Rapport Final ISA

Winter4Everybody

Équipe B

DUMANOIS Arnaud
ORFILA Marie
REMY Clément
ZANDIAN Nikan

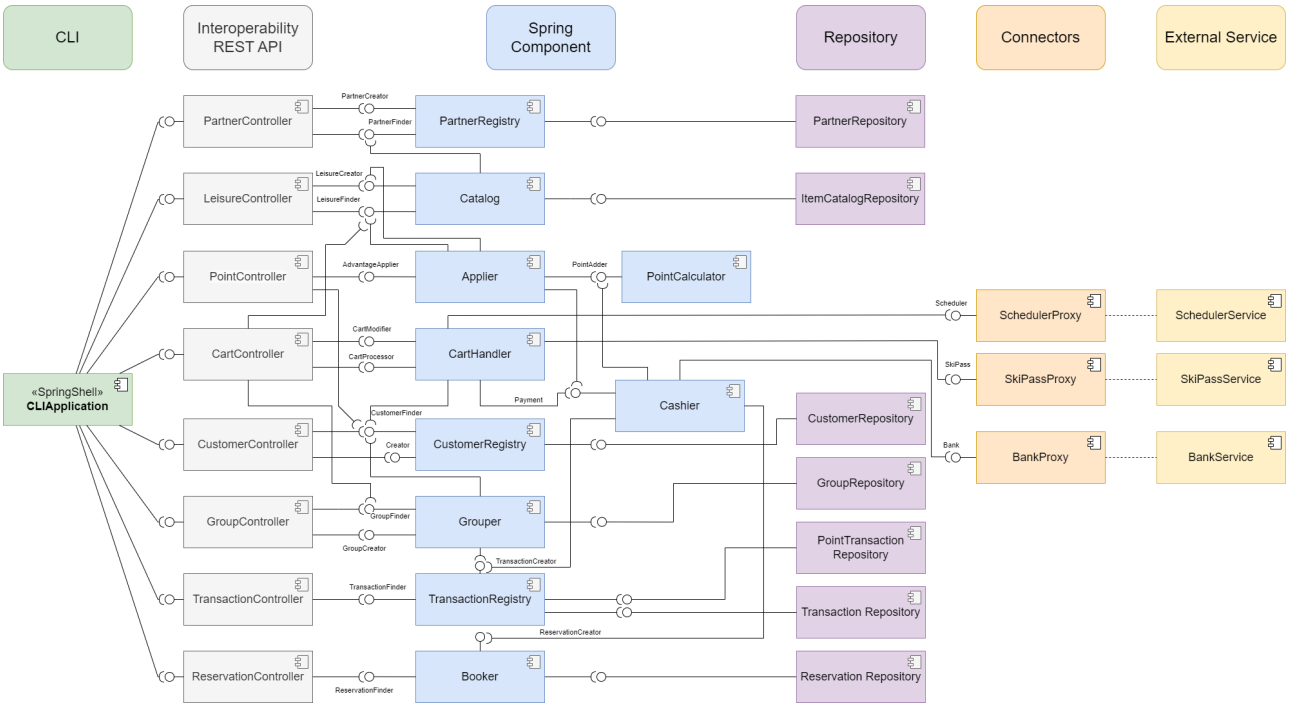
2023/2024

Table des matières

Architecture

Nous avons implémenté un système qui gère les différentes fonctionnalités du sujet selon ce que nous avions défini dans notre rapport d'architecture en y apportant quelques modifications. Ce système communique avec trois services externes, la **Bank** pour les paiements, le **Scheduler** pour le planning des activités ainsi que le **SkiPass** pour gérer les forfaits de ski.

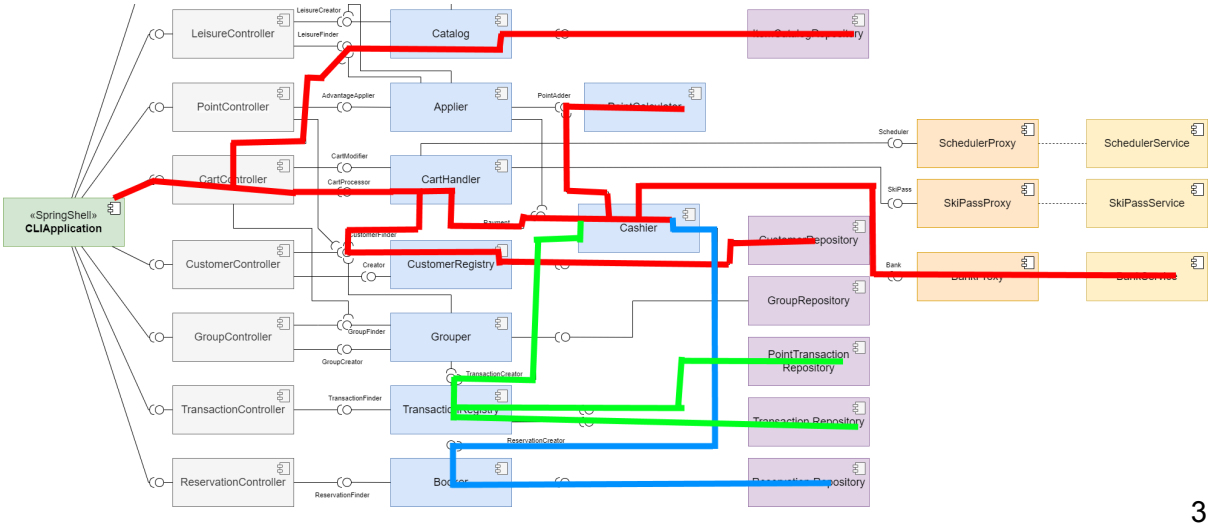
Diagramme de composants



Fonctionnalités réalisées

Les deux composants **PartnerRegistry** et **CustomerRegistry** sont présents pour faire du CRUD pour les partenaires de la station et les clients du système.
En qui concerne les autres composants, nous allons expliquer les fonctionnements et l'intérêt de les avoir à travers les différents scénarios.

Réservation d'une activité



Nous avons fait le choix qu'un Customer possède un panier contenant pour ce qu'il souhaite acheter sous forme item, un service, une activité selon un type précis de réservation, un avantage. Lorsqu'un Customer veut réserver une activité, elle doit déjà être dans son panier. Pour valider une activité plusieurs choix selon le type de réservation, choisi :

- Un appel à un service externe qui s'occupe lui-même de vérifier la validité
- Vérifier en amont qu'un group existe si la réservation est pour un groupe

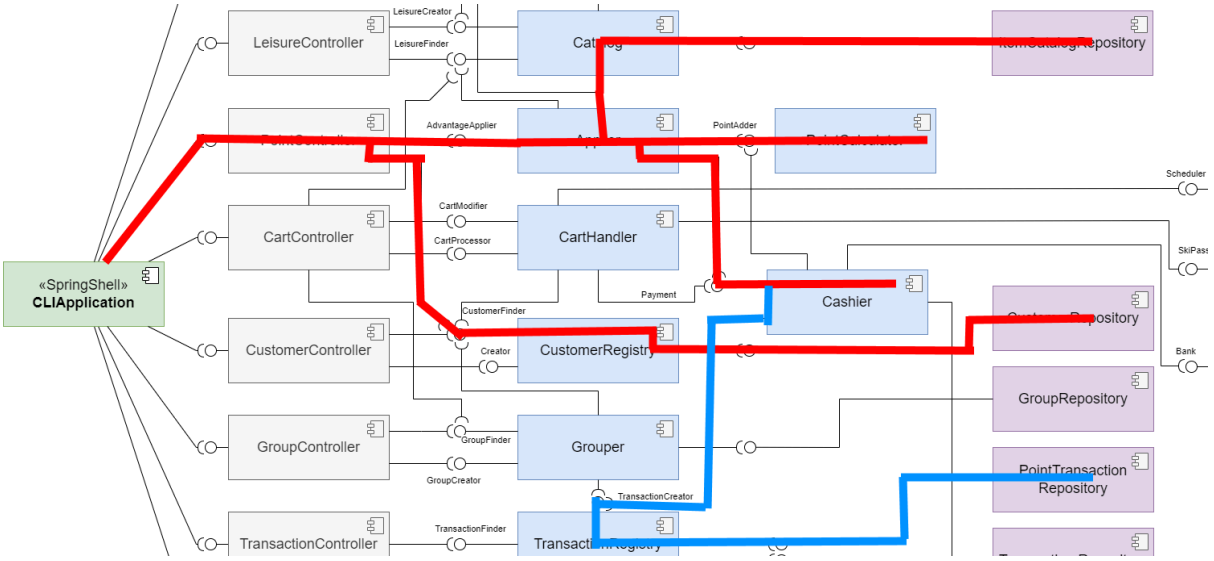
Il nous a semblé cohérent de créer un composant pour gérer le panier du Customer, "**CartHandler**" qui fera l'appel aux différents services. Ainsi lors du paiement de l'activité celle-ci sera déjà conforme sans vérification supplémentaire. Pour gérer le panier deux interfaces :

- **CartModifier** pour mettre à jour les éléments du panier, (ajout et modification pour chaque type item)
- **CartProcessor** pour la validation de l'activité selon son type et

Ainsi nous gérons le remplissage et la validation séparément.
Lors de la validation de l'activité, nous procédons au paiement de l'élément du panier. Pour ce faire nous avons choisi de déléguer cela à une autre composant "**Cashier**" grâce à l'interface **Payment** qui fait la liaison entre les deux.
Le **Cashier** a pour responsabilité l'appel à la **Bank** pour le paiement, et la création de la réservation voulue "**ReservationCreator**" ainsi qu'une transaction correspondant à cette réservation "**TransactionCreator**", et l'ajout de point selon le montant "**PointAdder**"
La création de transaction nous permet d'avoir un historique de toutes ce qui a été effectué lors d'un paiement. Et ainsi avoir quelques simples statistiques.
Nous avons délégué la gestion d'ajout de point à un autre composant **PointCalculator**, pour éviter d'avoir des composants "dieu" responsables de plusieurs fonctions. Ainsi **PointCalulator** fait uniquement des conversions de montant en point et inversement grâce à l'interface **PointAdder**.
Les transactions et réservations sont enregistrées dans leur repository respectif, via deux autres composants : **Booker** et **TransactionRegistry**.

L'intérêt de **Booker** permet de déléguer la recherche **ReservationFinder** et la création des réservations **ReservationCreator** dans le repository. Il en est de même pour **TransactionRegistry** avec **TransactionFinder** et **TransactionCreator**, les transactions sont soit des transactions de points soit celles faite avec de l'argent.

Utilisation d'un avantage

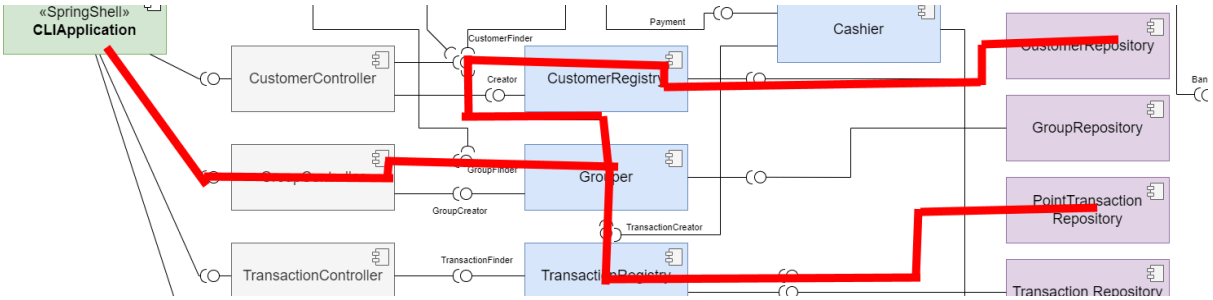


Nous avons pris la décision que pour utiliser des avantages, il fallait qu'il soit dans le panier du Customer, afin de pouvoir avoir une base pour la vérification. Ainsi pour mettre un avantage dans son panier il faut avoir assez de points pour l'ajouter et qu'il existe dans le **Catalog,** composant qui faire la gestion de la création des différents objets du Catalog à savoir Activité, Service et Avantage avec leur interface de création correspondant symboliser dans le diagramme par **LeisureCreator**, ainsi que la recherche avec leur Finder respectif appelé **LeisureFinder.**
Une fois la vérification effectuée. Nous avons fait le choix d'implémenter deux manières différentes d'utilisation. En effet chaque avantage a un type (COCKTAIL, VIP, LOCAL_SPECIALITY, REDUCTION), si le type est une réduction alors, un Customer peut avoir une réduction sur une activité ou service présent dans son panier. Si ce n'est pas de type réduction alors, le Customer en utilisant son avantage va créer un nouvel item de service dont le prix est nul puisqu'il a été payé par l'utilisation de l'avantage et donc des points du Customer dans son panier.

La réduction dépend du prix en point de l'avantage, grâce à **PointAdder** de **PointCalculator**. L'idée de créer un nouvel item de service correspondant à l'avantage utilisé est de pouvoir matérialiser cette utilisation de manière concrète. Pour utiliser un service et ainsi l'enlever du panier, nous avons fait l'hypothèse que pour l'utiliser le Customer fait une action physique en badgeant sa carte auprès du partenaire qui détient ce service.

La gestion de l'utilisation des avantages se fait avec un composant nommé **Applier**, toujours dans l'optique de déléguer les responsabilités. Et le **Cashier** n'est pas surchargé par trop de responsabilités.

Échange de points dans un groupe



Pour échanger des points, nous avons utilisé un composant responsable que tout ce que concerne les groupes **Grouper**. En effet ce composant permet de créer des groupes avec l'interface **GroupCreator** en fonction des Customer déjà enregistrés dans le système. Les groupes sont ensuite enregistrés dans son repository. Et grâce à l'interface **GroupFinder** nous pouvons rechercher des groupes en fonction du leader du groupe. En plus de **Grouper** vérifie si les Customer sont dans le même groupe afin de pouvoir échanger des points. Dans le cas où le donneur de point et le receveur sont dans le même groupe alors les points donnés et deux transactions de points sont créées, une pour chaque Customer. Toujours pour avoir un historique

Fonctionnalités non réalisées

Statistiques

Comme nous l'avons évoqué plus haut, nous avons une sorte de statistique avec les différentes transactions créées en fonction de ce que font les Customers. Mais nous aurions voulu implémenter également les statistiques sous une autre forme.
Lors d'un des cours, la notion de programmation orientée aspect (POA) a été introduite. Dans ce contexte, elle concernait uniquement le système de journalisation. L'avantage de la POA pour la journalisation est que nous n'avons pas besoin d'écrire des appels de fonction à chaque fois que nous voulons consigner quelque chose. Nous spécifions simplement les signatures de fonctions comme des modèles et la magie de la bibliothèque génère automatiquement le code pour journaliser les appels. Notre idée innovante était d'utiliser ce même mécanisme de POA pour mettre en œuvre le système de statistiques. Comme la collecte de statistiques se produit dans de nombreuses classes et à travers de nombreuses méthodes, nous voulions éviter d'écrire les mêmes appels de méthode dans chaque classe et méthode.

Il existe plusieurs couches où les statistiques basées sur la POA pourraient être mises en œuvre, mais les deux couches les plus pertinentes pour nous sont :

Au niveau du contrôleur : Dans les spécifications du projet, il y a un utilisateur qui a besoin de statistiques sur l'utilisation du panier dans le système. La couche du contrôleur est le niveau qui s'aligne le mieux avec le diagramme des cas d'utilisation et donc avec l'utilisation du panier.

Au niveau du proxy : Les services externes échappent à notre contrôle, il est donc judicieux de les surveiller pour leurs performances et leur disponibilité. Ces couches seraient également possibles, mais il y a des raisons qui les rendent moins prioritaires.

Au niveau du service : Pour que les statistiques à ce niveau soient intéressantes, nous devrions introspectionner les charges utiles des fonctions pour avoir plus d'informations sur les détails des informations transmises entre les composants.

Au niveau du référentiel : Le niveau du référentiel n'est pas très intéressant car nous pourrions nous connecter directement à la base de données pour recueillir des statistiques.

Ainsi, pour mettre en œuvre le système de statistiques à l'aide de la POA dans notre projet, nous créerions une nouvelle classe appelée StatisticsLogger.java, similaire à la classe ControllerLogger.java du projet de l'usine de cookies fourni. Cette classe serait responsable de la journalisation des statistiques du système. Le logger aurait un singleton qui instancierait un objet statistique global. Étant donné que les statistiques sont enregistrées à haute fréquence et que nous n'avons pas besoin de valeurs exactes, nous prévoyons de conserver les données en mémoire et de vider périodiquement les données importantes à long terme dans une base de données, en utilisant potentiellement une approche de fenêtre mobile.

La classe StatisticsLogger.java aurait les responsabilités suivantes :

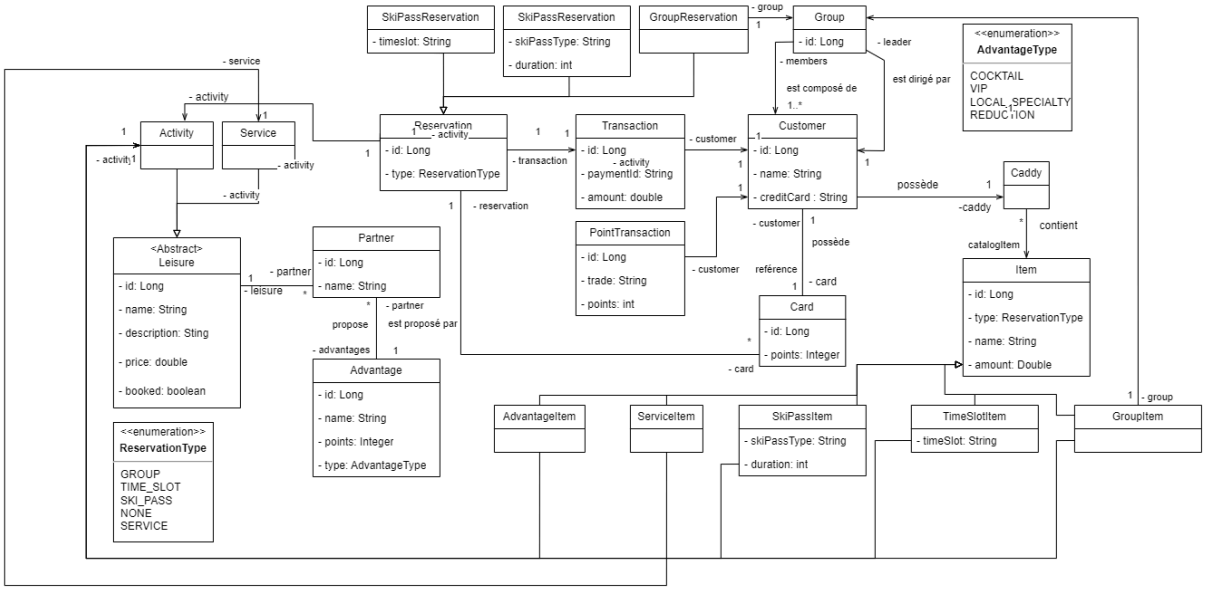
1. Maintenir un objet statistique global qui suit diverses métriques, telles que le nombre de transactions par client, les revenus totaux par client, la valeur moyenne des transactions par client et toute autre métrique pertinente basée sur les exigences.
2. Fournir des méthodes pour mettre à jour ces statistiques chaque fois qu'un événement pertinent se produit, comme la création d'une nouvelle transaction. Ces méthodes seraient appelées à l'aide de la POA, sans avoir besoin de les ajouter manuellement dans chaque classe et méthode.
3. Mettre en œuvre un mécanisme de vidage périodique pour persister les données statistiques importantes à long terme dans une base de données, en utilisant un thread d'arrière-plan ou une tâche planifiée.
4. Fournir potentiellement un moyen d'interroger l'état actuel des statistiques, soit via une API dédiée, soit en exposant directement l'objet statistique.

L'avantage clé de l'utilisation de la POA pour ce système de statistiques est que nous pouvons découpler la logique de collecte des statistiques de la logique métier principale de notre application. Cela rend le code plus maintenable, car nous n'avons pas besoin de nous soucier d'ajouter du code lié aux statistiques à plusieurs endroits. L'aspect POA gérera automatiquement les mises à jour des statistiques chaque fois que les méthodes pertinentes seront appelées.

De plus, en conservant les données statistiques en mémoire et en ne les vidant périodiquement que dans la base de données, nous pouvons améliorer les performances du système, car l'écriture dans la base de données peut être une opération relativement lente. L'approche de la fenêtre mobile peut nous aider à conserver uniquement les données à long terme les plus pertinentes, tout en éliminant les informations plus anciennes et moins utiles.

Dans l'ensemble, ce système de statistiques basé sur la POA devrait fournir un moyen robuste et efficace de collecter et de gérer les statistiques d'utilisation de notre application, sans encombrer la logique métier principale.

Modèle Métier



Item
Elle est hérité par les classes SkiPassItem, TimeSlotItem, GroupItem, ServiceItem et AdvantageItem qui représente chacune un type de produit mis dans le panier. Le but des Item, c'est de pouvoir créer un objet qui diffère des objets présents dans le catalogue, soit les Leisure et Advantage que contient les Partner. Ainsi un item peut être un service, un avantage, une activité en fonction, ce qui fait que chaque activité peut être réservée avec plusieurs types différents sans avoir plusieurs instances de cette activité dans différent type et donc simplifier la base de donnée. Les Item sont éphémères et lorsqu'ils disparaissent cela n'impact pas le catalogue.

Reservation
Reservation est une classe Abstraite qui représente la réservation de l'activité correspondant à un item. La réservation est créée avec la transaction générée par le paiement du panier. Nous avons décidé que pour chaque activité dans le panier il y ait une réservation et une réservation pour tout, car cela permet d'éviter de rechercher une activité particulière dans une liste où elles seraient toutes.

Caddy
Le Caddy fait office de support pour contenir les avantages, services ou activités que le Customer veut acheter qui sont tous des items du panier. C'est pour regrouper tout ce qui peut être acheté au même endroit et ainsi simplifier son utilisation.

Group
Les groupes sont composés à la fois d'un leader et d'un ou plusieurs membres. Le leader n'apparaît pas en tant que membre du groupe. Pour éviter d'avoir modifier l'enregistrement du Customer leader pour le désigner leader avec un booléen. Il vous a paru plus compréhensible de le faire de cette manière, pour éviter de parcourir tous les membres pour trouver le leader et d'avoir deux références d'un même Customer.

Choix d'implémentation de la persistance

Nous avons fait plusieurs choix en termes de persistance en s'appuyant sur différents concepts.

Transaction

Pour les méthodes permettant de rechercher dans les repositories les différents Entity voulues, nous avons utilisé un @Transactional(readOnly = true), ainsi cela permet d'être sûr de ne pas modifier les Entity en les parcourant. Lors du paiement du panier du **Customer** que ce soit pour une activité, un service ou un avantage, nous voulons avoir la certitude que ces paiements soient dans une transaction c'est pourquoi nous les avons mis en @Transactional(propagation = Propagation.*MANDATORY*)

Cascading

En termes de cascading nous avons utilisé un CascadeType.*ALL*, pour les **Customer**, **Card**, **Partner** et le **Caddy**.
Nous avons besoin d'avoir la création et la suppression d'une **Card** lorsqu'un **Customer** est créé ou supprimé, cela ne sert à rien d'avoir une **Card** sans Customer. Il en est de même pour la **Card**, sans **Card** les réservations seules ne sont pas utiles. Lorsque l'un **Partner** est supprimé tous les services, avantages et activités, n'ont plus de propriétaire et donc cela n'est pas utile de les garder. De même pour le **Caddy** et des items à l'intérieur.

Lazy loading

Nous avons eu des problèmes de chargement des données avec un FetchType.*EAGER*, nous chargions beaucoup de données inutiles lors de la création d'un panier avec les items à l'intérieur, qui eux aussi chargé des données inutiles. Ce qui menait à une erreur de Read time out, lors de l'appel au niveau CLI. Pour ce faire nous avons utilisé au niveau des différents items dans le **Card**, un FetchType.*LAZY*, ainsi cela a réduit le chargement de données inutiles en amont et de les charger uniquement lorsque c'est nécessaire.

Inheritance

Pour représenter les **Activity** et **Service**, nous avons pris le parti de dire qu'une activité et un service sont identiques sauf sur un point, une activité est réservable et un service non. Partant de ce postulat, nous avons créé une classe abstraite **Leisure** dont **Activity** et **Service** en hérite avec un InheritanceType.*SINGLE_TABLE*. Comme il s'agit de la même classe à un true/false différent, avoir une seule table pour deux Entity représente une bonne solution En ayant deux Entity différentes cela évite qu'à chaque fois qu'un **Leisure** est utilisé une vérification sur le fait qu'il soit réservable ou non, et cela complexifierait initialement les requêtes.

En ce qui concerne les items, nous avons utilisé InheritanceType.*TABLE_PER_CLASS* pour l'Entity **Item** car chaque classe qui en hérite (**ServiceItem**, **AdvantageItem**, **TimeSlotItem**, **GroupItem** et **SkiPassItem**) et diffère beaucoup les unes des autres, et donc avoir une classe par Entity permet éviter d'avoir une trop grande table où la plupart des colonnes de cette table sera vide. De plus avec notre implémentation l'impact des recherches sur la classe mère est limitée puisque nous n'avons pas d'héritage profond

Prise de recul

Forces

Pour ce qui est de notre architecture nous sommes plutôt fiers de son organisation, la quantité de composants nous permet un bon découpage des responsabilités métiers, permettant ainsi d'éviter l'apparition de composants avec trop de responsabilité.

L'atout principale de notre architecture reste le système de réservation et de paiement , en effet grâce à l'utilisation de services externes, cela réduit considérablement les dépendances entre les composants de notre architecture, et nous permet de nous dispenser d'une implémentation complexe d'un système d'emploi du temps par exemple.

Nous avons réalisé l'intégralité des fonctionnalités demandées, sauf des statistiques performantes.

Le système avantages permet d'avoir un fonctionnement simple de leur utilisation, où les activités et les avantages ne sont reliés que par les partenaires.

Les Enum pour les types de réservations et avantages facilitent l'utilisation des avantages et la gestion des réservations.

Faiblesses

Nous avons une gestion très mince des erreurs retournées par le backend dans la CLI. Par exemple, lorsque nous avons une erreur pour la création d'une Entity qui existe déjà, nous avons juste un code de retour sans pour autant expliquer l'erreur. Lorsqu'il y a des exceptions qui sont lancées pour un identifiant non trouvé ou autre. La CLI retourne une erreur 500 pour certaines routes. Et il faut aller voir dans les logs du backend qu'il s'agit bien d'une bonne erreur voulue.

L'affichage dans la console de la CLI des éléments, n'est pas très clair. Les DTOs sont faits avec les mêmes attributs que leur Entity, donc lorsqu'il y a beaucoup d'éléments à afficher, ça peut être moins lisible.

Une fois qu'une réservation est créée et enregistrée, il n'est pas possible de la modifier.

Les interfaces **CustomerFinder** et **LeisureFinder** sont utilisées par plusieurs composants différents.

Capacité d'évolution

À l'heure actuelle, nous avons une seule station avec un seul club, c'est pour cela que nous n'avons pas fait CRUD pour le club ou la station, que nous avions défini dans l'ancien rapport d'architecture. Si maintenant, le système est adapté à plusieurs stations et à plusieurs clubs, il faudra revoir le système de catalog avec les activités, services, et avantages. Savoir si la carte de fidélité d'un customer est valable dans les mêmes clubs mais de stations différentes. Revoir la gestion de la base de données, parce que pour le moment, certaines Entity n'acceptent pas de duplication au niveau du nom.

Si à l'avenir, il y a une précision sur ce qu'est une activité et un service qui diffère beaucoup, il faudra revoir la conception au niveau des **Leisure**. Par contre rajouter de nouveaux types de réservations ou avantages, il n'y aura pas de problème de changement d'architecture.

Répartition des points

Membre de l'équipe	Points
Arnaud DUMANOIS	106
Marie ORFILA	140
Clément REMY	77
Nikan ZANDIAN	77