CASTEXSKI

Rapport Final
ISA / DevOps

Membres de l'équipe F :

João BRILHANTE

Armand FARGEON

Ryana KARAKI

Ludovic MARTI

Valentin ROCCELLI

UNIVERSITÉ CÔTE D'AZUR | POLYTECH NICE-SOPHIA

I. Table des matières

I. Introduction

CastexSki est une application permettant la gestion de plusieurs stations de ski. Son catalogue propose différents forfaits et cartes disposant d'accès personnalisés aux remontées mécaniques de ses stations partenaires. De plus, elle permet la fidélisation de ses clients en offrant de nombreuses réductions avant et après achat.

Avec notre contexte sanitaire, *CastexSki* dispose aussi d'un dispositif de supervision analysant des indicateurs tels que la quantité de visiteurs d'une station à chaque instant, le nombre de ventes ou encore l'affluence sur une remontée à double portique.

Finalement, des notifications informant des conditions météorologiques, des statistiques d'affluence ou de réductions spontanées pourront être envoyées aux abonnés ou commerçants locaux par mail ou téléphone.

II. Présentation du travail effectué

1. ISA : Fonctionnalités de notre application

Notre application couvre un maximum des fonctionnalités demandées. Un client peut consulter les différents forfaits et cartes qui existent. Après avoir effectué son achat, s'il commande une carte, le client devra la récupérer dans une station de ski. Il peut également effectuer ses achats directement dans une station.
En caisse, un(e) employé(e) sera chargé(e) de lier la carte, via son numéro unique, au compte du client. Ceci permet à ce dernier de directement y ajouter un forfait depuis son compte lorsqu'il effectue un achat en ligne.

Sur les pistes, le skieur aura accès aux remontées qui correspondent à son forfait. S'il dispose d'une *SuperCartex*, toutes les remontées lui seront accessibles et à chaque badgeage, il sera débité sur sa carte bancaire comme pour un télépéage.
Différents types de réductions existent :

- Pour les cartes normales, il y a des réductions sur des forfaits d'un mois, ainsi que des réductions pour des jours particuliers, activées par les employés
- Chaque forfait propose une version "enfant" à prix réduit
- Pour les *SuperCartex*, la première heure de ski est gratuite, ainsi que le huitième jour de ski consécutif

Un employé d'une station de ski peut ajouter, modifier ou supprimer des forfaits sur le site. Il pourra également consulter les statistiques d'achat des clients, afin d'adapter la proposition de réductions. L'employé sera également chargé, après avoir créé un forfait, d'indiquer à quelles remontées il permet d'accéder.

Lorsqu'une nouvelle station est ajoutée à l'application, on peut lui attribuer différentes remontées mécaniques, pistes de ski, ainsi que des panneaux d'affichage.

Les pistes de ski sont accessibles via des remontées. Chaque badgeage est comptabilisé afin de générer, à la fin de la journée, un rapport d'affluence : il peut être consulté par les employés de la station, et est envoyé par mail aux commerçants qui le souhaitent, ainsi qu'à la préfecture correspondante. De ce fait, ces derniers pourront adapter leur stratégie tout au long de l'année. De plus, les responsables de la station pourront également arrêter les ventes de forfaits en ligne pour la journée lorsqu'ils constatent qu'il y a un très grand nombre de personnes sur place. Ces statistiques sont consultables en temps réel par les employés.

Certaines remontées de ski, populaires, disposent d'un système de double badgeage : ainsi, on peut déterminer le temps d'attente moyen dans la queue et voir rapidement s'il devient trop important. De plus, le badgeage d'une carte avec un forfait "enfant" est détectable afin de déclencher un signal sonore pour avertir le perchiste.

Afin d'informer les utilisateurs de promotions ou de chutes de neige, nous avons implémenté un système de notifications faisant appel à différents services externes :

- un service mail afin d'envoyer les rapports de statistiques aux inscrits
- un service SMS pour les avertir de promotions ou de chutes de neige
- un système de gestion des panneaux d'affichage, afin de faire parvenir différentes informations aux skieurs (indications sanitaires, piste fermée…)
- un système permettant de récupérer la météo d'une certaine ville
- une API bancaire, pour les achats en ligne

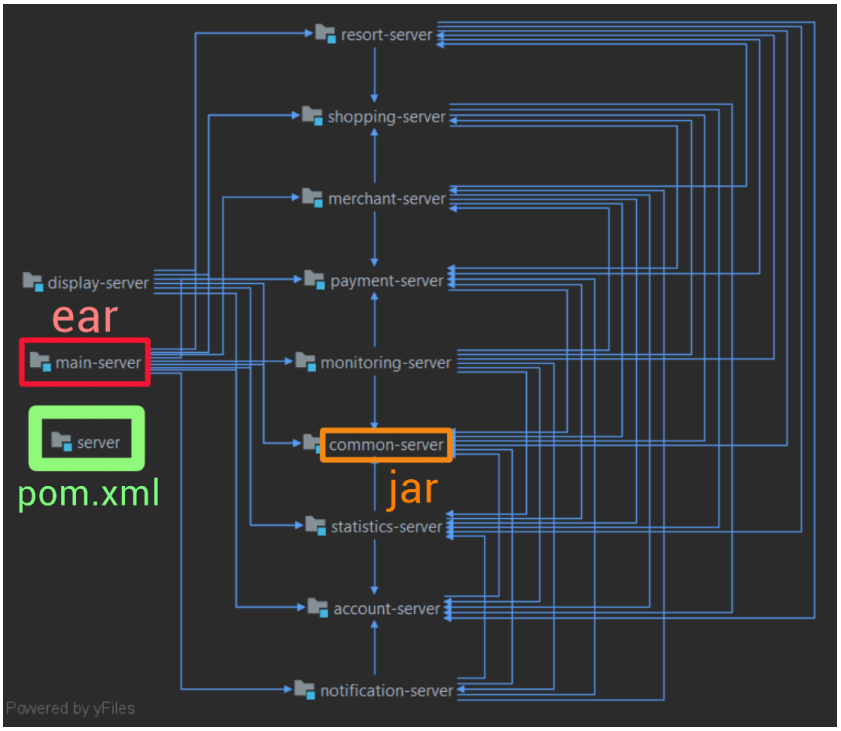
Pour la partie client, l'interface en ligne de commande que nous avons réalisée permet une gestion complète de l'application, de la création de nouvelles stations à celle de notifications automatisées en passant par une gestion des clients et du catalogue. Avec cette dernière, nous avons créé un espace simple d'utilisation et adaptable aux utilisateurs en permettant de se connecter à un compte client ou en tant qu'employé de la station pour disposer de droits et commandes différents.

2. DevOps

a. Structure du projet et mise en place des outils de déploiement

Notre projet est structuré en différents modules. A la racine de notre projet se trouvent trois sous-modules GitHub :

- *client*, représentant notre interface client
- *external*, qui regroupe tous les sous-modules de nos services externes
- *server*, qui contient tous les sous-modules de notre partie serveur et permettant sa compilation



Chaque sous-module de notre serveur possède son propre fichier *pom.xml* (packaging EJB).
Le *main-server* est présent uniquement pour générer un *ear* qui sera ensuite exécuté par TomEE.
Le *pom.xml* situé à la racine de *server*, contient les définitions des modules et permet la compilation en local du serveur.

Chacun de nos dépôts GitHub contient, au moins, un fichier *Jenkinsfile* qui correspond à une tâche sur Jenkins, que nous avons installé sur la machine virtuelle fournie. Ce fichier nous permet de spécifier ce que Jenkins doit utiliser (logiciels, variables d'environnement,...) et les différentes étapes à effectuer lors de ces tâches. Ces étapes nous permettent de bien identifier celles qui peuvent potentiellement échouer. On a mis en place pour l'ensemble des modules plusieurs sécurités pour empêcher qu'une tâche échoue pour une autre raison qu'un code incorrect. Parmi ces mesures de sécurités, nous adressons par exemple plusieurs *ping* aux services externes utilisés lors des tests d'intégrations, et nous exécutons les tests d'intégration seulement si ces services sont bien joignables.

Une dernière étape, que beaucoup de nos sous-modules ont, est le déploiement sur notre *Artifactory*. En effet, en plus des dépôts GitHub permettant la gestion des versions de notre code, nous avons un dépôt permettant une gestion des versions de nos binaires. Nos binaires, une fois compilés et vérifiés (avec des tests unitaires et d'intégration) par Jenkins, sont téléversés sur notre *Artifactory* et sont, au besoin, téléchargés lors des compilations et des tests exécutés localement par les développeurs. Ceci nous évite

d'avoir à télécharger le code de chaque module, de les garder à jour et de les recompiler, dans le cas où on travaille sur un module qui dépend d'un autre.

Ainsi, avoir plusieurs étapes sur le *Jenkinsfile* permet de déployer uniquement un binaire qui compile, et permet à chaque développeur d'avoir une version stable du projet.
Lorsqu'un des modules a été modifié et que son déploiement a réussi, la pipeline de *main-server* (module permettant la création de l'EAR) va se déclencher. Il va donc être recompilé avec les derniers modules de l'*Artifactory*, et si la tâche réussit, l'EAR construit sera déployé.

La phase finale va être le déclenchement de la pipeline Docker du *main-server*. L'EAR précédemment déployé va être compilé, puis une nouvelle image Docker va être construite et le conteneur en cours d'exécution sera suspendu. La pool d'images locales (espace limité) va être nettoyée puis l'image sera déployée sur notre dépôt Docker Hub. Enfin, la dernière étape de la tâche va lancer un conteneur avec la dernière image en exposant les ports requis sur le même réseau que *Jenkins*. En exécutant ainsi les conteneurs Docker, il nous est possible de lancer des tests d'intégration de nos différents modules du serveur sur *Jenkins*.

Ce processus est le même pour notre client, ainsi que nos services externes, disposant chacun d'une pipeline pour une compilation puis l'appel de la pipeline pour la création et le déploiement de l'image Docker.
Le déploiement sur Docker Hub va nous permettre par la suite de récupérer des images sans avoir à cloner, compiler et déployer à chaque fois, ce qui s'avère pratique avec un docker-compose qui se chargera d'aller simplement récupérer les images sur internet.

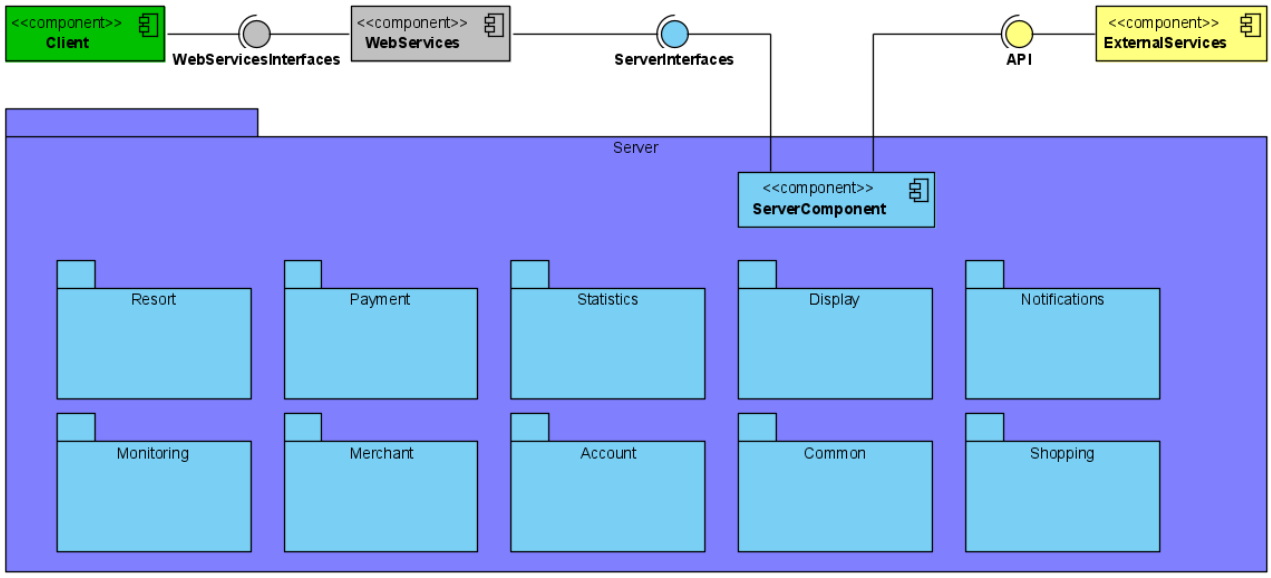
b. Tests de bout-en-bout

Nous avons également effectué depuis le Client de nombreux tests Cucumber, afin de bien représenter l'utilisation de son interface en ligne de commande. Ces tests traversent la totalité de notre application, partant des commandes utilisateurs jusqu'aux services externes. De tels tests ont également étés écrits dans notre couche application lorsque c'était nécessaire et utile pour pouvoir vérifier des comportements suivant un scénario.

III. Diagramme de Composants[1]

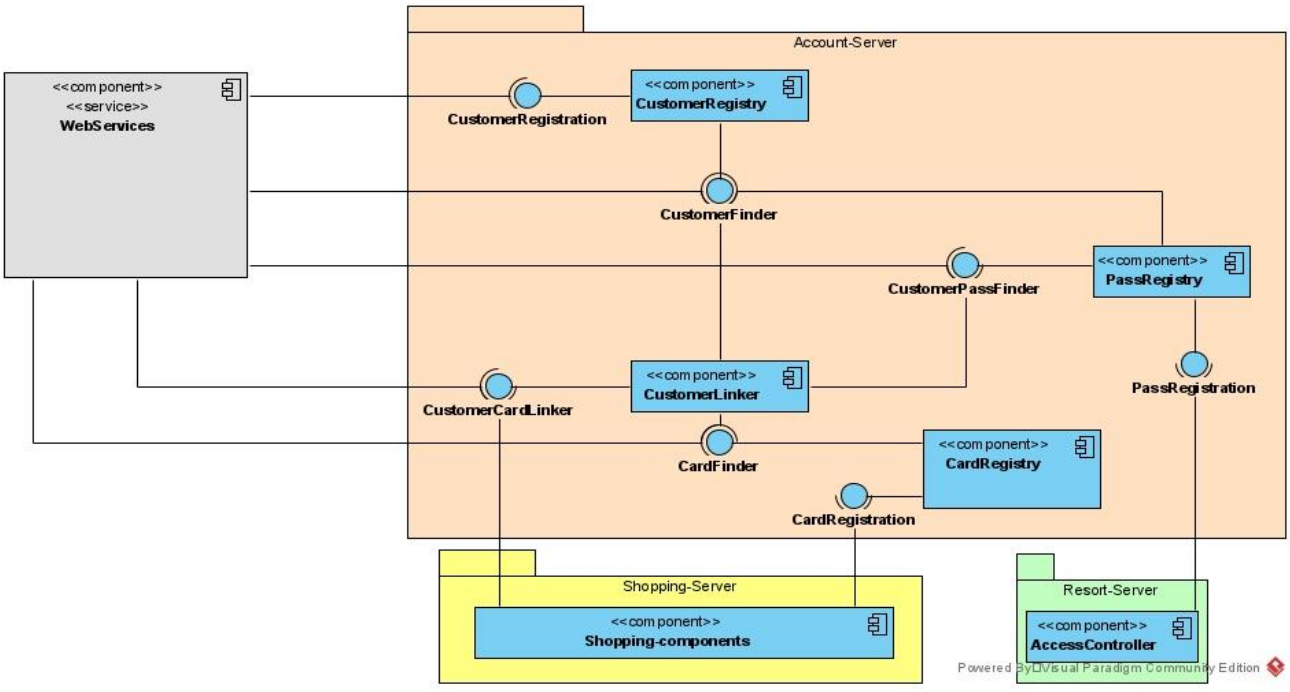
Notre projet respecte une architecture 3-tiers dans laquelle nous avons développé une partie client et une partie serveur, en utilisant la base de données gérée par JPA. Pour la partie serveur, nous avons découpé notre projet en différents modules :

Common-server contient toutes les entités du projet. Étant donné que certaines entités



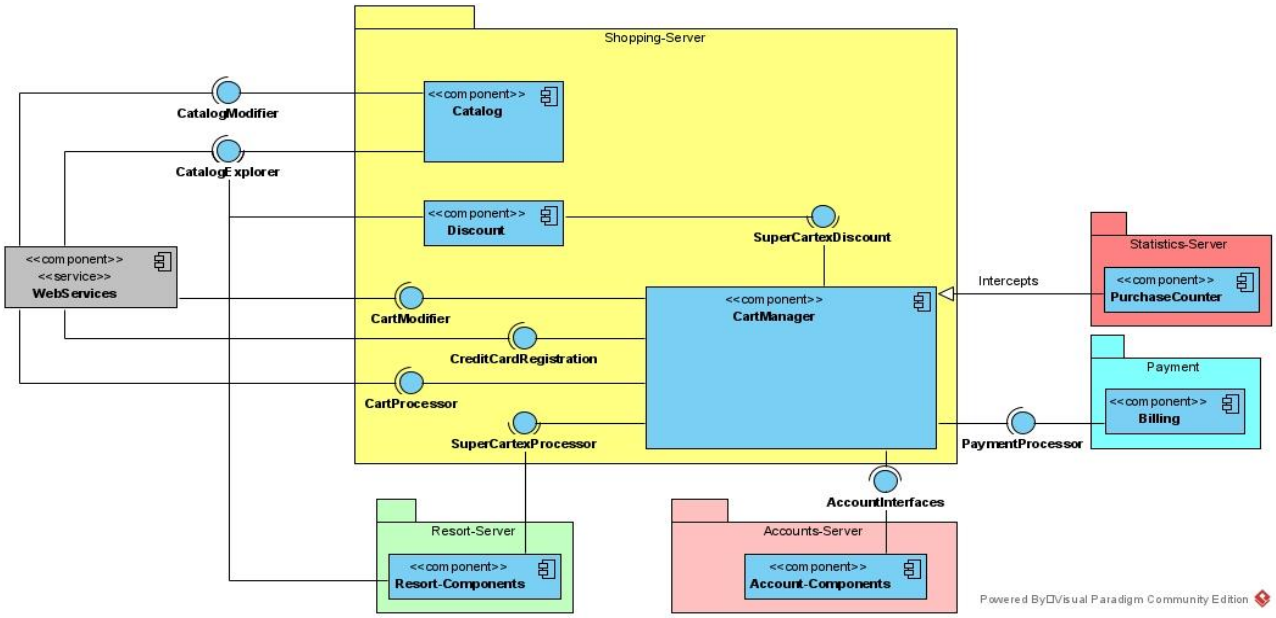
sont utilisées par plusieurs autres modules, nous avons fait le choix de toutes les regrouper au même endroit afin d'éviter les dépendances cycliques.

Account-server permet la gestion des comptes utilisateurs, ainsi que l'association de cartes et de forfaits à un compte :

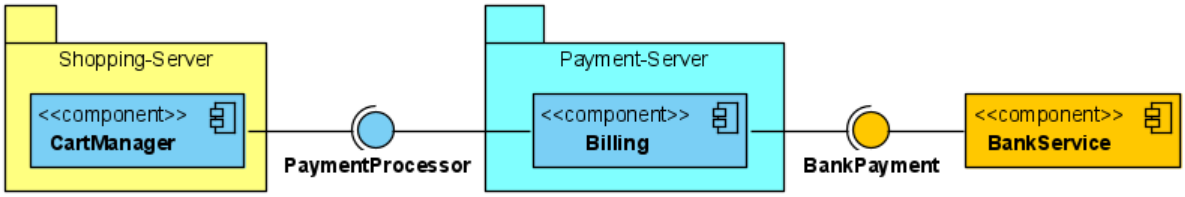


[1] Lien vers les diagrammes zoomés et le diagramme général

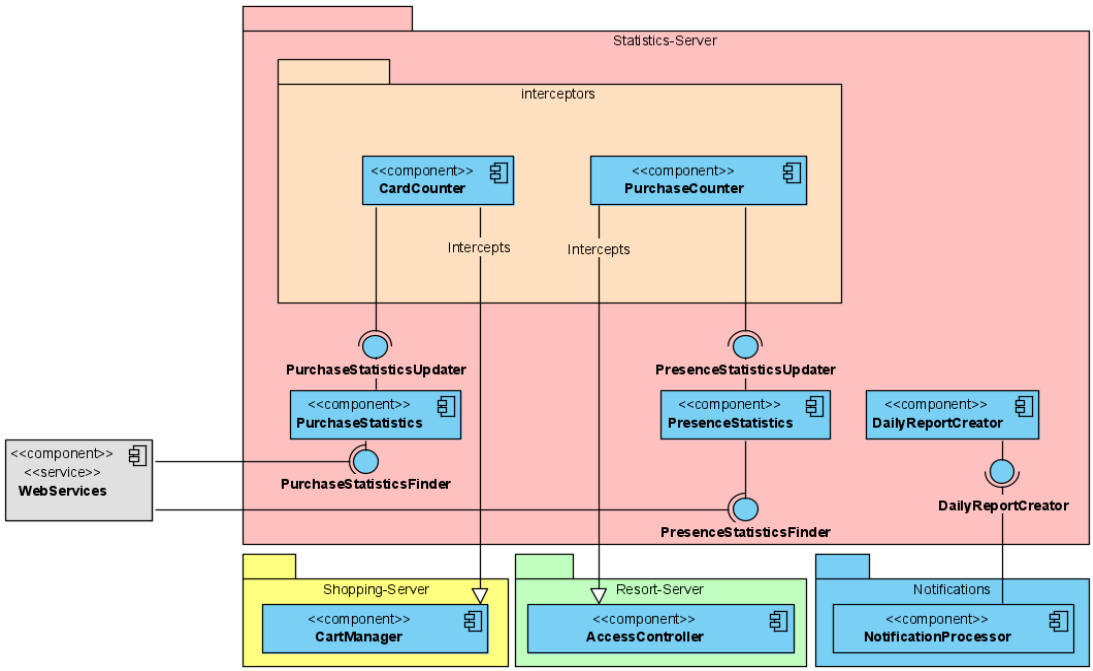
Shopping-server regroupe la partie achat du projet, avec le catalogue d'achat, le panier et les réductions :



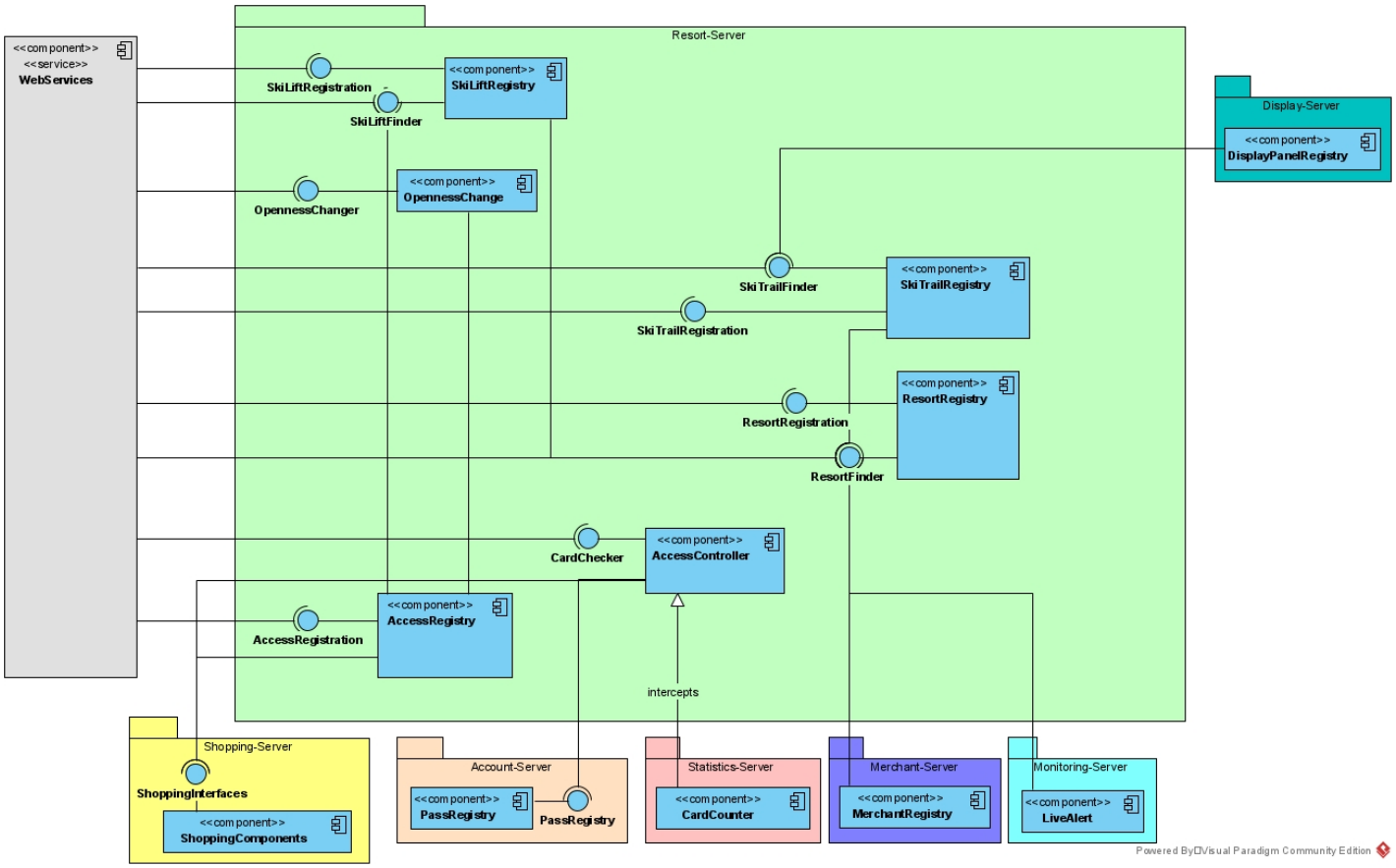
Payment-server communique avec l'API externe de la banque afin de lui déléguer la gestion du paiement, en lui donnant toutes les informations nécessaires :



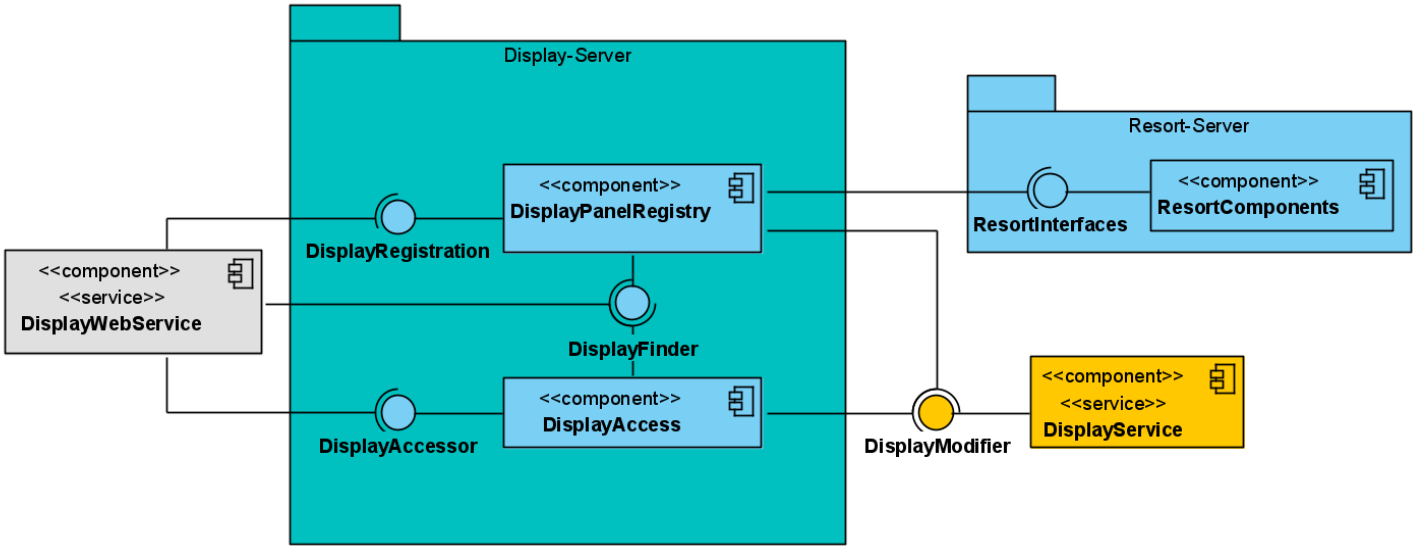
Statistics-server génère deux types de statistiques : achat et présence. Grâce à des intercepteurs, au moment d'un achat ou d'un badgeage de carte, les statistiques sont mises à jour :



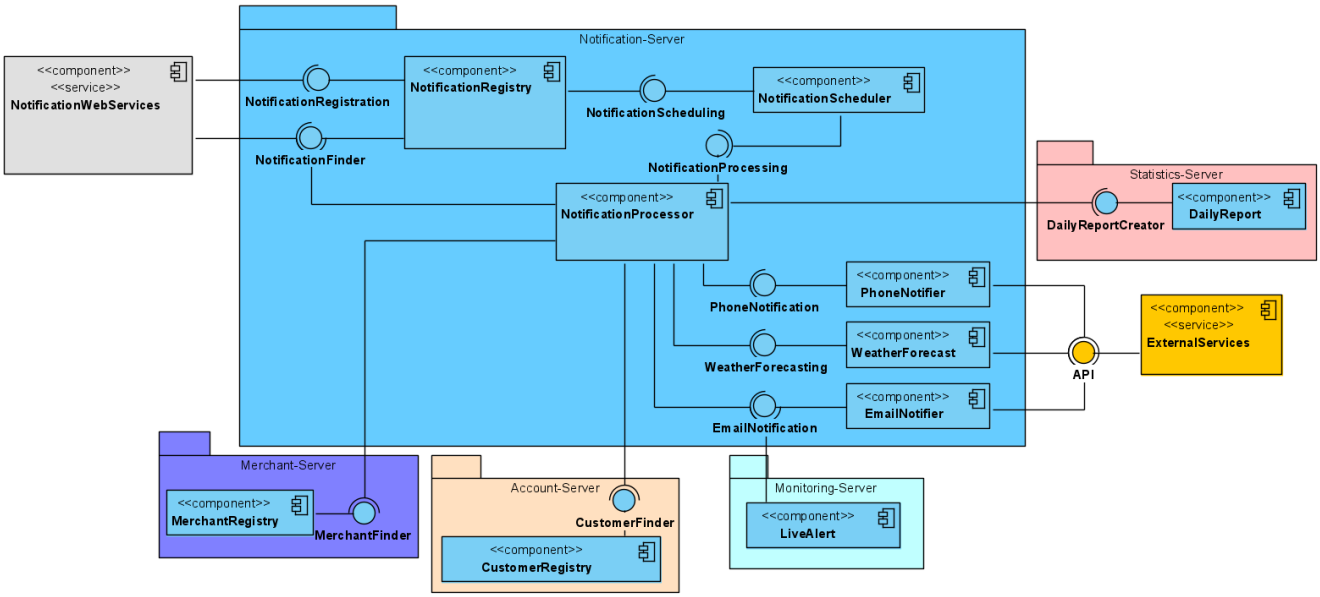
Resort-server permet la gestion d'une station de ski, avec les remontées et les pistes associées. La vérification d'accès s'y trouve également :



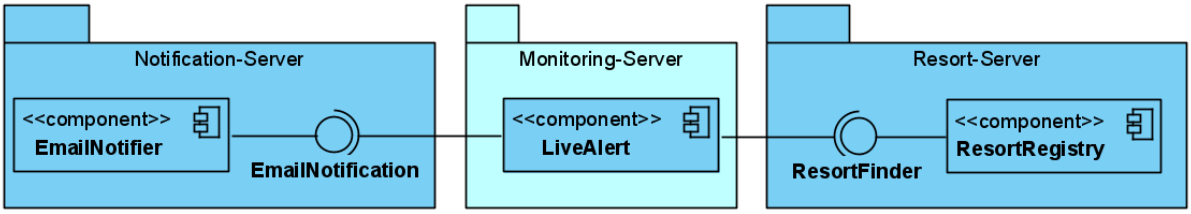
Display-server permet d'envoyer des informations au service externe de panneaux d'affichage :



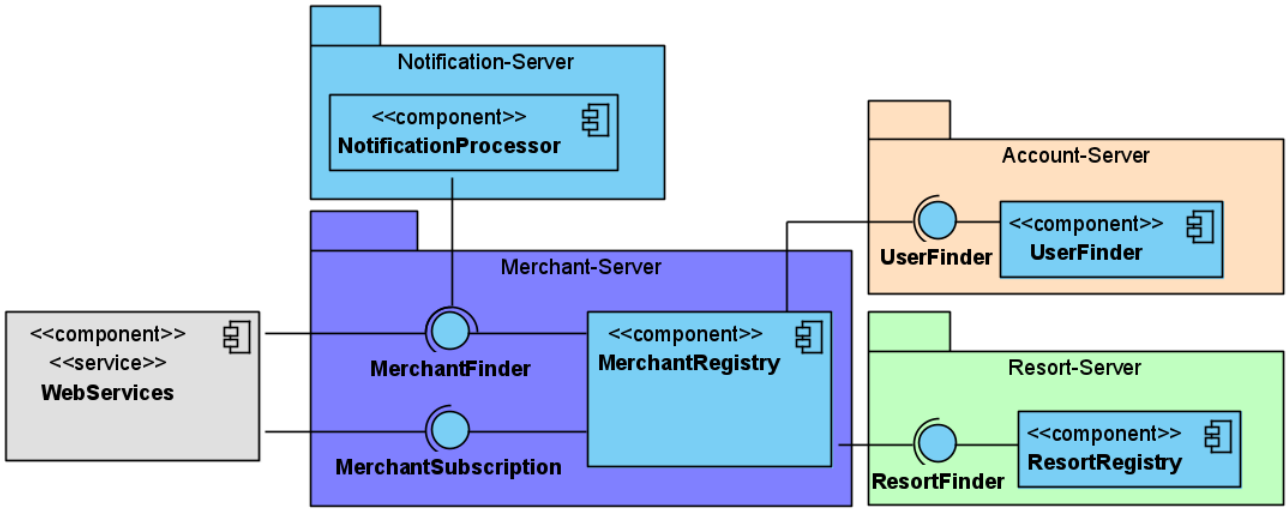
Notification-Server regroupe les différents types de notification pour un utilisateur. Il y a une newsletter pour les commerçants, et des alertes promotionnelles ou météorologiques pour les skieurs :



Monitoring-Server permet de superviser en direct l'état de la station et d'entreprendre des actions particulières lorsque nécessaire, de façon automatique. Il orchestre notamment la gestion particulière du système de double portique : il va générer des alertes en fonction du nombre de personnes en train d'attendre, afin que les contrôleurs puissent rediriger les usagers en cas d'attente trop longue :



Merchant-Server permet de retrouver facilement l'ensemble des commerçants souhaitant suivre de près les derniers chiffres de fréquentation d'une station de ski :



IV. Interfaces

1. Module Account-Server

Les cartes de ski sont retrouvées, ajoutées et activées grâce aux interfaces : *CardFinder* et *CardRegistration*. Afin d'associer un compte utilisateur à une carte, l'interface *CustomerCardLinker* le permet grâce à l'identifiant physique de la carte. Grâce à *CustomerFinder*, *CustomerRegistration*, et *CustomerUpdater*, on peut ainsi rechercher un client selon différents critères, par mail ou par nom par exemple.

On peut également enregistrer un nouveau client et mettre à jour un compte. Les forfaits étant liés à un compte utilisateur, l'interface *PassRegistration* permet d'ajouter un forfait à un client, mais également d'activer un forfait. La recherche d'un forfait s'effectue grâce à l'interface *CustomerPassFinder* : on peut ainsi retrouver l'ensemble des forfaits d'un client, un forfait particulier ainsi que les ensembles de forfaits activés ou non.

2. Module Display-Server

Dans le but de diffuser des messages informatifs aux usagers de la station, ce module communique avec le service externe grâce à l'interface *DisplayAccessor*. Elle permet d'accéder aux informations d'un panneau particulier et de modifier le message affiché. Les interfaces *DisplayFinder* et *DisplayRegister* permettent d'ajouter de nouveaux panneaux dans le système et de les retrouver.

3. Module Payment-Server

L'interface *PaymentProcessor* permet de recueillir et d'envoyer toutes les informations nécessaires au service externe de la banque, afin d'effectuer un achat.

4. Module Resort-Server

Une nouvelle station de ski peut être ajoutée grâce à l'interface *ResortRegister*. Elle pourra être ensuite retrouvée au moyen de *ResortFinder*. Les pistes de ski d'une station pourront être ajoutées, modifiées et retrouvées grâce aux interfaces *SkiTrailFinder* et *SkiTrailRegister*.
Les remontées associées aux pistes seront gérées par les interfaces *SkiLiftFinder* et *SkiLiftRegister*. De plus, le contrôle d'accès à une piste sera réalisé grâce à *CardChecker*, qui vérifie l'état d'ouverture d'une piste, le forfait de la carte scannée, sa validité et s'il s'agit d'un forfait enfant.
Si c'est une *SuperCartex* dont le forfait n'est plus valide, il fera appel à l'interface *SuperCartexProcessor* du module *shopping-server*, qui se chargera de le recharger.
L'état de la station, ainsi que l'état des remontées ou des pistes pourra être changé entre "ouvert" et "fermé" grâce à *OpennessChanger*. Ce changement d'état empêche l'accès aux pistes ou l'accès aux ventes de forfait, si la station ferme. *AccessRegister* permet d'associer un forfait à une remontée, permettant d'autoriser les possesseurs de ce forfait d'accéder à la remontée correspondante.

5. Module Shopping-Server

Un catalogue de produits, présentant les différents forfaits et cartes est mis à disposition des clients : c'est le but de *CatalogExplorer*. *CatalogModifier* quant à lui permet aux employés d'ajuster le catalogue avec de nouveaux produits, ou modifier les produits existants. Une entrée de catalogue peut être une carte ou un forfait et l'employé peut décider de la rendre privé, utile pour l'ajout de réduction sur la super cartex ou par exemple de remises nécessitant une vérification de justificatif en caisse.
La *SuperCartex* étant associée à une carte bancaire, elle fonctionne comme un télépéage, et le débit lors d'un badgeage sera géré par le *SuperCartexProcessor*. Possédant également des réductions spécifiques, l'interface *SuperCartexDiscount* est chargée de les appliquer à la *SuperCartex* en fonction des critères définis. Cette dernière va en effet vérifier quel forfait est le plus adapté au possesseur de la SuperCartex. Elle vérifiera entre autres si c'est sa première heure de ski, si il existe une quelconque réduction sur le jour ou le mois courant ou encore si c'est son huitième jour consécutif de ski.
Pour effectuer des achats en ligne, un utilisateur dispose d'un panier qui sera modifié au moyen de l'interface *CartModifier*, et le paiement d'un panier sera coordonné par *CartProcessor*. L'interface *CreditCardRegistration* permet à un utilisateur d'ajouter ou de retirer les informations de sa carte bancaire.

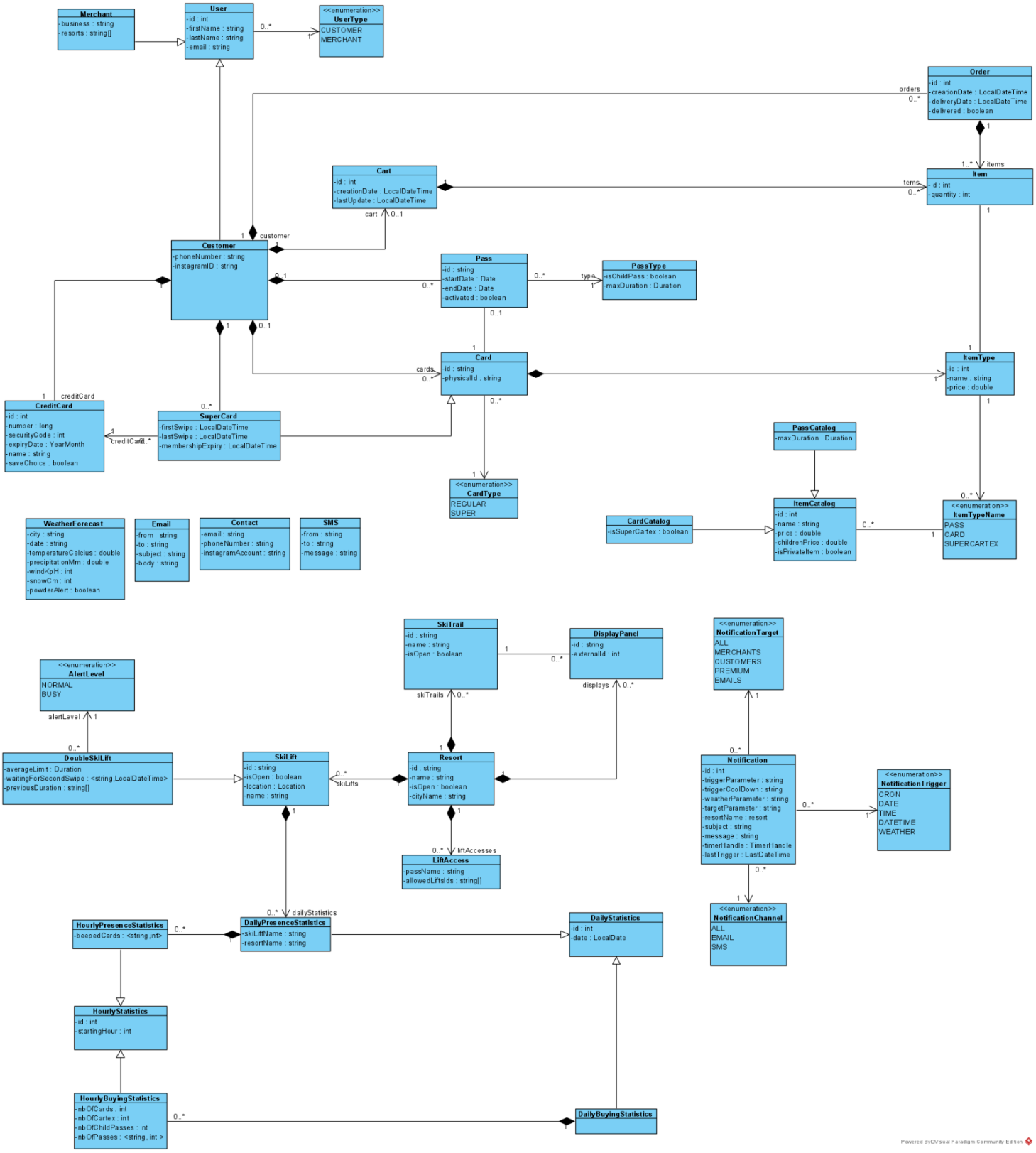
6. Module Statistics-Server

Les statistiques sont divisées en deux types : les statistiques de présence, spécifique à une station de ski, et les statistiques d'achats, communes à toutes les stations de ski. Dès qu'une carte est badgée, l'intercepteur *CardCounter* met les statistiques à jour au moyen des interfaces *PresenceStatisticsFinder*, *PresenceStatisticsRegistration* et *PresenceStatisticsUpdater*. Dans le cas où la carte est badgée sur une remontée mécanique équipée d'un système de double badgeage, l'intercepteur va se charger de mettre à jour le temps moyen de badge entre les deux portillons. De la même manière, à chaque fois qu'un panier est payé, l'intercepteur *PurchaseCounter* met les statistiques d'achat à jour grâce à *PurchaseStatisticsFinder*, *PurchaseStatisticsRegistration*, et *PurchaseStatisticsUpdater*. Le rapport journalier à envoyer par mail est construit grâce à *DailyReportCreator*.

7. Module Notification-Server

Les interfaces *NotificationFinder*, *NotificationProcessing*, et *NotificationRegistration* permettent la gestion générale des différents types de notifications. Leur fréquence d'envoi est programmée grâce à *NotificationScheduling.*
Les interfaces *PhoneNotification*, *EmailNotification*, *WeatherForecasting* servent d'intermédiaires entre le système et les API externes. On pourra ainsi envoyer des SMS et Email personnalisés, et récupérer les informations de la station météo d'une ville.

V. Diagramme de classes [2]



[2] Lien vers le diagramme en meilleure qualité

VI. Choix d'implémentation

1. Persistance

Grâce aux notions de *Spring* acquises lors du PS7, nous avons incorporé la persistance dès le début du projet. Ainsi, nous n'avons jamais eu besoin d'un *bean stateful,* qui contiendrait de l'information et dont les attributs seraient modifiés par des échanges avec ce dernier. Nous avons une unité de persistance située dans *Common-server* et donc partagée par l'ensemble des modules. Nous avons choisi pour l'ensemble de nos entités de les déclarer avec l'annotation @Entity, indépendantes les unes des autres avec un identifiant généré automatiquement par JPA. Ceci nous permet d'avoir des objets plus facilement maintenables qu'avec par exemple certains objets qui auraient pu être mappés par les annotations @Embedded ou @Embeddable. Par exemple, la carte de crédit est actuellement utilisée par un *Customer* mais nous pourrions à l'avenir avoir besoin de les persister pour retrouver un client ou utiliser directement la fonction sans contact pour ouvrir les portillons.
Nous avons clairement défini le cycle de vie des objets grâce à leurs paramètres de cascade. Pour revenir à l'exemple de la carte de crédit, lorsqu'un client est supprimé du système, sa carte l'est également. Par contre, supprimer une remontée ne détruit pas la station qu'elle contient. Durant nos tests unitaires, nous avons toujours pris soin de tester de façon poussée les composants se chargeant de persister et de modifier nos objets.
Nous avons également dû adapter nos attributs ne pouvant être persistés par JPA. C'est le cas des types liés à la durée dans Java (comme *LocalDate* ou *LocalDateTime*) qui ne sont pas directement persistables. Il nous fallait donc annoter le type non persistable avec @Transient et persister son type au format *string*. Ensuite, au moment où la classe était chargée, une fonction @PostLoad permettait d'initialiser l'objet et donc de pouvoir utiliser leurs fonctions très utiles, nous évitant de les réécrire.

2. Types de composants

Notre projet utilise uniquement des composants *stateless*. Nous avons fait ce choix car ils permettent un passage à l'échelle plus simple : les stations peuvent se multiplier dans toute la France et le nombre de clients peut devenir très important. Ce passage à l'échelle est facilité par JPA qui s'occupe de l'attribution et de l'allocation des *beans* : avoir des *beans stateless* nous permet d'avoir des *beans* interchangeables, que l'on peut créer et détruire à volonté.

3. Intercepteurs

Nous avons choisi d'utiliser des intercepteurs au niveau de deux aspects du projet :

- le double badgeage, lorsqu'une remontée populaire est équipée d'un système de double portique
- les statistiques d'affluence et d'achat

En fonction des données d'affluence ou d'achat, différentes interfaces s'occupent d'ajouter en base de données les statistiques qui correspondent. Elles sont utilisées par nos deux intercepteurs : *CardCounter* et *PurchaseCounter*, qui interceptent les bonnes méthodes dans les modules *resort-server* et *shopping-server*. En effet :

- *resort-server* possède l'interface *AccessController*, qui vérifie lors d'un badgeage si une personne a le droit de passer ou non
- *shopping-server* possède l'interface *CartManager*, qui permet la gestion et le paiement du panier d'un client. Les statistiques seront mises à jour une fois que le client aura effectué son achat.

Intercepter ces méthodes permet de récupérer facilement le contexte nécessaire à l'ajout de statistiques, mais offre également l'avantage de ne pas rajouter du code dans les méthodes déjà implémentées des modules *resort-server* et *shopping-server*. De plus, mettre à jour les statistiques ne correspond pas au rôle de ces deux modules.
Pour le double badgeage, les intercepteurs ont permis de l'implémenter de façon plus tardive sans avoir à modifier les méthodes de *resort-server*, venant très bien s'intégrer à l'architecture existante et pouvant réutiliser l'intercepteur implémenté *CardCounter*.

4. Détails supplémentaires

Dans le souci de rendre l'application modulable pour le futur, nous avons créé la classe-mère *User*, de laquelle héritent les classes *Merchant* et *Customer*. Les *Merchant* peuvent posséder des locaux dans différentes stations, et recevront ainsi un rapport journalier pour chacune de ces stations. D'autres types d'utilisateurs pourront être rajoutés en héritant de *User*, comme par exemple un *Cashier* ou un *Controller*.

Concernant l'implémentation des statistiques, nous les avons implémentés de la manière suivante : un objet *DailyStatistics* représentant les statistiques journalières, contient plusieurs *HourlyStatistics*, représentant les informations par heure. En effet, un commerçant est intéressé par le détail horaire : lui fournir le total de personnes sur une journée n'est pas pertinent. Cette représentation permet de simplifier la représentation des statistiques : *DailyStatistics* ne se retrouve pas surchargé de différentes informations horaires, et s'occupe simplement de ses différents *HourlyStatistics*, qui sont créés et mis à jour dès qu'une nouvelle information est enregistrée.

Avant d'introduire toutes les entités dans le *Common-Server*, nous avons envisagé de mettre les entités dans le module qui leur correspondait le mieux : par exemple *Customer* dans *Account-Server*, et *SkiLift* dans *Resort-Server*. Cependant, ceci risque d'introduire des dépendances cycliques alors qu'on cherche juste à utiliser un objet depuis un autre module ! Par exemple : imaginons que l'entité *Item* (super-objet représentant les différents types d'achats) soit placée dans le module *Shopping-Server* qui lui correspond. Comme le module *Statistics-Server* utilise *Item* pour mettre à jour les statistiques d'achat, ce module devra dépendre de *Shopping-Server* afin de pouvoir le

récupérer. Or, l'intercepteur *PurchaseCounter* se trouve dans le module *Statistics-Server*, donc *Shopping-Server* a déjà une dépendance envers lui... C'est pour éviter ces situations délicates que nous avons choisi de mettre l'ensemble des entités dans un seul module, qui n'aurait pas d'autre rôle que de les contenir.

Les entités liées à l'achat représentent de nombreuses classes et peuvent sembler compliquées au premier abord. C'est pourtant un mécanisme auquel nous avons bien réfléchi et qui n'a pratiquement pas changé depuis notre premier diagramme de classe au début du projet, mélangeant plusieurs concepts comme le polymorphisme et l'utilisation du patron de conception *factory*. Les objets *ItemCatalog* vont permettre de spécifier le nom, le prix adulte et enfant, pour une carte, si c'est une *SuperCartex*, ou une durée pour un forfait. Ainsi, lorsqu'un utilisateur souhaite acheter un forfait, un *ItemType* sera construit grâce à *l'ItemCatalog* correspondant mais aussi à la précision apportée par l'utilisateur sur le forfait (adulte ou enfant), permettant d'obtenir le bon prix. Cet *ItemType* sera encapsulé dans un objet *Item* pour apporter la quantité du même objet, et c'est cet objet final qui sera ajouté au panier.

VII. Rétrospective et conclusion

1. Composants Orientés Messages

Les composants orientés messages se montrent utiles en cas de charge élevée ou à l'exécution de longues opérations : en effet, on peut leur attribuer une tâche et les laisser s'en charger, sans avoir à attendre la fin de l'exécution. Bien que nous n'ayons actuellement pas utilisé de composants orientés message dans notre projet, nous avons des pistes pour une éventuelle implémentation future, principalement au niveau des notifications.
Actuellement, les commerçants sont inscrits à un rapport journalier qu'ils reçoivent par mail, avec les statistiques d'affluence de la journée.
Cependant :

- si le nombre de commerçants devient très élevé,
- si dans le futur on doit générer différents rapports journaliers, adaptés aux destinataires, toutes ces tâches vont prendre beaucoup plus de temps et ralentir le système. Ainsi, on pourra attribuer ces tâches à un composant orienté message, qui s'en occupera de son côté tandis que le système continue ses propres tâches.

De plus, pour les notifications promotionnelles ou d'alerte météo à destination des skieurs, le problème de ralentissement se présente également. En effet, le nombre de clients à notifier peut devenir très important, mais on peut également supposer que les clients voudront choisir d'autres biais de communication en fonction de leurs préférences : aux SMS et à Instagram pourront s'ajouter Facebook, Twitter… Ainsi, ce processus pourra également être implémenté par des composants orientés messages.

2. Critique de notre implémentation actuelle

Notre objet *SkiLift* possède actuellement une liste de statistiques journalières qui doit donc être réinitialisée chaque jour pour éviter un trop grand nombre d'objets dans le système. Une amélioration possible pourrait être de sauvegarder les statistiques dans un attribut qui serait sauvegardé dans la base de données chaque jour et réinitialisé pour le suivant. Ou sinon, chaque statistique journalière pourrait simplement connaître un identifiant du *SkiLift* associé afin de le retrouver et d'éviter l'imbrication.

Le service de météo est implémenté et fonctionnel dans l'application. Cependant, par manque de temps, il n'est pas possible de modifier la météo depuis le client. En effet, ce service récupère l'information météorologique dans un fichier *json* qui doit donc être modifié en temps réel pour pouvoir constater un changement et interagir avec les notifications.

Le module *account-server* a pris beaucoup d'ampleur : il permet la gestion des clients, de leurs cartes, de leurs forfaits mais également la gestion et l'inscription des commerçants. Nous avons entamé la simplification de ce module en créant *merchant-server*, qui s'occupe de la gestion des commerçants de la station. Cependant, ce découpage n'est pas suffisant. Il faut encore simplifier *account-server* en séparant dans de nouveaux modules les différents types d'utilisateurs, ou selon les grosses responsabilités : pour les comptes clients, la gestion des cartes clients, les employés…

On constate le même défaut pour le module *resort-server*. Il faudrait effectivement mieux distribuer ses responsabilités en le séparant en trois parties :

- un module qui s'occupe des autorisations d'accès, rôle actuel de l'interface *CardChecker*. On peut également lui attribuer la gestion des accès (*AccessRegister*)
- un module pour la gestion générale et simple d'une station de ski, avec les remontées et les pistes
- un pour l'état d'ouverture d'une station, rôle actuel d *'OpennessChanger*. Ainsi, à la fermeture d'une station, ce module se chargera d'effectuer toutes les actions nécessaires, comme bloquer les ventes en cas de fermeture longue ou changer les informations des panneaux d'affichage. En effet, ces actions devraient être automatiques et ne devraient pas nécessiter d'intervention de la part d'un administrateur. Actuellement, le changement d'affichage des panneaux se fait en ligne de commande depuis le client.

Finalement, un dernier point concernant notre travail a été le manque de temps par rapport aux objectifs fixés. En effet, nous nous sommes peut-être surestimés en fixant des objectifs un peu trop ambitieux sur la fin du projet, et nous n'avons pas réussi à atteindre tous nos buts. Malgré ce point, nous sommes assez fiers du travail fourni et de notre équipe, qui a su avancer et franchir tous les obstacles.