Rapport final ISA-DevOps

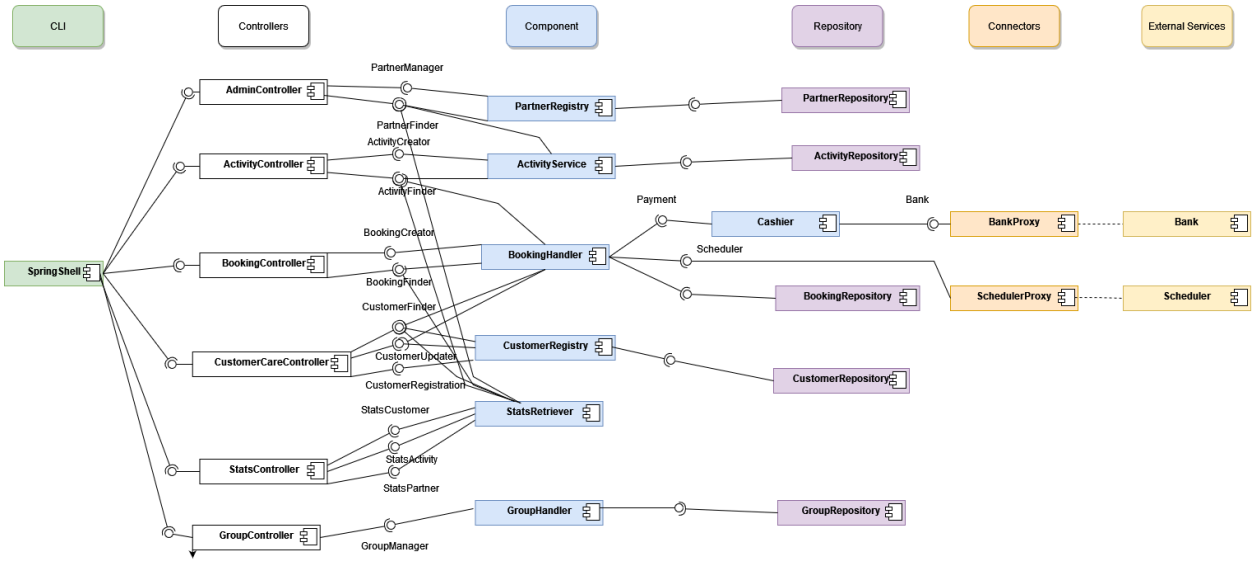
Groupe G

FROMENT Lorenzo
MAUROIS Quentin
STENGEL Damien
BEUREL Simon

Tables des matières :

1. Architecture de composants

Pour pouvoir réaliser correctement le projet Winter4Everybody, nous avons réalisé une architecture composée de 6 controllers, 7 composants qui implémentent différentes interfaces. Nous avons également 2 services externes à l'application implémentés en NestJS, qui sont la Bank et le Scheduler qui seront détaillés plus tard dans le rapport.



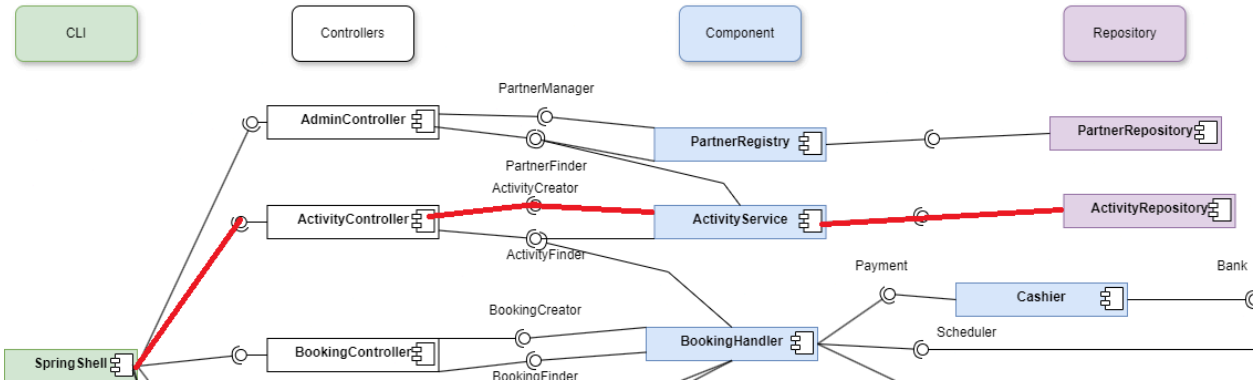
Pour pouvoir mieux expliquer notre diagramme de composants, nous allons étudier à travers différents points la liste des spécifications qui étaient demandées dans le projet original, et nous allons voir comment nous avons implémenté ces spécificités.

A. Spécificités liées aux partenaires

Les partenaires jouent un rôle très important dans l'application Winter4Everybody, car ce sont eux qui vont notamment pouvoir créer les différentes activités que les clients vont pouvoir réserver pour ainsi gagner des points pour les extras par exemple.

a. Ajouter une nouvelle activité

Une des actions prioritaire pour un partenaire, est de pouvoir créer une activité et ainsi de pouvoir l'intégrer à la liste des activités proposées sur l'application Winter4Everybody. Pour se faire, voici le chemin parcouru à travers notre application quand un partenaire souhaite ajouter une nouvelle activité :



Comme vous pouvez le constater, pour réaliser cette demande, nous allons tout d'abord traverser le controller *ActivityController* qui sera celui qui récupérera de manière générale toutes les requêtes liées aux activités. Une fois dans ce controller, nous allons ensuite nous rendre dans le composant *ActivityService* qui est le composant qui est donc relié au *PartnerRepository* mais également à *l'ActivityRepository* qui correspondent à la base de données des partenaires mais également à celle des activités respectivement. *L'ActivityService* à notamment besoin d'avoir en attribut le *PartnerRepository* car quand nous allons créer une activité dans le base de données, nous allons lui donner en attribut l'ID correspondant au partenaire qui vient de créer cette activité. Cela nous permettra plus tard notamment de pouvoir réaliser différentes statistiques et donc de pouvoir analyser quels sont les meilleurs partenaires en termes de réservation d'activités. Quand un partenaire décide de créer une activité, il faut qu'il spécifie : le nom de l'activité, le lieu, le nombre de places, le prix en €, le prix en points de réservation, et l'ID du partenaire auquel l'activité sera liée.

Améliorations potentielles : L'un des points qui serait important à améliorer dans le futur est le lien entre Activité-Partenaire, car dans notre implémentation, une activité possède en attribut l'ID du partenaire, mais pas l'inverse, ainsi, si l'on souhaite supprimer un partenaire de la base de données, son activité sera toujours présente dans le catalogue.

b. Consulter les statistiques de réservation

Les statistiques sont des éléments très importants dans l'application du même style de Winter4Everybody car elles permettent de pouvoir analyser dans ce cas ici, quelles activités ne sont pas attrayantes auprès des clients de la station, et ainsi cela permet de

pouvoir réfléchir à des changements potentiels.

Le controller appelé pour obtenir les différentes statistiques de la station est le *StatsController* (que nous reverrons dans la partie liée aux gestionnaires de la station). Ce dernier va être appelé sur la route "`/partner/{partnerId}`", c'est-à-dire que l'on va lui demander de nous retourner différentes statistiques en fonction de l'ID du partenaire dans la base de données. Le controller va donc lui envoyer la demande au composant *StatsRetriever* qui va ensuite dans un premier temps :

- Retrouver les différents informations liées au partenaire comme son nom, sa description, sa location (grâce à *PartnerRegistry*)
- Récupérer les différentes activités liées au partenaire (grâce à *ActivityService*)
- Récupérer toutes les réservations effectuées pour ce partenaire en question

Ainsi, grâce à ces 3 étapes, nous possédons des informations permettant aux différentes propriétaires des établissements partenaires de pouvoir faire un état des lieux de leur entreprise.

Améliorations potentielles : L'une des remarques qui peut-être prise en compte pour un futur refactoring de l'application, est que le composant StatsRetriever possède à demi-mot le rôle de "classe Dieu" car il possède un accès à *CustomerRegistry*, *PartnerRegistry*, *BookingHandler* et *ActivityService*. Il serait donc peut-être utile de réfléchir à une meilleure séparation en différents composants si l'on souhaite pouvoir étendre les statistiques (sauvegarde des statistiques, etc…)

B. Spécificités liées aux gestionnaires de la station (ADMIN)

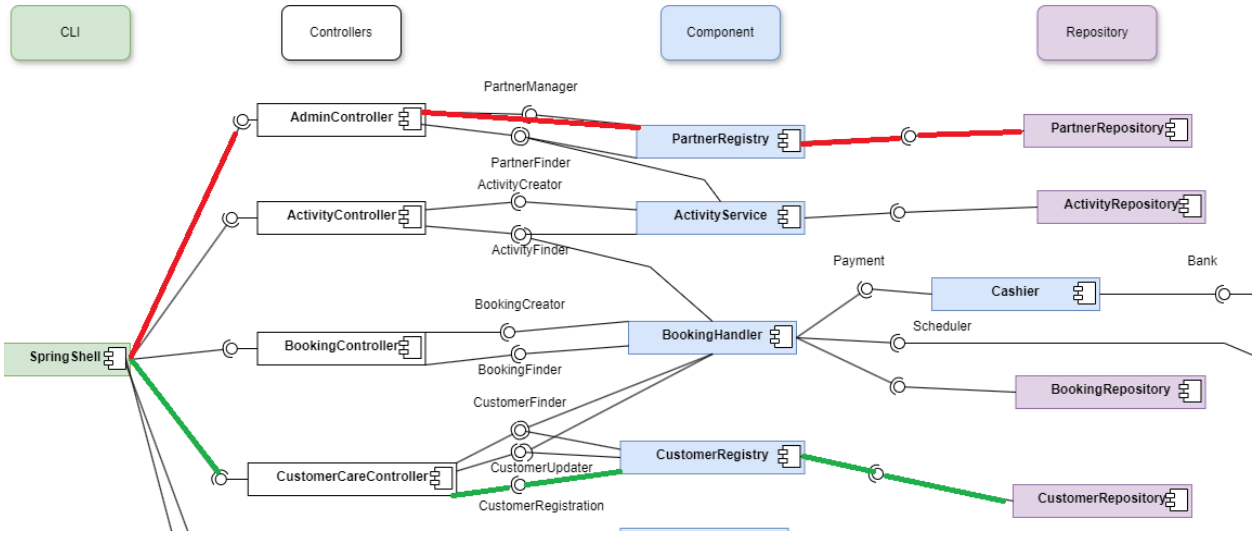
Dans le document montrant les différents acteurs qui utilisent l'application Winter4Everybody, on mentionne les gestionnaires de la station. Dans notre cas, nous allons appeler ces personnes des **ADMINS** car ils ont le rôle d'administrer la station et donc de s'assurer du bon fonctionnement de celle-ci. Pour ces admins, nous avons implémenté ces différents points :

a. Ajouter différents acteurs dans la base de données

L'ajout de différents acteurs dans la base de données est une spécificité essentielle et obligatoire en tant qu'administrateur de l'application. Ici, quand nous parlons de "différents acteurs", nous incluons :

- Les clients
- Les partenaires

Nous n'allons pas passer beaucoup de temps sur cette partie, car elle est assez CRUD, néanmoins vous trouverez ci-dessous en **VERT** le parcours réalisé pour ajouter un client et en **ROUGE** le parcours réalisé pour ajouter un partenaire.



Pour l'ajout de clients, l'admin a seulement besoin de renseigner le nom du client et son numéro de carte de crédit, et pour l'ajout d'un partenaire, l'administrateur doit renseigner le nom du partenaire, sa localisation et sa description.

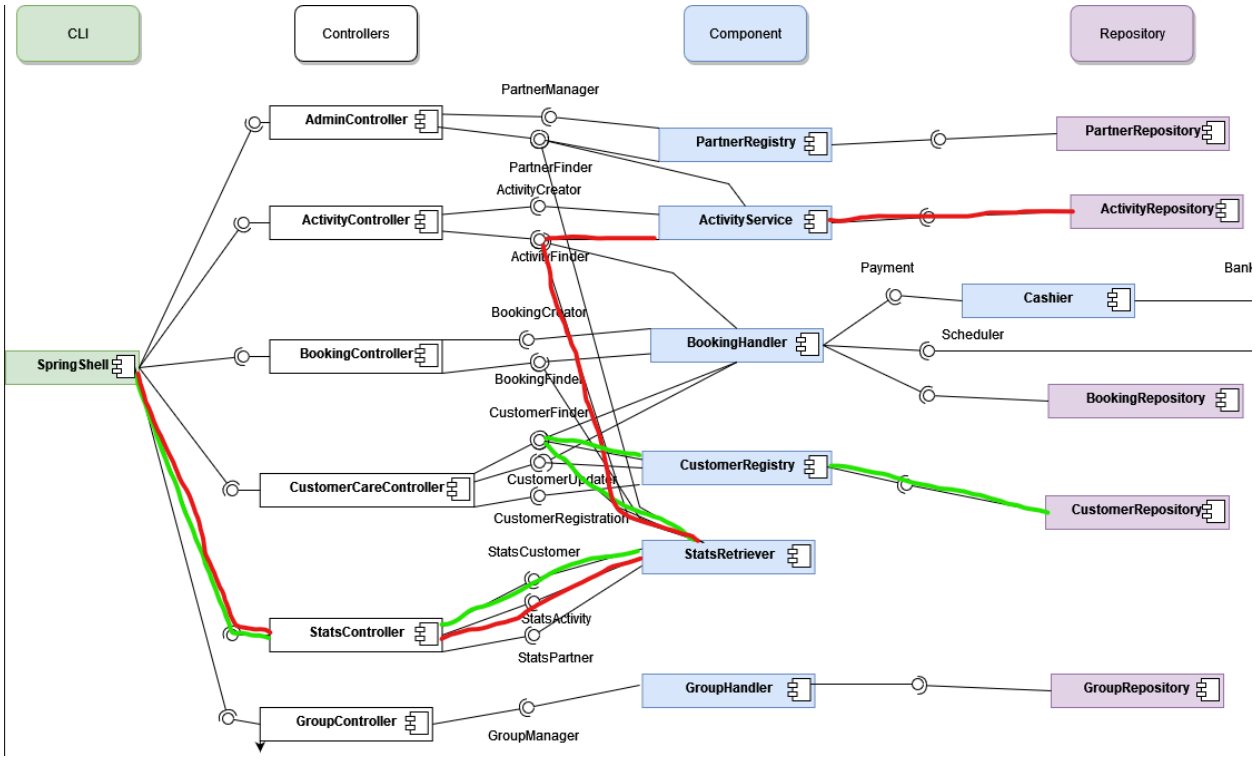
Améliorations potentielles : Étant donné que le processus est CRUD, les seules améliorations potentielles que nous pouvons intégré sont l'ajout de données complémentaires pour améliorer le profil des clients (adresse email, numéro de téléphone etc..) mais ces ajouts sont simples à intégrer dans le code de notre architecture car il suffit simplement de modifier les entités et les paramètres que l'on passe lors de la création d'un clients.

b. Consulter les statistiques de la station

Comme nous l'avons vu précédemment avec les spécificités liées aux partenaires, il est important de pouvoir consulter différentes statistiques pour pouvoir faire une analyse et un état des lieux de la station. Dans le cas des Admin, nous avons décidé de réaliser 2 types de statistiques :

- Les statistiques liées aux clients
- Les statistiques liées aux activités

Vous trouverez ci-dessous en **VERT** le parcours réalisé pour les statistiques liées aux clients et en **ROUGE** le parcours réalisé pour les statistiques liées aux activités. Ces différentes statistiques permettent notamment de pouvoir vérifier l'activité d'un client, ou alors la popularité d'une activité en fonction de son nombre de réservations.



Améliorations potentielles : De manière générale, l'implémentation de toutes ces statistiques (y compris celles liées aux partenaires) ne sont qu'un **MVP,** pour aller plus loin dans l'élaboration de nouvelles statistiques plus poussées ou alors dans l'implémentation d'une sauvegarde des statistiques (pour pouvoir suivre l'évolution via une courbe) il serait nécessaire de réaliser un travail de refactoring sur l'architecture de

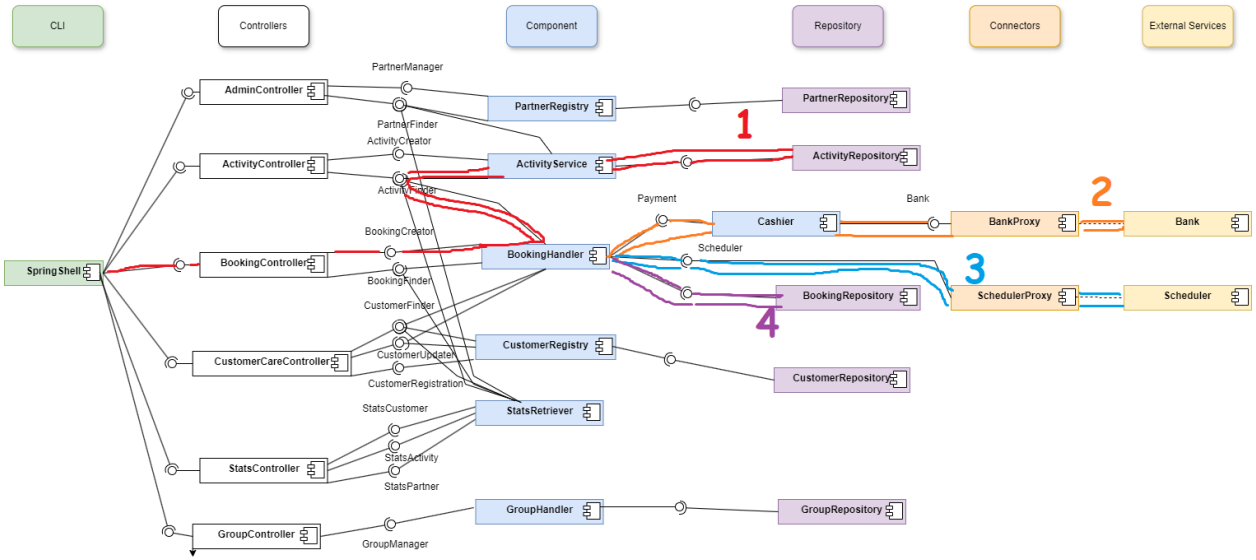
ces statistiques pour pouvoir séparer la logique métier et désengorger les responsabilités portées par le composant *StatsRetriever* qui s'apparente donc à une classe Dieu.

C. Spécificités liées aux clients

Évidemment, comme vous vous en doutez très certainement, les clients sont les utilisateurs principaux de l'application Winter4Everybody, car ce sont eux qui vont notamment rapporter de l'argent aux partenaires, et qui vont faire vivre la station. Nous avons donc implémenté différentes fonctionnalités pour les satisfaire :

a. Réserver une activité

La réservation d'une activité est primordiale pour se rendre dans une activité. Le chemin qui permet de réserver une activité est assez long, le voici dessiné sur notre diagramme de composants :



Ce qui est très important à noter ici, est que pour réserver une activité, nous avons intégré deux composants externes au backend qui sont la Bank et le Scheduler. Sur le plan fonctionnel, la Bank sert à valider le paiement d'un utilisateur (si ce dernier paye avec sa carte bancaire) pour l'activité, et le Scheduler sert à pouvoir réserver un créneau attribué à l'utilisateur pour l'activité en question. Sur le plan technique, ces deux composants externes ont été réalisés grâce à la technologie NestJS.

Différentes règles ont également été intégrées indirectement dans la réservation d'une activité, comme le fait que pour 1€ dépensé lors de la réservation d'une activité, vous recevez 2 points qui vous permettront de pouvoir réserver une nouvelle activité ou un extra (ie une activité payable seulement avec des points). De plus, nous avons

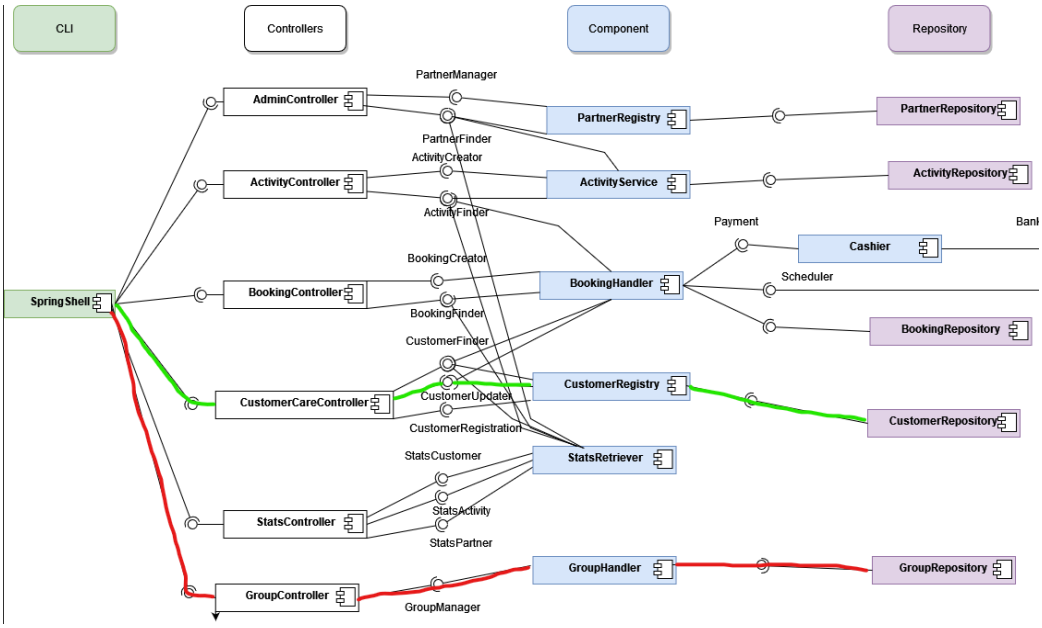
implémenté le fait que si l'activité en question est un forfait de ski, alors cette dernière ne rapporte pas de points sur le compte de l'utilisateur.

Améliorations potentielles : La principale amélioration que nous pouvons dire, est qu'il est possible de pouvoir créer de nouveaux composants pour pouvoir alléger la charge métier que supporte le composant BookingHandler qui peut donc se rapprocher d'une classe Dieu. De plus, comme vous pourrez le lire dans la prochaine patie consacrée au modèle métier, il serait intéressant de pouvoir ajouter un ENUM pour définir le type d' activité (forfait de ski, réservation à l'hôtel etc…) car ici notamment pour la règle du forfait de ski, nous utilisons une regex pour pouvoir vérifier la règle ce qui n'est pas scalable dans le futur.

b. Groupes d'utilisateurs

La gestion des groupes d'utilisateurs est une fonctionnalité essentielle pour permettre aux utilisateurs de créer des groupes, d'ajouter des membres à ces groupes et d'avoir des avantages lors du transfert de points. En effet dans notre implémentation, lorsque deux utilisateurs se transfert des points alors qu'ils ne font pas partis du même groupe ils subissent un malus de 10% de perte de point sur le transfert.

Les groupes sont modélisés dans notre système par l'entité `UserGroup`, qui contient un identifiant unique, un nom pour le groupe, et une liste des membres du groupe. Chaque membre est une instance de l'entité `Customer`, permettant ainsi une relation entre les groupes et leurs membres. L'information de transfert de point entre le cli et le backend transite par un objet TransfertPointRequest, voici le chemin parcourut pour le transfert de point d'un utilisateur à un autre en VERT et en ROUGE la création d'un groupe.



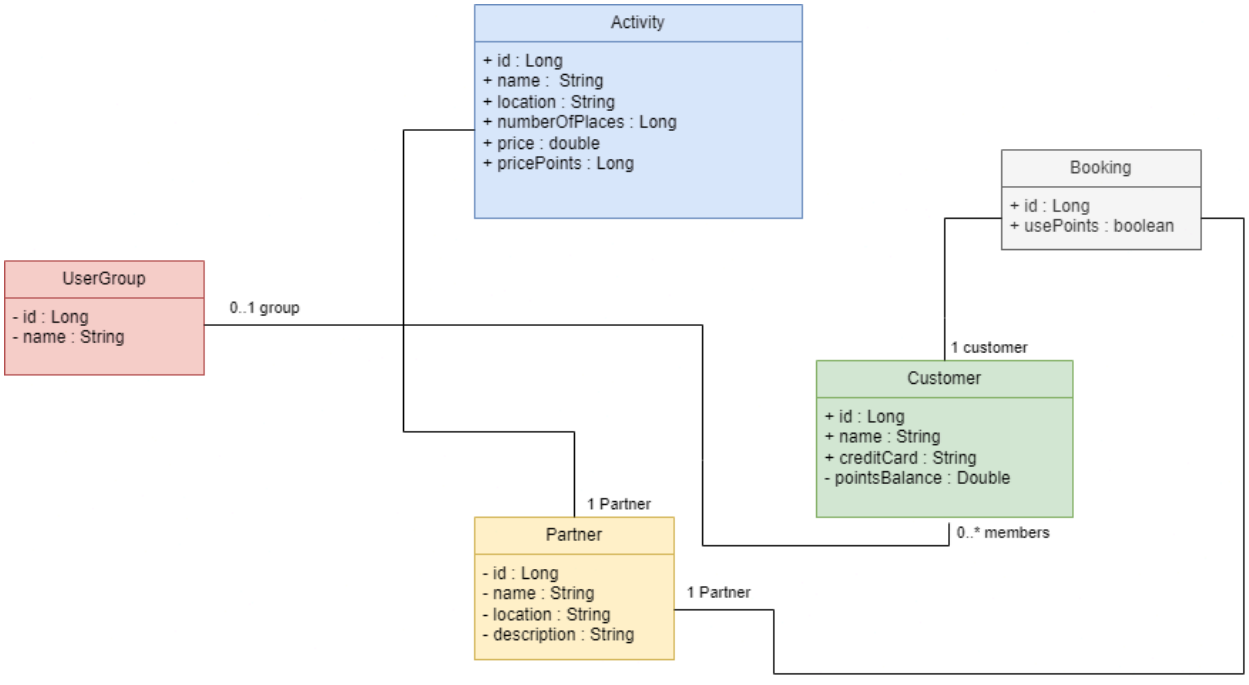
Améliorations potentielles : On aurait aimé faire en sorte que lorsqu'un utilisateur rejoint un groupe il rejoint aussi les activités non terminées sur lesquelles il reste des places. Faire aussi en sorte qu'un utilisateur dans le groupe soit en quelque sorte administrateur du groupe et puisse ajouter des activités aux membres de celui- ci. Un autre aspect aurait été de mettre en place des réductions sur les activités réservées en groupe.

c. Avoir accès au catalog d'activités

Pour pouvoir garantir une UX importante sur l'application Winter4Everybody, nous avons également mis en place un catalog permettant de pouvoir lister toutes les activités possibles dans la station, mais également un catalog permettant de référencer tous les extras (les activités seulement réglables avec des points) de la station :
L'implémentation du catalog est relativement simple et il n'y a pas grand chose à détailler dessus, cela se base juste sur la récupération des données présentes dans la base de données et ensuite de mettre en forme de String pour pouvoir l'afficher sur l'écran de l'utilisateur.

2. Modèle métier

Dans cette partie, nous allons vous expliquer le modèle métier que nous avons implémenté lors de la réalisation de notre projet Winter4Everybody. Pour pouvoir vous présenter ce modèle métier, nous avons réalisé un diagramme modélisant les différents objets métiers que voici :



Notre modèle métier est assez simpliste et proche d'un MVP, mais nous en sommes contents car comme vous avez pu le lire dans la partie précédente, nous avons pu

réussir à implémenter les différentes spécificités demandées dans le sujet du projet. Cependant, quelques points peuvent être améliorés :

- Premièrement, il serait important d'intégrer une liaison plus forte des éléments entre eux. Par exemple, comme mentionné précédemment, une activité connaît le partenaire associé, mais le partenaire ne connaît pas directement les activités associées, ainsi cela peut complexifier l'architecture et le code quand on souhaite retrouver toutes les activités d'un partenaire en particulier.
- Deuxièmement, dans le but de pouvoir étendre potentiellement l'application Winter4Everybody pour d'autres saisons (Spring4Everybody, Summer4Everybody ?) Il pourrait être judicieux de pouvoir intégrer un **"ENUM"** qui pourrait contenir les différents types d'activités qui sont proposés. Ainsi, grâce à cet **"ENUM"**, il serait beaucoup plus simple d'implémenter différentes règles en fonction du type de l'activité (comme par exemple la règle *"Hébergement et petits déjeuners ne rapportent aucun point sauf pour des séjours longue durée dépassant 1 semaine.*)

3. Forces et faiblesses de l'architecture

Maintenant que nous avons détaillé les différents points liés à notre architecture, nous allons pouvoir vous expliquer les différentes forces et faiblesses que nous avons pu établir à la suite de la réalisation de ce projet.

Force de l'architecture :

L'une des principales forces de notre architecture est le fait qu'elle est simple à prendre en main et à comprendre, et qu'elle permet notamment de pouvoir y ajouter de nombreux composants/controllers de manière assez simple si l'on souhaite étendre les fonctionnalités de Winter4Everybody. Par exemple, comme nous l'avons mentionné précédemment, si nous souhaitons ajouter un attribut "TYPE" aux activités, il suffit simplement de créer un ENUM et de le lier aux activités et ensuite d'adapter le code des composants pour y intégrer les règles que l'on souhaite.

Cette force est très importante car elle permet de pouvoir garder une certaine maintenabilité de notre code notamment si on imagine qu'une nouvelle équipe de développeurs serait chargée de continuer le développement de notre produit.

Faiblesse de l'architecture :

La principale faiblesse de notre architecture réside principalement dans la mise en place de notre modèle métier, et dans le fait que les éléments ne sont pas forcément montés en cascade. En effet, il manque beaucoup de liens entre les différents objets métiers et ainsi on perd de l'information sur ces derniers, ce qui peut notamment poser des soucis si l'on ajoute des éléments CRUD notamment comme la modification de partenaires, la suppression de partenaires….

4. Répartition des points

Tous les membres du groupe sont fiers du travail effectué durant ce projet d'ISA-DevOps. Même si nous avons accumulé du retard au début du projet, nous ne nous sommes pas découragés et nous avons réussi à instaurer un bon processus de travail, qui a permis à tout le monde de pouvoir toucher à la partie ISA et également à la partie DevOps. Nous avons donc décidé de répartir équitablement le nombre de points :

- MAUROIS Quentin : 100 points
- FROMENT Lorenzo : 100 points
- STENGEL Damien : 100 points
- BEUREL Simon : 100 points

IMPORTANT :

Certains pushs sur Github ont été faits avec le mauvais compte de l'élève malheureusement, ainsi le compte "Zimitim" appartient à Simon Beurel, et le compte "Vranox" appartient à Damien Stengel.