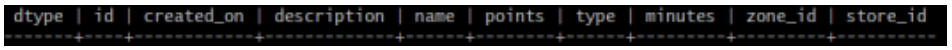
- Fidelity_card : 

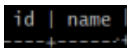
- Store : 

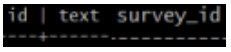
- Schedule : 

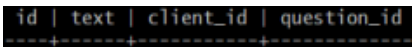
- Store_client_favorite : 

- Transaction : 

- Offer : 

- Survey : 

- Question : 

- Answer : 

Dans notre architecture de données, nous avons opté pour Single-Table Inheritance pour gérer l'héritage pour les transactions et les offres. Cette décision a été prise en raison de ses nombreux avantages, notamment la simplification de la structure de notre base de données et l'amélioration des performances en évitant les jointures entre les tables. De plus, cela nous permettra de maintenir notre base de données intacte en cas d'évolution de l'architecture et de refactoring. En ce qui concerne la relation entre le magasin et ses cadeaux et horaires, nous avons décidé d'utiliser “cascadeType = All” et "orphanRemoval" pour la relation @ManyToOne. Cette approche nous permet d'appliquer toutes les opérations effectuées sur le parent (Magasin) à ses enfants (Cadeaux et Horaires), ainsi que de supprimer les cadeaux et horaires orphelins. Cette cascade nous a permis de libérer de l'espace dans notre base de données sans avoir à effectuer de suppression implicite. On a appliqué également la même stratégie pour le sondage et les questions ainsi que les questions et les réponses.

En utilisant ces techniques dans notre modèle de données, nous pouvons garantir la cohérence et l'intégrité de nos données tout en améliorant les performances de notre base de données.

3. Diagramme de composants :

La figure ci-dessous montre le diagramme de composants de notre système de carte multi-fidélité: *(https://github.com/pns-isa-devops/isa-devops-22-23-team-c-23/blob/main/doc/diagrams/component-diagram.png)*

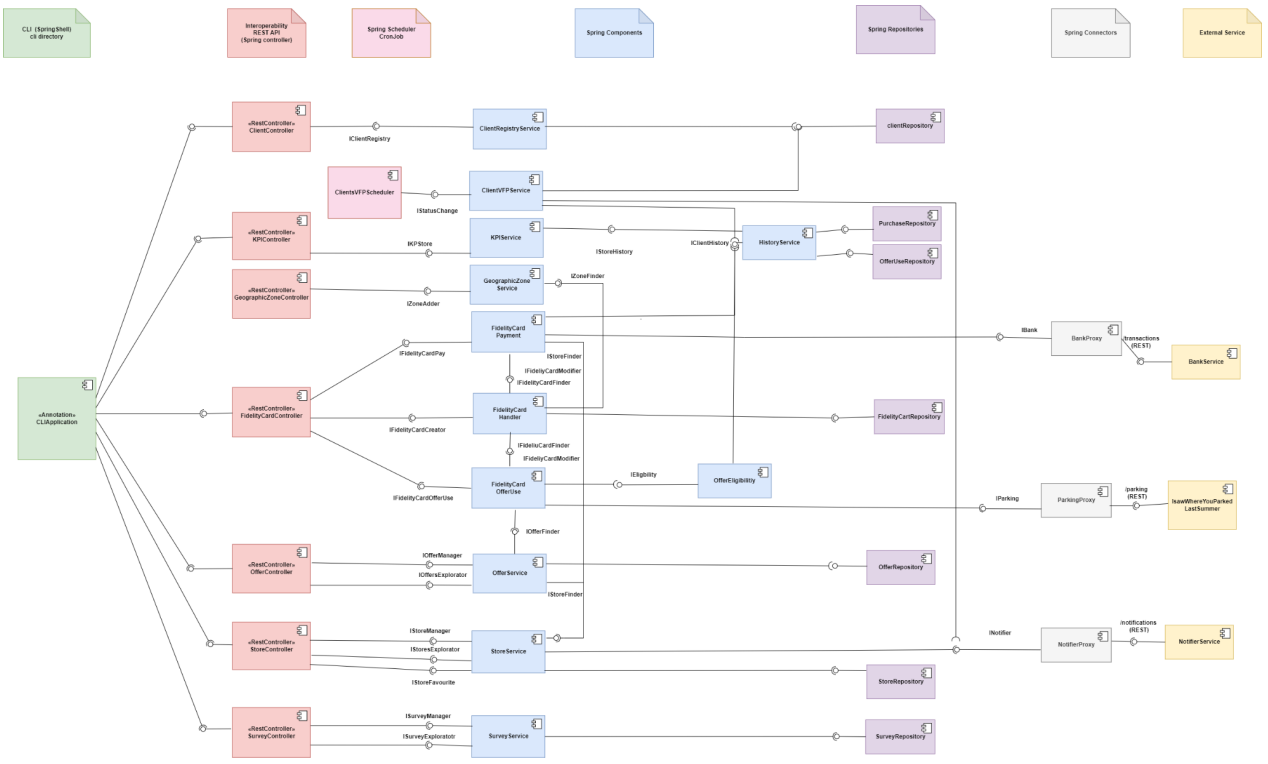


Diagramme de composants du "Multi Fidelity Card System"

- **Explication des interfaces fournies :**

IClientRegistration: Cette interface permet d'enregistrer de nouveaux clients dans le système;

```
public interface ClientRegistration {
  Client register(String username, String email) throws AlreadyExistingClientException;
}
```

IClientStatusManager: Cette interface permet de changer le statut d'un client donné.

```
public interface ClientStatusManager {
  void changeStatus(Client client);
}
```

IFidelityCardHandler: Cette interface permet de modifier la carte de fidélité en ajoutant ou retirant des points ainsi qu'un montant.

```
public interface IFidelityCardHandler {
  FidelityCard retrievePoints(FidelityCard fidelityCard, int nbPoints) throws NotEnoughPointsException;
  FidelityCard addPointsFromAmount(FidelityCard fidelityCard, double amount);
  FidelityCard addAmount(FidelityCard fidelityCard, double amount);
  FidelityCard retreiveAmount(FidelityCard fidelityCard, double amount) throws InvalidPaymentException;
}
```

IFidelityCardPaymentUse: Cette interface permet de charger et payer avec la carte en utilisant un montant précis.

```
public interface IFidelityCardPaymentUse {
    FidelityCard pay(Long cardId, double amount, String store)  throws FidelityCardNotFoundException,
InvalidPaymentException, StoreNotFoundException;
  FidelityCard recharge(Long cardId, double amount, String creditCard) throws InvalidPaymentException,
      FidelityCardNotFoundException;
}
```

IFidelityCardOfferUse: Cette interface permet d'utiliser les offres intégré dans notre système (cadeaux et parking dans notre cas) à travers la carte de fidélité.

```
public interface IFidelityCardOfferUse {
        FidelityCard useParking(Long parkingId, Long cardId)  throws NotEnoughPointsException,
NotEligibleClientException, OfferNotFoundException, FidelityCardNotFoundException,
ParkingProxyException;
        FidelityCard receiveGift(Long giftId, Long cardId)throws NotEnoughPointsException,
NotEligibleClientException, OfferNotFoundException, FidelityCardNotFoundException ;
}
```

IStoreKPIs: Cette interface permet d'avoir les KPIs en termes de chiffre d'affaires et cadeaux d'un magasin afin d'avoir des indicateurs sur l'utilisation du programme auprès des autres partenaires.

```
public interface IStoreKPIs {
  List<KPIsCA> getKPIsCAEvolution(Long storeId);
  List<KPIsOffers> getKPIsGiftsEvolution(Long storeId);
}
```

IStoreHistory: Cette interface permet d'avoir l'historique d'un magasin pour l'affichage des KPIs.

```
public interface IStoreHistory {
  Map<YearMonth, Double> getPurchaseHistoryPerMonth(Long storeId);
  Map<YearMonth, Integer> getGiftsHistoryPerMonth(Long storeId);
```

```
}
```

IClientHistory: Cette interface permet de connaître les achats hebdomadaires pour le statut VFP ainsi que les achats dans un store pour les cadeaux.

```
public interface IClientHistory {
  Map<LocalDate, Long> findNumberPurchasesLastWeekPerDay(FidelityCard card, LocalDateTime date);
  List<PurchaseAction> findPurchasesInStore(Store store, FidelityCard card);
 }
```

IStoreManager: Cette interface permet d'ajouter un nouveau magasin qui correspond au commerçant associé, ainsi que la possibilité de modifier les horaires du magasin.

```
public interface IStoreManager {
  Store addSchedule(Long storeId, ScheduleDTO scheduleDTO) ;
  Store register(String name, String address, String zone)  throws GeographicZoneNotFoundException;
  void deleteStore(Long storeId);
}
```

IStoresExplorator: Cette interface permet d'afficher les magasins disponibles dans une zone géographique donnée.

```
public interface IStoreExplorator {
  List<Store> consultStoresPerZone(String zone);
  List<Store> consultAll();
}
```

IStoreFavourite: Cette interface permet d'ajouter des magasins favoris aux clients pour qu'ils soient notifiés lors de la modification des horaires.

```
public interface IStoreFavourite {
  boolean addStoreToFavourite(Store store, Client client);
}
```

IOfferManager: Cette interface permet de gérer les offres avec l'ajout, la suppression et la modification des offres des commerçants.

```
public interface IOfferManager {
  void deleteOffer(Long offerId) throws OfferNotFoundException;
      Gift createGift(String name,String description, int points, Long store, OfferType type) throws StoreNotFoundException;
   ParkingOffer createParkingOffer(String name,String description, int points, int minutes, OfferType type, Long geographicZoneId) throws GeographicZoneNotFoundException;
}
```

IOffersExplorator: Cette interface permet de rechercher et de consulter les différentes offres.

```
public interface IOfferExplorator {
  List<Offer> exploreOffersByGeographicZone(Long geographicZoneId);
  List<Gift> exploreAllGifts();
```

```
  List<ParkingOffer> exploreParking();
}
```

ISurveyManager: Cette interface permet de créer, répondre et supprimer les sondages.

```
public interface ISurveyManager {
  Survey createSurvey(String name, List<Question> questions);
    Question respondToQuestion(Long surveyId, Long questionId, String answer, Long clientId) throws
NoSurveyFoundException;
  void deleteSurvey(Long surveyId) throws NoSurveyFoundException;
}
```

ISurveyExplorator: Cette interface permet de consulter les sondages créés.

```
public interface ISurveyExplorator {
  List<Survey> viewSurveys();
  Survey viewSurvey(Long surveyId) throws NoSurveyFoundException;
}
```

- **Explication des composants métiers :**

- ClientVFPService: responsable de la gestion des changements de statut des clients, en fonction de leur nombre d'achats hebdomadaires sur leur carte de fidélité. Il utilise l'interface *IClientHistory* du composant ***HistoryService*** pour communiquer avec ***PurchaseRepository***, ainsi que l'interface *INotifier* du ***NotifierProxy*** pour notifier les utilisateurs en cas de changement de statut. De plus, il fournit l'interface *IStatusChange* au cron job ***ClientVFPScheduler*** pour effectuer les changements de statut pour tous les clients chaque semaine.

- KPIsService: responsable de la gestion des KPIs des magasins fournit au ***KPIsController*** à travers l'interface IStoreKpis, il utilise ainsi l'interface *IStoreHistory* du composant ***HistoryService***.

- FidelityCardHandler: joue un rôle clé dans la gestion des cartes de fidélité. Il fournit ***IFidelityCardCreator*** au ***FidelityCardController*** pour la création de la carte et sert d'intermédiaire pour gérer la carte en modifiant les points et le montant pour les opérations de ***FidelityCardPayment*** et ***FidelityCardOfferUse*** grâce à ***IFidelityCardHandler***. De plus, il communique avec ***FidelityCardRepository*** pour stocker et récupérer les données relatives à la carte de fidélité.

- FidelityCardPayment: responsable de la gestion des paiements et recharges de la carte de fidélité en fournissant *IFidelityPaymentUse* au **FidelityCardController**. Il utilise l'interface *IBank* du ***BankProxy*** pour effectuer les opérations bancaires et l'interface *IFidelityCardHandler* pour mettre à jour le montant de la carte et ajouter des points.

- FidelityCardOfferUse: permet d'utiliser les différentes offres proposées à travers l'interface *IFidelityCardOfferUse* fournie au ***FidelityCardController***. Pour cela, il utilise l'interface *IEligibility* fournie par ***OfferEligibility*** pour déterminer l'éligibilité de l'utilisateur à recevoir une offre en se basant sur les transactions fournies par le ***HistoryService***. Dans le cas d'une offre comme le parking, il communique également avec le ***ParkingProxy***. De plus, le composant utilise ***IFidelityCardHandler*** pour mettre à jour la carte de fidélité.

- StoreService: permet de gérer les magasins en offrant des fonctionnalités d'ajout, de modification des horaires et de suppression via *StoreManager*, ainsi que la consultation des magasins avec *StoreExplorator*. De plus, il permet aux clients d'ajouter des magasins favoris en utilisant l'interface *IStoreFavorite*. En cas de modification des horaires, le composant communique avec ***NotifierProxy*** pour informer les clients favoris. Il est également lié à ***StoreRepository*** pour sauvegarder toutes les modifications effectuées sur les magasins.

- OfferService: permet principalement d'ajouter et consulter les différentes offres en fournissant *OfferExplorator* et OfferManger au ***OfferController***. Il communique également avec ***OfferRepository***.

- SurveyService: permet, à la demande du ***SurveyController***, d'ajouter, répondre et de consulter des sondages à travers ISurveyManger et ISurveyExplorator utilisant le SurveyRepository.

- **Analyse de l'architecture : forces, faiblesses et capacités d'évolution :**

L'architecture que nous avons développée repose sur l'utilisation de composants qui délivrent des services spécifiques, offrant ainsi une plus grande modularité et une réutilisation des fonctionnalités grâce à la séparation des responsabilités entre ces différents composants. Cette approche contribue à une meilleure maintenance du système, car chaque composant peut être mis à jour de façon indépendante, tout en renforçant également son extensibilité, en permettant l'ajout de nouveaux composants sans impacter les composants existants.

De plus, le respect des principes fondamentaux de l'architecture REST nous a garanti une communication claire entre les différents composants de l'architecture, ce qui facilite la coordination des services et l'intégration des nouveaux services. En effet, la documentation détaillée fournie par Swagger facilite également l'adoption et l'utilisation des APIs, ainsi que la maintenance et l'évolution de l'architecture.

En outre, notre architecture intègre des designs patterns bien établis, notamment l'injection de dépendances et le proxy, pour garantir la flexibilité et la modularité globales du système. Par

exemple, l'utilisation du proxy pour communiquer avec des services externes, comme les banques, les notifications ou les parkings, simplifie la mise en œuvre et assure la résilience du système en cas de panne des services externes. En outre, si nous devions changer de fournisseur de services de notification d' e-mail à SMS par exemple, nous pourrions facilement créer une nouvelle classe qui implémente l'interface *INotifier* et la configurer pour utiliser le nouveau fournisseur, sans compromettre la performance globale du système.

Ajoutant que notre architecture est pensée pour être hautement évolutive et adaptable aux besoins futurs. Les interfaces entre les différents composants sont définies et documentées avec soin, facilitant l'ajout de nouvelles fonctionnalités ou services. Par exemple, si l'on souhaite ajouter des fonctionnalités à la carte de fidélité, il suffit de modifier le composant ***FidelityCardHandler*** en ajoutant l'interface adéquate sans affecter les autres parties du système. De même, l'ajout d'une nouvelle offre de transport ne nécessite que l'ajout d'un nouveau composant de transport qui utilise les interfaces existantes dans OfferEligibility et FidelityCardOfferUse pour intégrer les nouvelles fonctionnalités. Si l'on souhaite également ajouter une fonctionnalité de gestion de stock pour les offres, un nouveau composant ***StockService*** peut être ajouté, fournissant l'interface *StockHandler* au composant ***OfferService*** pour augmenter le stock lors de l'ajout des offres, et au composant ***FidelityCardOfferUse*** pour réduire le stock lors de l'utilisation des offres, en communiquant avec OfferRepository.

Enfin, bien que notre architecture soit conçue pour être évolutive, cela peut également entraîner des défis en termes de gestion de la complexité et de dépendances entre les différents composants et qui nécessitent une attention particulière pour maintenir la simplicité et la clarté du système. De plus, nous avons constaté que notre CLI actuelle présentait une charge de travail importante, nous envisageons donc de la refactoriser en trois CLIs distinctes pour les différents acteurs du système, à savoir les administrateurs, les clients et les commerçants.

4. Auto évaluation :

BEN AISSA Nadim	BOUBIA Marouane	EL GARMIT Youssef	SAISSI Omar	ZOUBAIR Hamza
100	100	100	100	100

Nous avons assuré une répartition équitable des tâches entre les membres de notre équipe. Ainsi, à chaque étape de notre projet, nous avons attribué des tâches spécifiques à chaque membre en fonction de ses compétences et de ses intérêts, en alternant entre les parties ISA et DevOps pour une meilleure compréhension de l'ensemble du projet. Nous avons travaillé en étroite collaboration tout au long du projet pour nous assurer que chacun de nous était en mesure de remplir ses responsabilités de manière efficace et dans les délais impartis.

5. Conclusion :

Au cours de ce projet, nous avons eu l'occasion de renforcer nos compétences en architecture logicielle et en DevOps. Nous avons appris à concevoir une architecture évolutive en utilisant les patrons de conception appropriés et à travailler en équipe pour développer une application fonctionnelle. Nous avons également découvert les avantages de la méthodologie DevOps, qui nous a permis de mettre en place un processus de développement et de déploiement continu, garantissant ainsi la fiabilité de notre application. En utilisant des outils tels que Jenkins et Docker, nous avons automatisé la construction, les tests et le déploiement de notre application, ce qui nous a permis de gagner du temps et d'améliorer la qualité de notre code.

Globalement, ce projet a été une expérience enrichissante qui nous a permis de développer nos compétences techniques et de mieux comprendre les enjeux liés à l'architecture logicielle et à DevOps.

Enfin, nous tenons à exprimer notre gratitude à toute l'équipe enseignante pour leur encadrement et leur soutien tout au long du semestre. Leurs retours constructifs et leurs conseils ont été d'une grande aide pour nous permettre de mener à bien ce projet.