
# RAPPORT FINAL ISA

## TEAM A

DEAL Emma
LORCERY Morgane
MARINI Claire
MEHDI Khadidja
PRINCE Jules

Université Nice Sophia Antipolis

POLYTECH NICE-SOPHIA

# TABLE DES MATIÈRES

# INTRODUCTION

La carte multi-fidélité a pour but de favoriser le commerce local en incitant les clients à acheter chez plusieurs commerçants d'une même zone commerciale. Ce système permet de fidéliser les clients sur la zone et de créer une communauté de clients fidèles. Les mairies sponsorisent ce type d'action en offrant des avantages aux abonnés à la carte multi-fidélités, ce qui permet de renforcer la dynamique commerciale de la zone.

Notre application permet à des consommateurs d'obtenir une carte de multi-fidélité dans leur ville, de réaliser des achats avec cette dernière et d'obtenir des avantages.

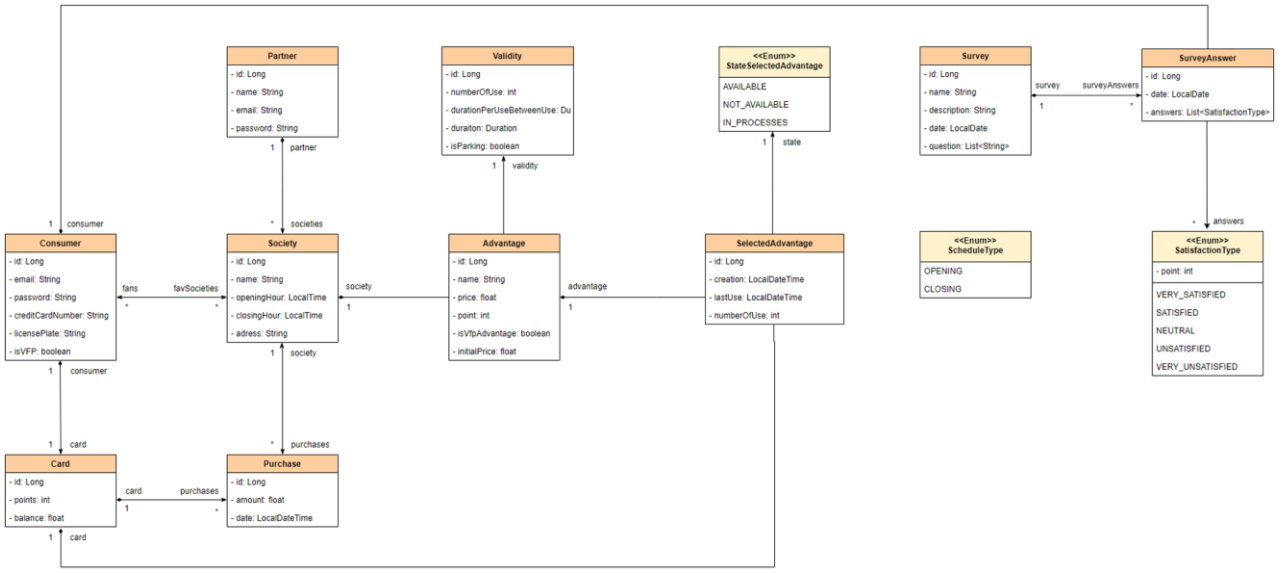
Cela permet aux commerçants de mettre en avant leur société, des créer des avantages personnalisés et d'avoir un suivi des statistiques d'achats et des bénéfices réalisés.

Les avantages peuvent être des biens matériels, comme des cafés offerts par exemple, mais aussi des services, comme des accès au cinéma ou au bus de la commune durant une période définie.

Les mairies quant à elles peuvent extraire des statistiques afin de voir si la carte procure un bénéfice à leur ville. Elles peuvent également créer des sondages de satisfactions auprès des consommateurs et en extraire les résultats.

# MODELE DE DONNEES

Voici notre modèle de données :




Nous allons faire des zooms sur certains éléments à clarifier :

## Les sociétés et partenaires

Les sociétés et les partenaires sont deux entités distinctes puisqu'un partenaire peut posséder plusieurs sociétés. Par exemple, le partenaire "Lidl" peut posséder plusieurs magasins Lidl dans la ville avec des adresses et des horaires d'ouvertures distinctes.

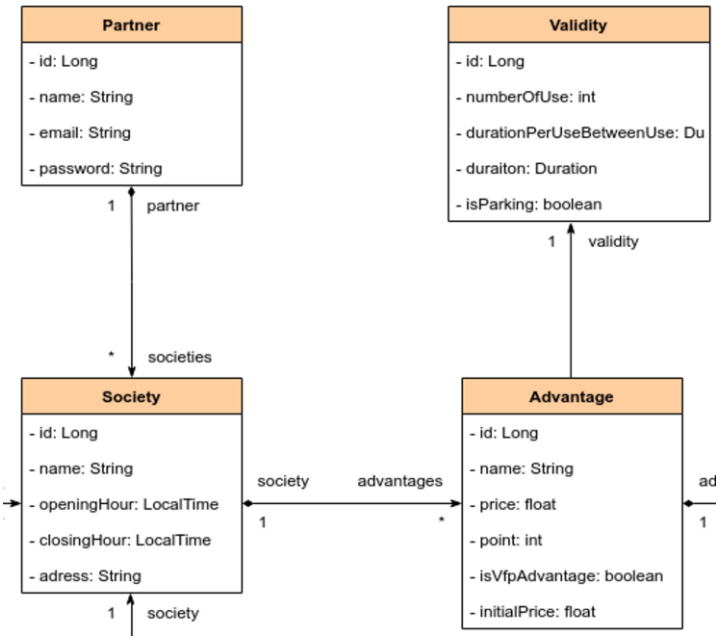
## Les achats

Lorsqu'un achat est fait celui est enregistré en base données dans la table "Purchase" afin de garder une trace pour le calcul de statistique et de statut VPF.

## Les avantages

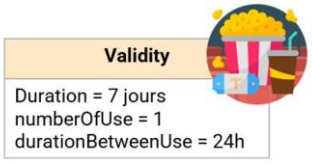
Nous allons plus particulièrement nous intéresser aux avantages :
Les avantages peuvent être des biens matériels, comme des cafés offerts, mais aussi des services, comme par exemple des accès au cinéma ou au bus de la commune durant une période définie.

Pour les biens matériels, l'objet avantage est assez simple. Il possède un nom, une société créatrice, un prix (qui peut être égal à zéro si le cadeau est offert dans sa globalité), un prix d'origine (utile pour calculer les pertes des sociétés), un nombre de points nécessaire pour en bénéficier et un champ "isVfpAdvatange" qui nous indique s'il faut être VFP pour en profiter.



Si c'est un service, l'avantage va également posséder un objet de validité. Cet objet va permettre d'indiquer la durée totale de l'avantage, le nombre d'utilisation autorisé et la durée minimum à attendre entre chaque utilisation.
Voici un exemple concret d'avantage de type service :

**Exemple** – Accès au cinéma valable une semaine avec autorisation de voir un film par jour

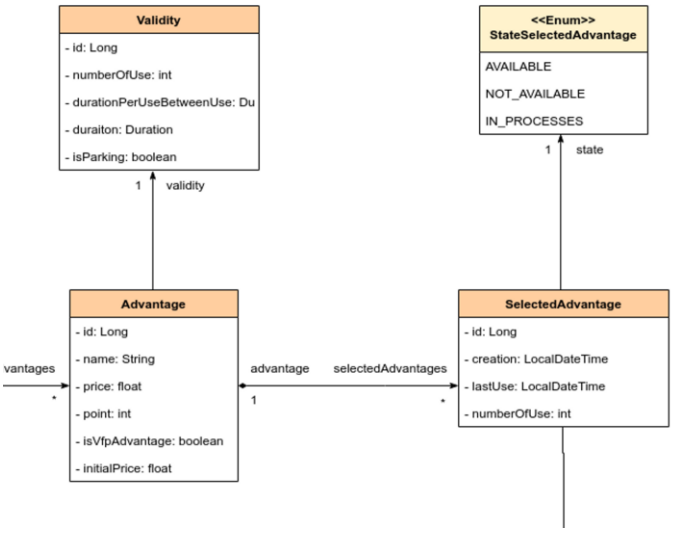



La durée totale est d'une semaine, le nombre d'utilisation est d'une fois toutes les 24h.

## Utilisation des avantages

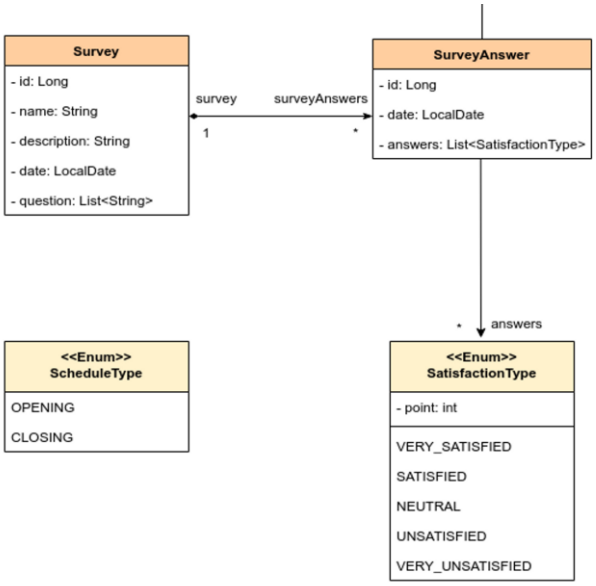
Nous avons réfléchi à une utilisation en deux temps.
Premièrement, le consommateur va pouvoir "sélectionner" l'avantage qui lui plaît. La sélection sera effective si le consommateur possède assez de points sur sa carte et si son statut VFP le permet.
De cette sélection, va être créé un objet "SelectedAdvantage" avec un état "Available". Cet objet a pour but de garder une trace des avantages souscrit pour le calcul des statistiques, mais également de garder une trace des avantages sélectionnés par un consommateur qui pourront par la suite être utilisés.
Dans l'application, ce "SelectedAdvantage" correspond au QR code.
Ensuite, le consommateur va pouvoir utiliser son "selectedAdvantage"

Dans le cas d'un bien matériel, il devra se rendre à la société qui propose l'avantage et montrer son QR code. Une fois scanné, cela aura pour action de faire passer l'état du "SelectedAdvantage" en "Not_Available" et le commerçant pourra donner le cadeau au consommateur.
Dans le cas d'un service, le consommateur pourra scanner son QR code en rentrant dans le bus ou au cinéma par exemple et cela aura pour action de vérifier si la validité de l'avantage est valable (si le nombre d'utilisation n'est pas dépassé ou la durée d'utilisation terminée). S'il est bien valable, le nombre d'utilisation du "SelectedAdvantage" sera incrémenté de 1 et sa dernière utilisation sera l'heure et la date actuelle. Dans le cas contraire, si la durée de l'avantage est finie, il passera en état "Not_Available".




Pour les avantages de parking, il y aura un bouton à appuyer pour ouvrir une session de parking d'une durée de 30 minutes. Cela aura pour action de vérifier si la validité de l'avantage est valable (si le nombre d'utilisation n'est pas dépassé ou la durée d'utilisation terminée). S'il est bien valable, le nombre d'utilisation du "SelectedAdvantage" sera incrémenté de 1 et sa dernière utilisation sera l'heure et la date actuelle. Une requête est envoyée à l'API externe pour donner le début de la session et l'état du "SelectedAdvantage" passera en "In_Process".
Après 30 minutes, l'API de parking enverra une requête à l'application ce qui aura pour effet de passer l'état du "SelectedAdvantage" en "Available" et de notifier le consommateur.

## Sondages de satisfaction

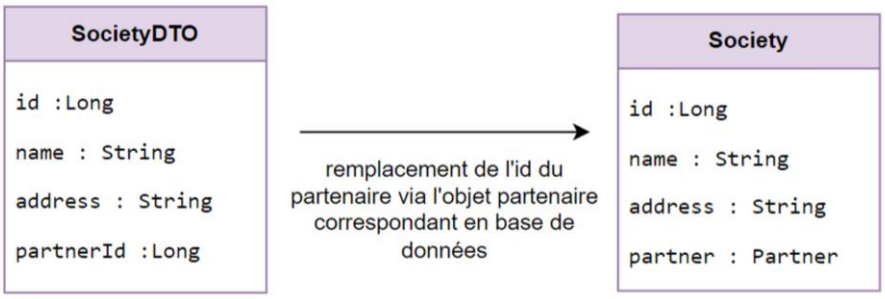



Un sondage est un objet comportant une liste de questions de taille n. Une réponse à un sondage est faite par un consommateur et comporte un nombre n de réponses. Les réponses sont des énum de satisfactions. Ils correspondent à un nombre de points sur 5 en termes de satisfaction. L'extraction des résultats d'un sondage correspond à une note de satisfaction sur 5.

# COMPOSANTS

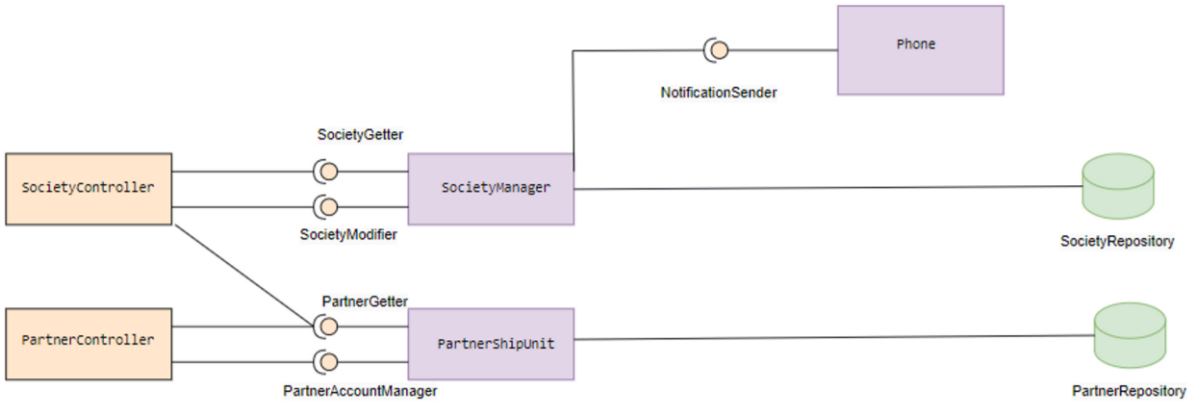
Voici des diagrammes de composants de l'architecture de notre projet. Ils illustrent les différentes fonctionnalités implémentées.
Dans la plupart des cas, nos contrôleurs reçoivent des objets de type DTO qui ne contiennent que des ID et doivent respecter des contraintes grâce à des validateurs.
Ces objets sont mappés dans les contrôleurs en entité qui ne contiennent plus d'ID, mais d'autres entités.
Cela est permis grâce à la collaboration d'interfaces de type "getter" pour récupérer les objets correspondant aux ID.

Par exemple, voici un objet SocietyDto qui est mappé en entité Society :



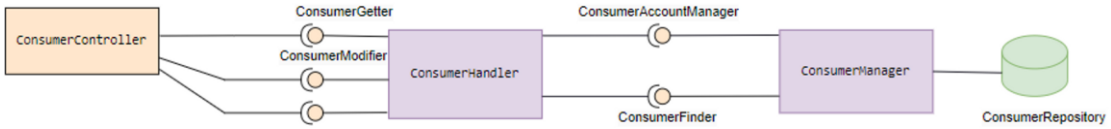

## Création et modification des partenaires et de leur société

Voici des diagrammes de composants de l'architecture de notre projet. Ils illustrent les différentes fonctionnalités implémentées.



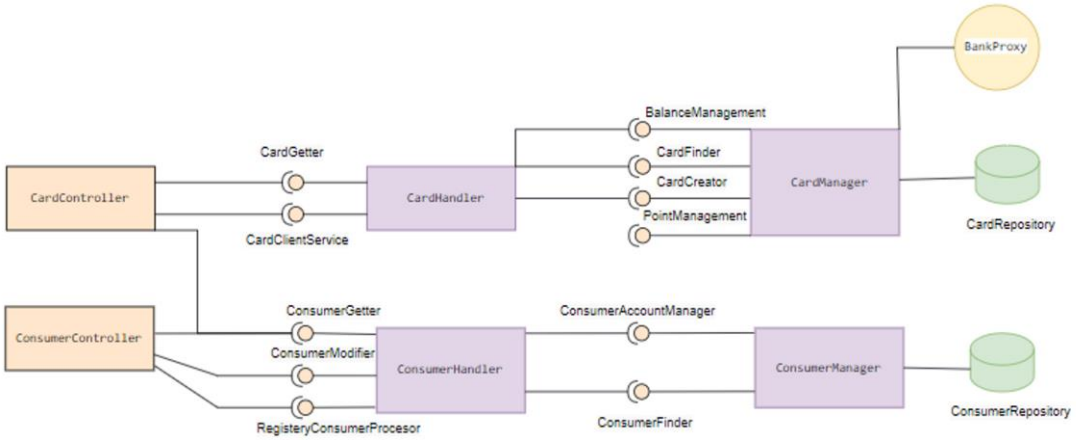

Lorsque des administrateurs créent des partenaires ou des sociétés sur l'application, les contrôleurs reçoivent des objets DTO et réalisent un mappage vers des entités via les interfaces PartnerGetter et SocietyGetter qui seront ensuite transmises aux composants SocietyManager et PartnerShipUnit. Les composants SocietyManager et PartnerShipUnit n'ont pas de fortes responsabilités, ce sont des composants dits "CRUD". Ils permettent de créer, modifier et supprimer des objets de type "Partner" et "Society". Ils peuvent aussi enregistrer des modifications en base de données.
SocietyManager communique avec un composant Phone, via l'interface NoitifcationSender. Il permet de notifier les consommateurs si les horaires d'une société ont été modifiées.

## Création et modification des consommateurs




Lorsqu'un consommateur crée ou modifie son compte sur l'application, le contrôleur reçoit un objet DTO et réalise un mappage vers des entités qui seront ensuite transmises au composant ConsumerHandler.
Le composant ConsumerHandler a pour responsabilité de vérifier la conformité et l'unicité des objets Consumer à créer ou modifier. Les mails, numéros de cartes et plaques d'immatriculation doivent être uniques. Si les conditions sont respectées, les consommateurs seront créés ou modifiés via l'interface ConsumerAccountModifier.
Le composant ConsumerManager est un composant dit "CRUD". Il permet de créer, modifier ou supprimer des objets "Consumer".

## Création d'une carte et son rechargement




Lorsqu'un consommateur créé une carte sur l'application, l'ID du consommateur est reçu par le contrôleur de carte puis l'objet correspondant est récupéré via l'interface ConsumerGetter et est envoyé au composant **CardHandler.**

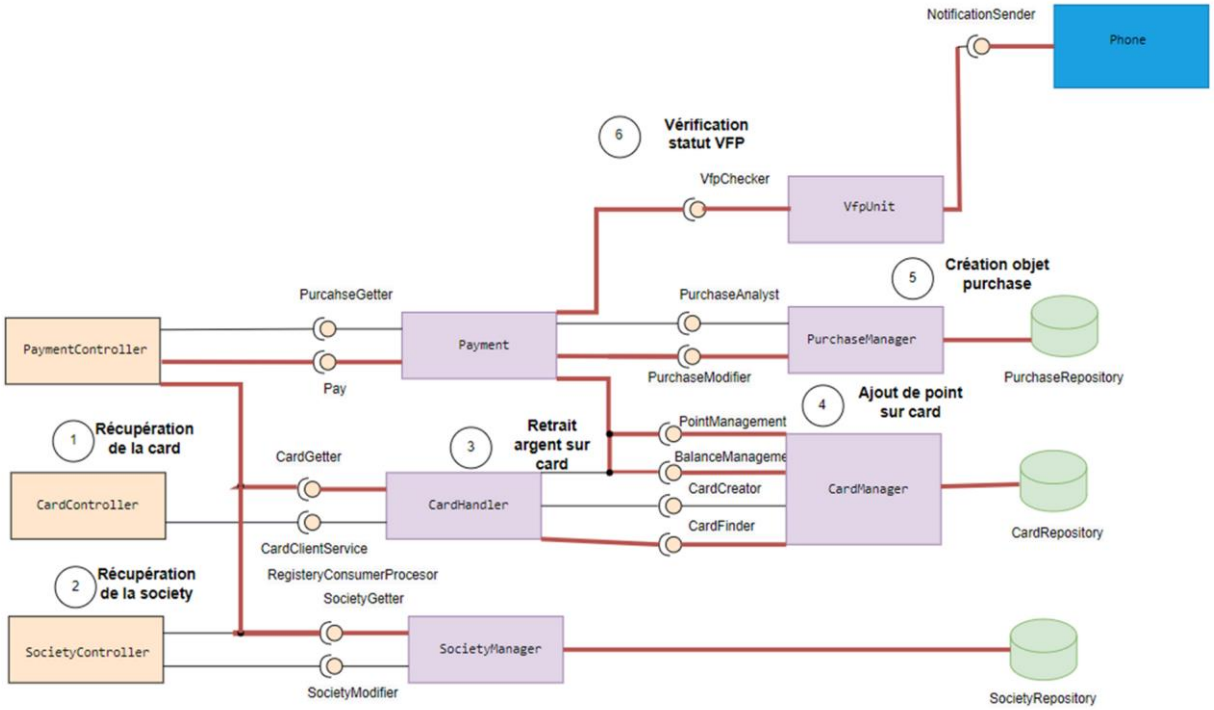
Le composant **CardHandler** a pour responsabilité d'ordonner la création d'une carte via l'interface **CardCreator**.
Lorsqu'un consommateur recharge le solde de sa carte sur l'application, le montant de la recharge et l'ID du consommateur est reçu par le contrôleur de carte puis l'objet correspondant est récupéré via l'interface ConsumerGetter et est envoyé au composant **CardHandler.**
**CardHandler** gère la modification du solde de la carte via l'interface **BalanceManagement,** tout en vérifiant qu'une somme limite n'a pas été dépassé.
Quant à lui, le composant **CardManager** a pour responsabilité de modifier les champs de la carte comme le nombre de points ou le solde. Pour le solde, le composant est en contact avec une API externe de banque. Selon la réponse de la banque, le solde de la carte sera augmenté ou non.
Voici les logs de la banque lors d'un ajout de 12 euros sur la carte d'un consommateur.

```
2023-04-06 08:23:44 [Nest] 1  - 04/06/2023, 6:23:40 AM     LOG [NestFactory] Starting Nest application...
2023-04-06 08:23:46 [Nest] 1  - 04/06/2023, 6:23:46 AM     LOG [InstanceLoader] AppModule dependencies initialized +5
2023-04-06 08:23:47 [Nest] 1  - 04/06/2023, 6:23:47 AM     LOG [RoutesResolver] AppController {/cctransactions}: +849ms
2023-04-06 08:23:47 [Nest] 1  - 04/06/2023, 6:23:47 AM     LOG [RouterExplorer] Mapped {/cctransactions, GET} route +466ms
2023-04-06 08:23:47 [Nest] 1  - 04/06/2023, 6:23:47 AM     LOG [RouterExplorer] Mapped {/cctransactions, POST} route +76ms
2023-04-06 08:23:48 [Nest] 1  - 04/06/2023, 6:23:48 AM     LOG [NestApplication] Nest application successfully started +350ms
2023-04-06 09:39:42 Bank withdrawal made in the amount of 12 with credit card :8969830123456789896983
```

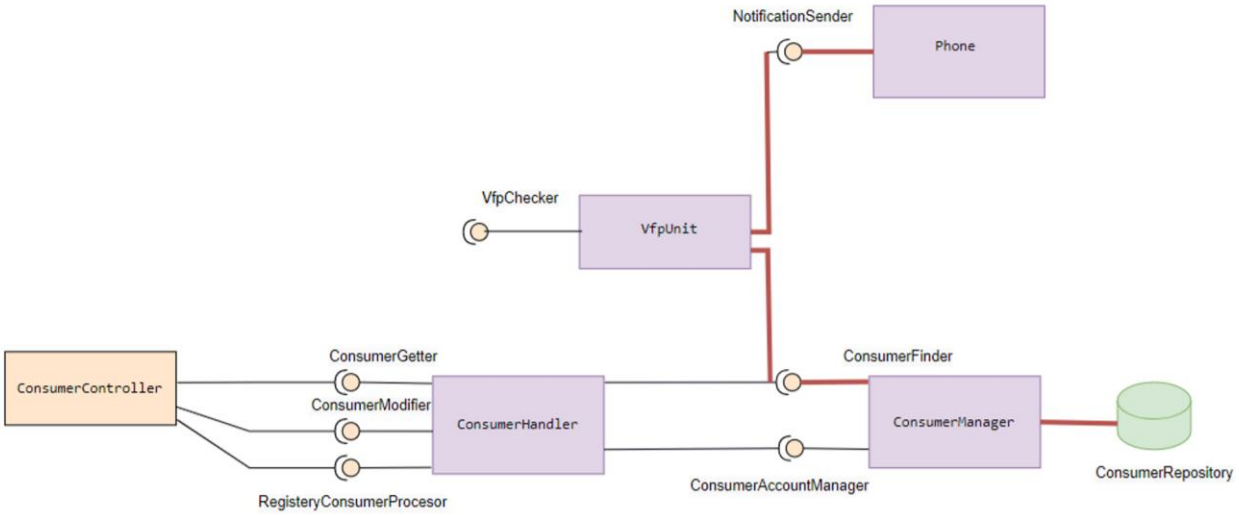
## Réaliser un achat avec sa carte multi-fidélités

Lorsqu'un consommateur réalise un achat en magasin avec sa carte, un objet paiement DTO est reçu par le contrôleur de paiement contenant l'ID de la carte et l'ID de la société concernée, l'objet est mappé vers une entité nommée Purchase.
Via l'interface Pay, le composant Payment reçoit l'objet Purchase. Le composant Payment a une forte responsabilité puisqu'il orchestre toutes les étapes autour d'un achat :

- Retirer la somme de l'achat sur la carte via l'interface BalanceManagement.
- Ajouter des points sur la carte via l'interface PointManagement.
- Créer l'objet Purchase via l'interface PurchaseModifier.
- Vérifier le statut VFP du consommateur suite à cet achat via l'interface VfpChecker.



## Vérification automatique du statut VFP

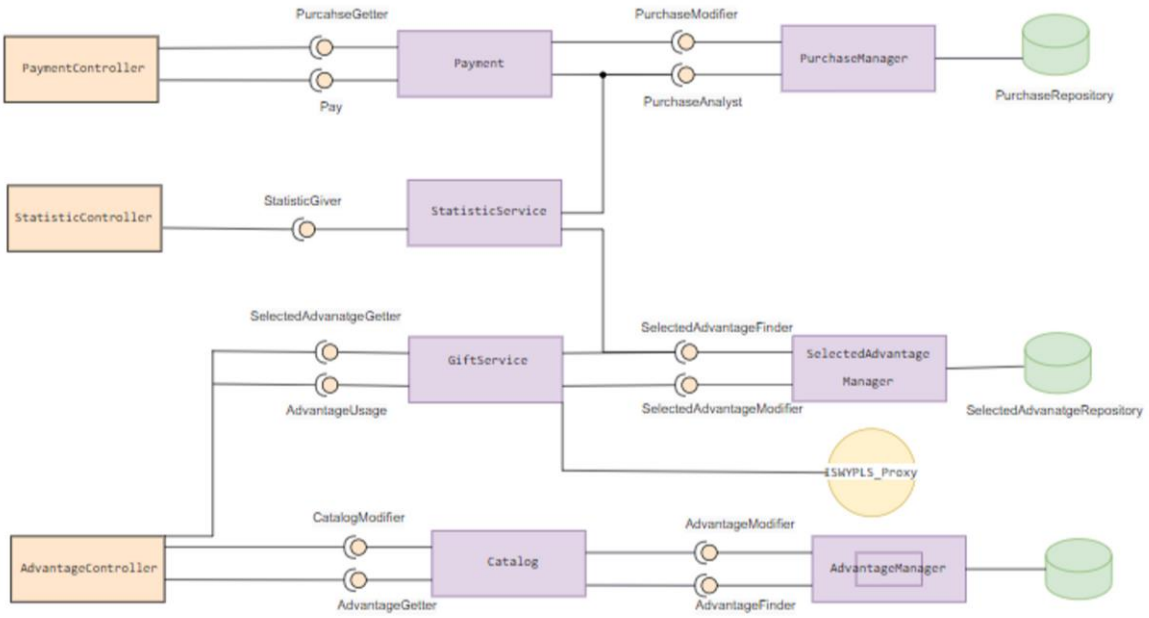



Comme on a pu le voir plus haut, le statut VFP d'un consommateur est vérifié à chaque fois qu'il réalise un achat, mais cela n'est pas suffisant. Il faut également vérifier si un consommateur n'est plus VFP après trop de temps passé sans avoir réalisé un achat. Si le consommateur va bientôt perdre son statut de VFP, nous voulons pouvoir le notifier avant qu'il ne le perde.

Cette responsabilité est assignée au composant VFPUnit dont le rôle est de récupérer tous les consommateurs via l'interface ConsumerFinder et, pour chacun, de vérifier si leur fréquence d'achat permet d'être VFP ou non. Il peut également notifier des consommateurs via l'interface NotificationSender.

Cette fonction est réalisée par une fonction automatique qui se déclenche toutes les 5 minutes.

## Extraction des statistiques




Lorsqu'un administrateur ou un partenaire veut extraire des statistiques à partir de l'application, une requête est reçue par le contrôleur de statistiques avec un ID de société ou non, car il existe deux types de statistiques :

- Les profits, que l'on peut extraire d'une société en particulier ou de la ville entière.

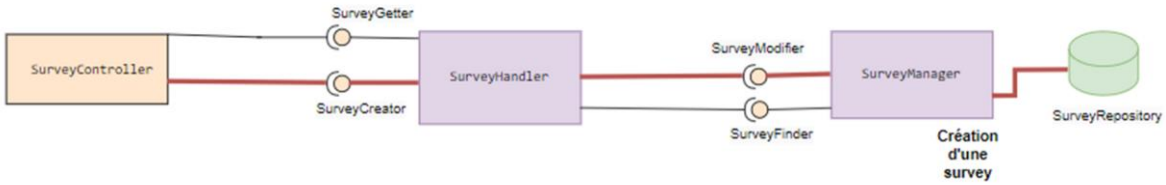
- Les fréquences d'achats dans les 7 derniers jours, que l'on peut extraire d'une société en particulier ou de la ville entière.

C'est le composant StatistiqueService qui retourne ces statistiques, il a donc une forte responsabilité.

Pour les profits, il réalise un calcul entre les entrées et les sorties d'argents. Les entrées d'argent sont récupérées via l'interface PurchaseAnalyst et les sorties via SelectedAdvanatageFinder.

Pour les fréquences d'achats, il extrait les derniers achats via l'interface PurchaseAnalyst et réalise un calcul pour obtenir une fréquence sur les 7 derniers jours.

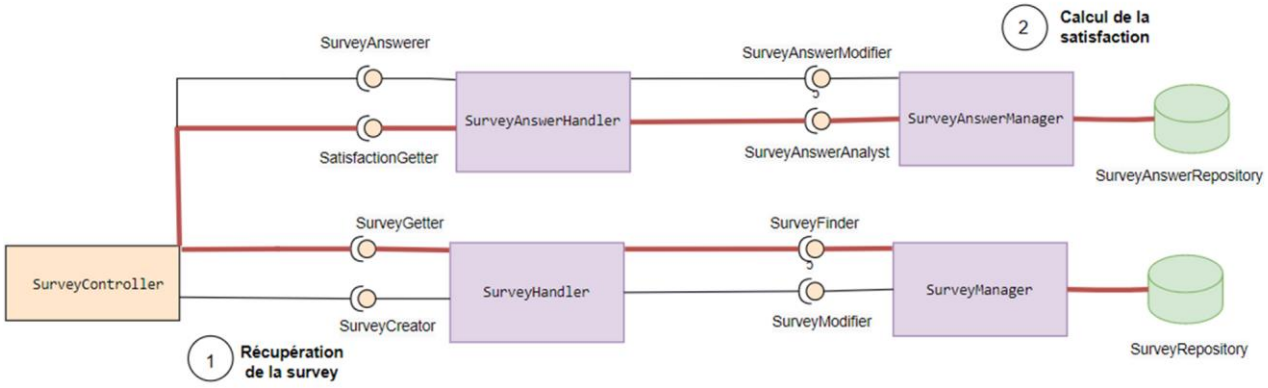
## Création d'un sondage de satisfaction




Lorsqu'un administrateur créé un sondage sur l'application, le contrôleur des sondages reçoit un objet SurveyDTO qu'il mappe en entité Survey puis qui le passe au composant SurveyHandler via l'interface SurveyCreator.

Le composant SurveyHandler a pour responsabilité de vérifier la conformité d'un sondage avant sa création. Le composant SurveyManager a un rôle "CRUD" permettant de créer des sondages.

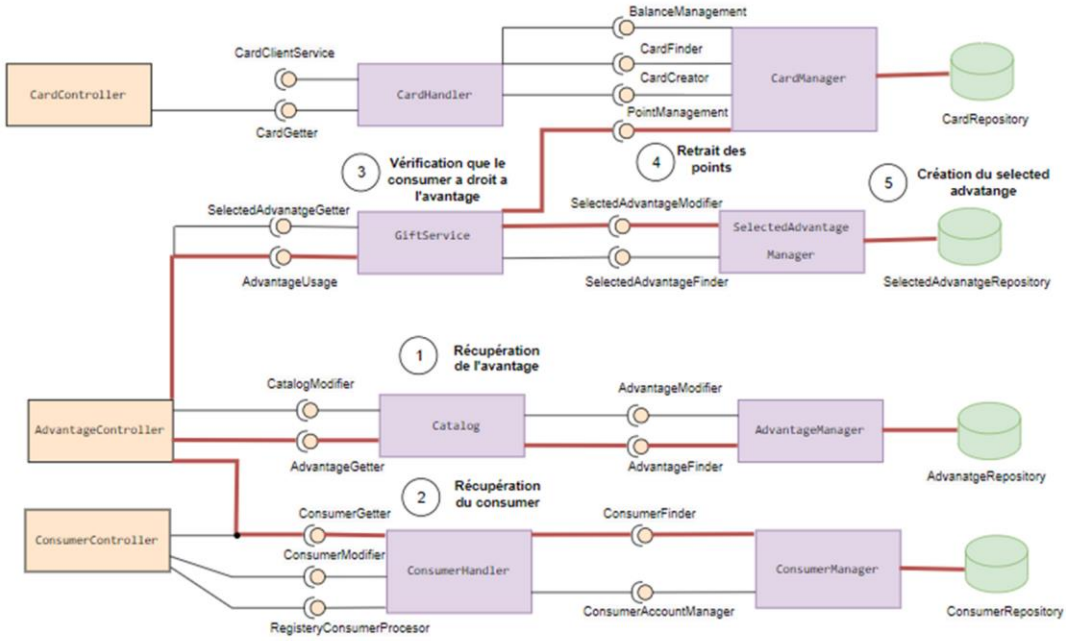
## Extraction des résultats de sondage




Lorsqu'un administrateur extrait les résultats d'un sondage sur l'application, le contrôleur des sondages reçoit un ID de sondage dont il récupère l'objet correspondant via l'interface SurveyGetter et l'envoi au composant SurveyAnswerManager via l'interface SurveyGetter.

Le composant SurveyAnswerManager a une forte responsabilité, en fonction d'un sondage, il va récupérer toutes les réponses associées et faire un calcul donnant la satisfaction moyenne sur 5 au sondage.

## Sélection d'un avantage

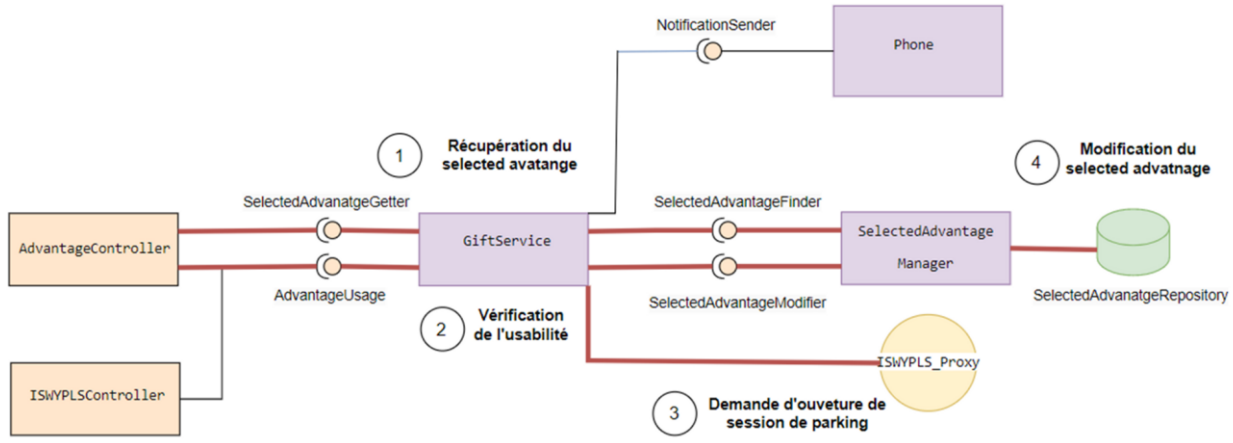



Lorsqu'un consommateur sélectionne un avantage sur l'application, l'ID du consumer et celui de l'avantage concerné sont reçu par le contrôleur d'avantage. Les objets sont récupérés via les interfaces ConsumerGetter et AdvantageGetter et passés au composant GiftService via l'interface AdvantageUsage.

Le composant GiftService a une forte responsabilité puisqu'il orchestre toutes les étapes autour de la sélection et l'utilisation d'un avantage :

- Vérifier que le consommateur ait le droit de sélectionner l'avantage (en fonction du nombre de points, du statut VFP et de la présence d'une plaque d'immatriculation, si c'est un avantage de parking).
- Faire un retrait de point sur la carte via l'interface PointManagement.
- Créer l'objet SelectedAdvanatge via l'interface SelectedAdvantageModifier

## Utilisation d'un avantage

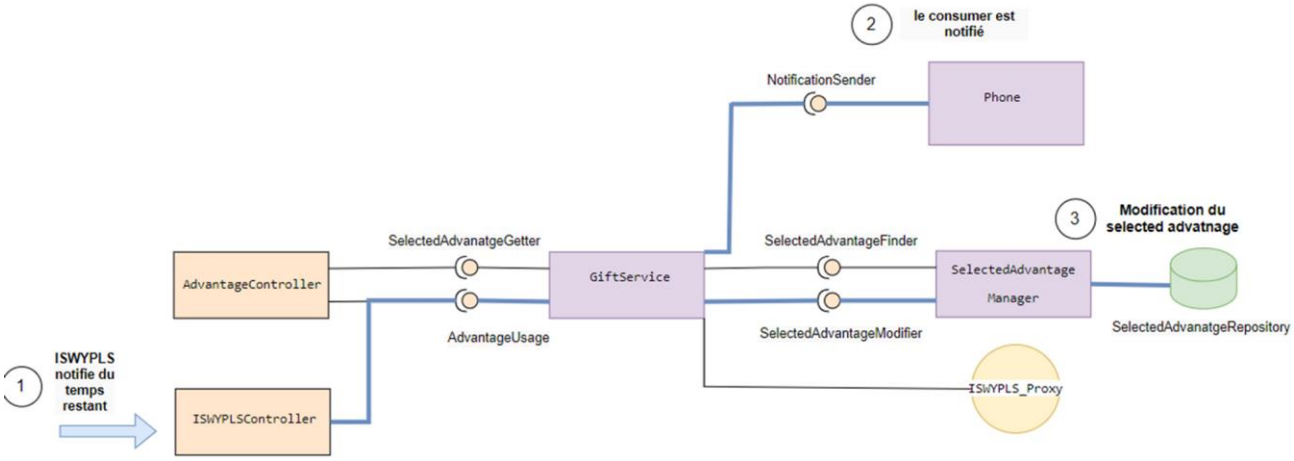



Lorsqu'un consommateur scan un QR code pour utiliser un avantage sélectionné, l'ID de l'avantage sélectionné est reçu par le contrôleur d'avantage. L'objet SelectedAdvatage est récupéré via l'interface SelectedAdvatagGetter et est passé au composant GiftService via l'interface AdvantageUsage.

Le composant GiftService a une forte responsabilité puisqu'il orchestre toutes les étapes autour de la sélection et l'utilisation d'un avantage :

- Vérification de l'usabilité de l'avantage si la validité de l'avantage n'est pas nulle (vérifier que la durée et le nombre d'utilisation ne sont pas dépassés).
- Vérification de l'état de l'avantage (il faut que celui-ci soit en Available).
- Demande de création de session auprès de l'API de parking si c'est un avantage avec une validité de type parking.
- Modification de l'objet SelectedAdvantage via l'interface SelectedAdvantageModifier, modification de l'état, incrémentation du nombre d'utilisation et modification de la date de dernière utilisation.

Si c'est un avantage avec une validité de type parking, 5 minutes avant la fin et à la fin de la session, l'API parking envoi des requêtes sur le contrôleur ISWYPLS. Cela produit deux actions : le consommateur concerné par la session est notifié via l'interface NotificationSender et le SelectedAdvantage concerné voit son statut modifier.




Voici les logs de l'API parking correspondant à l'ouverture puis à la fermeture d'une session de 30 minutes.

La conformité de la plaque d'immatriculation est vérifiée ainsi que le fait qu'aucune autre session ne soit en cours pour la même plaque.

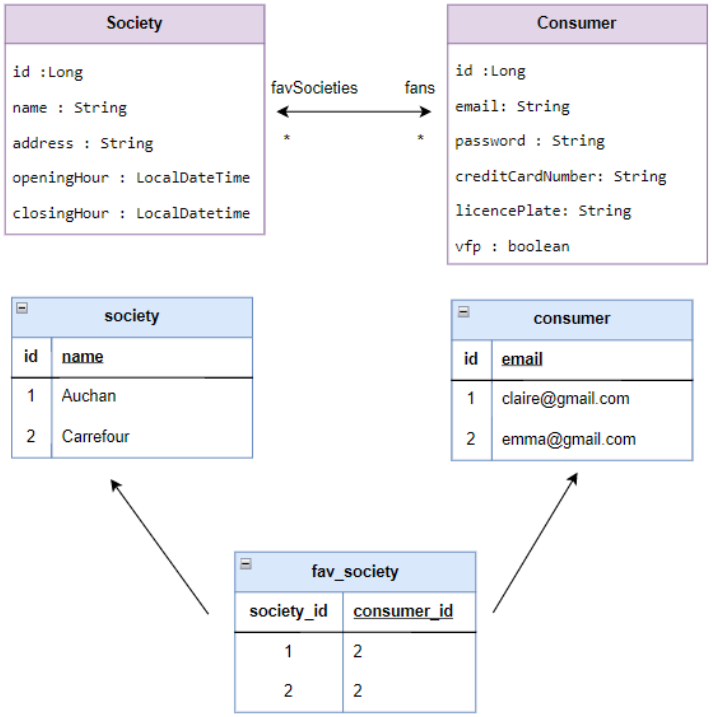
```
2023-04-09 08:35:24 [Nest] 1  - 04/09/2023, 6:35:24 AM     LOG [NestFactory] Starting Nest application...
2023-04-09 08:35:24 [Nest] 1  - 04/09/2023, 6:35:24 AM     LOG [InstanceLoader] DiscoveryModule dependencies initiali
2023-04-09 08:35:24 [Nest] 1  - 04/09/2023, 6:35:24 AM     LOG [InstanceLoader] AppModule dependencies initialized +1ms
2023-04-09 08:35:24 [Nest] 1  - 04/09/2023, 6:35:24 AM     LOG [InstanceLoader] ScheduleModule dependencies initialized +0ms
2023-04-09 08:35:24 [Nest] 1  - 04/09/2023, 6:35:24 AM     LOG [RoutesResolver] AppController {/iswypls}: +8ms
2023-04-09 08:35:24 [Nest] 1  - 04/09/2023, 6:35:24 AM     LOG [RouterExplorer] Mapped {/iswypls/sessions, GET} route +3ms
2023-04-09 08:35:24 [Nest] 1  - 04/09/2023, 6:35:24 AM     LOG [RouterExplorer] Mapped {/iswypls/sessions/:plate, GET} route +0ms
2023-04-09 08:35:24 [Nest] 1  - 04/09/2023, 6:35:24 AM     LOG [RouterExplorer] Mapped {/iswypls, POST} route +1ms
2023-04-09 08:35:24 [Nest] 1  - 04/09/2023, 6:35:24 AM     LOG [NestApplication] Nest application successfully started +5ms
2023-04-09 08:35:24 Service listening 👍 9080
2023-04-09 08:42:03 Create session iswypls
2023-04-09 08:42:03 { plate: 'FR12AB34BA', remainingTime: 30 }
2023-04-09 08:42:03 check if session exists
2023-04-09 08:42:03 check if plate is valid
2023-04-09 08:42:03 Session create successfully with plateFR12AB34BA at 2023-04-09T06:42:03.358Z
2023-04-09 08:42:03 POST /iswypls 201 100 - 14.412 ms
2023-04-09 09:07:03 The session will be valid for another 5 minutes : FR12AB34BA call multi fidelity api
2023-04-09 09:12:03 End of the session : FR12AB34BA call multi fidelity api
```

# PASSAGE A LA PERSISTANCE

Pour passer à la persistance notre implémentation a très peu changer.

Nos classes sont devenus des entités avec des clés étrangères.

Nous avons utilisé une table de jointure pour les sociétés favoris des consommateurs.En effet avant les sociétés favorites étaient représenté par une liste d'id de société dans la classe "consumer". Maintenant c'est une table de jointure nommé "fav_society" :




Pour la liste de questions des sondages et la liste des réponses qui est une liste de String nous avons utilisé nous avons utilisé l'annotation @elementcollection. Cela signifie que les éléments appartiennent entièrement aux entités contenantes : ils sont modifiés lorsque l'entité est modifiée, supprimés lorsque l'entité est supprimée, etc. Ils ne peuvent pas avoir leur propre cycle de vie. Ainsi les questions sont dans une table a part mais ne peuvent exister indépendamment des sondages.

# CLI

Nous disposons d'une CLI avec divers scénarios comme la création de consommateurs et de leur carte, la recharge d'une carte, la création de partenaires et de leurs sociétés, la sélection et l'utilisation d'avantage, la réalisation d'achat ou encore l'extraction de résultats de sondage et de statistique.

# TESTS

Nous possédons 273 tests au total avec des tests cucumbers, des tests de composants, de contrôleurs et de connecteurs.

Il y a des tests traversants, testant des fonctionnalités qui utilisent des API externes.

# FORCES ET FAIBLESSES

## Forces

Nous possédons une architecture solide avec des composants avec des fortes, bonnes et uniques responsabilités. L'ajout d'un objet validité pour les avantages fait que nous pouvons facilement ajouter de nouveaux avantages de type service avec des durées et des nombres d'utilisations limités, comme des accès au cinéma ou à la piscine de la ville.

## Faiblesses

Même si nous savons que la sécurité ne faisait pas partie de ce projet, nous avons énormément de failles que nous aurions aimé réparer. Par exemple, le fait que n'importe quelle personne peut accéder à toutes les actions de L'API ou que même un consommateur pourrait potentiellement créer un sondage. Pour pallier ce défaut, il faudrait sécuriser les routes avec un système de rôle.
Nous aurions également aimé ajouter des contraintes sur la base de données autre que les clés primaires et étrangères, comme des contraintes d'unicités.

# DISTRIBUTION DES POINTS

Toute l'équipe s'étant investie à son maximum, nous attribuons la note de 100 à chaque membre. Même si le travail a été majoritairement partagé entre ISA et DevOps, nous avons taché de tous toucher aux deux parties et de tous comprendre l'entièreté du projet.