UNIVERSITÉ CÔTE D'AZUR

POLYTECH

Polytech Nice Sophia

ISA

Rapport d'architecture

réalisé par l'équipe

H

composée de

Dan NAKACHE, Sasha POURCELOT, Thomas DI GRANDE
& Vincent TUREL

Année universitaire 2021/2022

Sommaire

Introduction

Voici le rapport d'ISA de notre projet de semestre 8 à Polytech'Nice Sophia. Dans ce rapport, nous n'allons aborder que les notions qui ont été abordées dans le cours d'ISA. Les notions associées au cours de DevOps seront, elles, détaillées dans le fichier *livrable.md* présent à la racine du dépôt. Dans un premier temps, nous allons vous présenter l'architecture de notre projet, ses forces, mais aussi ses faiblesses. Dans un second temps, nous verrons plus en détail cette architecture sous forme d'un diagramme de composant présentant la façon dont les composants interagissent avec leur contrôleur. Après cela, nous verrons plus en détail les objets métier avec leurs interfaces dans le diagramme de classe. Nous aborderons ensuite plus en détail les perspectives d'amélioration de notre projet en explorant l'implémentation de nouvelles fonctionnalités. Enfin, nous conclurons par l'explication de notre implémentation de la persistance, en détaillant pour chaque élément rendu persistant pourquoi et comment.

I/ L'architecture, forces et faiblesses

1. Le découpage en modules

L'application a été découpée en plusieurs modules. On a choisi de découper le système pour regrouper ensemble les éléments remplissant une fonction similaire. On distingue ainsi les modules suivants.

Le module *entities* regroupe tous les objets métiers de l'application ainsi que les DTOs à un seul endroit. Les différents objets métier sont détaillés dans la partie suivante du rapport au niveau du diagramme de classe. Ces objets permettent de modéliser chaque aspect de notre implémentation (par exemple, les *Room* représentent les salles avec leurs attributs) avec comme seule méthode les *mutators* et des *accessors*. Leur but est en effet de modéliser les objets de notre système, ils sont considérés comme des objets ne contenant que des données et ne contenant pas d'autres fonctions.

Le module *controller* permet d'exposer les fonctionnalités de notre application au monde extérieur, il contient l'ensemble des contrôleurs nécessaires pour pouvoir interagir avec nos composants métier. Ce module contient également les tests d'intégration de notre application, permettant de garantir le bon fonctionnement de l'ensemble de notre implémentation. Chaque *contrôleur* n'a pour seule responsabilité que de donner l'ensemble des méthodes nécessaires pour utiliser pleinement les composants métier qui lui sont rattachés. Comme cela nous avons pu diviser les responsabilités entre plusieurs contrôleurs en fonction du besoin utilisateur qu'ils représentent.

Le module *components* regroupe la logique métier de notre application, c'est-à-dire l'ensemble des composants métier qui permet d'utiliser nos objets métiers. Le but ici est de répartir aux mieux les responsabilités entre les différents aspects de notre projet. La plupart des composants métier intègre des méthodes pour pleinement utiliser nos objets métier. Certains intègrent également des méthodes faisant appel à d'autres composants ayant eux même leur propre responsabilité. Les tests unitaires sont les plus utilisés pour vérifier que tout

fonctionne dans un module donné. Cette fonction est également remplie par certains tests d'intégration. Les tests d'intégration sont construits sous forme de test JUnit (et pas cucumber) car leur but ici est de simplement vérifier que deux ou trois composants communiquent correctement entre eux. Il ne s'agit pas ici de scénario mais simplement d'intégration de deux (ou plus) composants ensemble.

Le module *event-app* contient les fonctions de démarrage de l'application. Il s'agit de la partie la plus petite du projet car elle ne contient pas de code métier mais uniquement la méthode main qui est lancée à l'exécution du programme (le backend).

2. Forces

Il y a plusieurs avantages à avoir découpé le projet en plusieurs modules. En effet, le premier le plus évident, est que ce découpage nous à permis de mieux répartir les tâches de développement entre les membres de notre équipe. Par exemple, un membre peut développer un nouveau composant métier avec son objet associé, et le tester jusqu'à pouvoir garantir son fonctionnement. Dans le même temps, un autre membre peut réaliser le contrôleur associé à ce nouvel objet, en se basant sur l'interface associée au composant métier. Dans ce cas de figure, comme les composants sont forcément testés, nous pouvons facilement garantir qu'à l'appel du composant dans un contrôleur, on aura le comportement voulu. En cas de comportement suspect, on pourra facilement trouver l'origine du problème dans notre projet. Si les tests unitaires ne passent pas, alors c'est le composant métier ou son objet métier associé qui est en défaut, sinon c'est le contrôleur.

Une autre grande force de cette architecture est que ce découpage en module permet de ne pas forcément recompiler l'ensemble de notre projet à chaque nouvelle version. En effet, si nous n'avons pas de nouvel objet métier à ajouter mais simplement un composant métier supplémentaire, seul le module contenant les composants sera recompilé. Le reste de nos artefacts sont précompilés et conservés dans un serveur Artifactory que nous avons déployé et dont le but est de les rendre accessibles par tous les membres du projet.

Enfin, un dernier avantage à ce découpage est qu'il permet une meilleure répartition des responsabilités entre les différentes parties de notre projet, en séparant clairement la partie contenant le code métier, les contrôleurs, les objets en eux même et l'application front end destiné à l'utilisateur (ici, c'est la CLI).

3. Limites, faiblesses

L'une des principales faiblesses de notre architecture est la complexité des interactions entre les différents composants. Cette complexité a eu tendance à se propager vers les contrôleurs. Une conséquence est que l'utilisation de la Command-Line Interface (CLI) requiert d'entrer des commandes particulièrement longues et compliquées. Pour pallier ce problème, nous pourrions imaginer relier notre application à une interface graphique permettant d'abstraire cette complexité du point de vue du client, et de l'assister via l'interface graphique. En effet, les chemins REST, uniques à chaque opération, rendraient le rajout de cette interface assez simple à mettre en place.

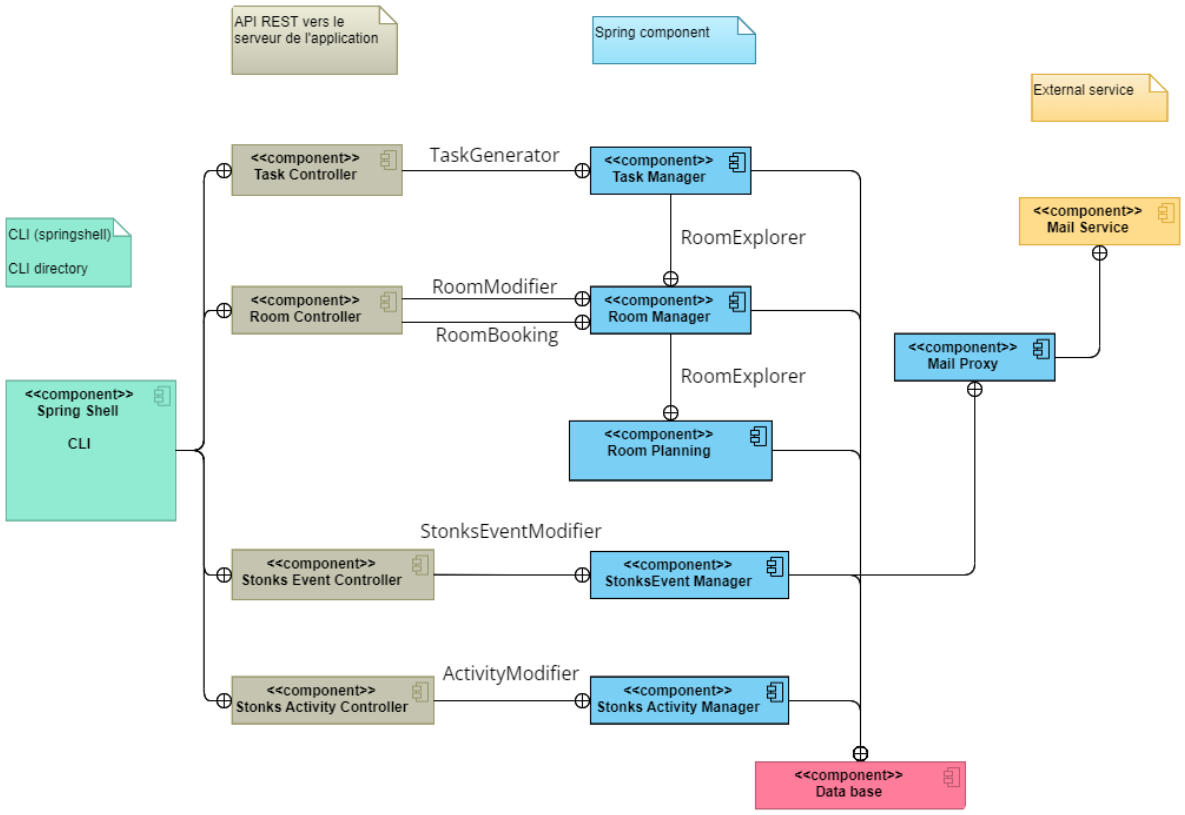
La séparation du code métier en composant accroît la capacité de notre code à pouvoir être facilement corrigé mais augmente également sa complexité. Cette grande complexité accroît ensuite le nombre de bugs possibles liés à la communication inter-composant. Pour cette raison, il faut multiplier les tests d'intégrations à chaque ajout de composants métier pour vérifier qu'il n'y ait pas de comportement indésirable.

Une autre faiblesse que l'on peut observer est qu'un changement minime au niveau d'un composant déclenche invariablement la recompilation du module entier. Ceci est dû au fait que nos composants ont beaucoup de responsabilité. Une possibilité d'amélioration serait de séparer une partie de nos composants et de nos contrôleurs en sous module en fonction de leurs responsabilités. Un bon exemple serait de séparer notre composant de création et de gestion de planning en deux composants pour limiter les responsabilités.

Enfin, une dernière limite à notre découpage est la grande quantité de responsabilité endossée par certains composants. Cela se répercute sur le contrôleur correspondant et invariablement sur la CLI. La responsabilité est bien partagée, mais une "zone de rencontre" de l'information a émergé. Une possibilité d'amélioration serait ici de redécouper ces composants en deux, en les associant à deux contrôleurs différents. Par manque de temps, cela n'a pas pu être réalisé.

II/ Nos composants

1. Diagramme de composants



2. Justification de l’architecture

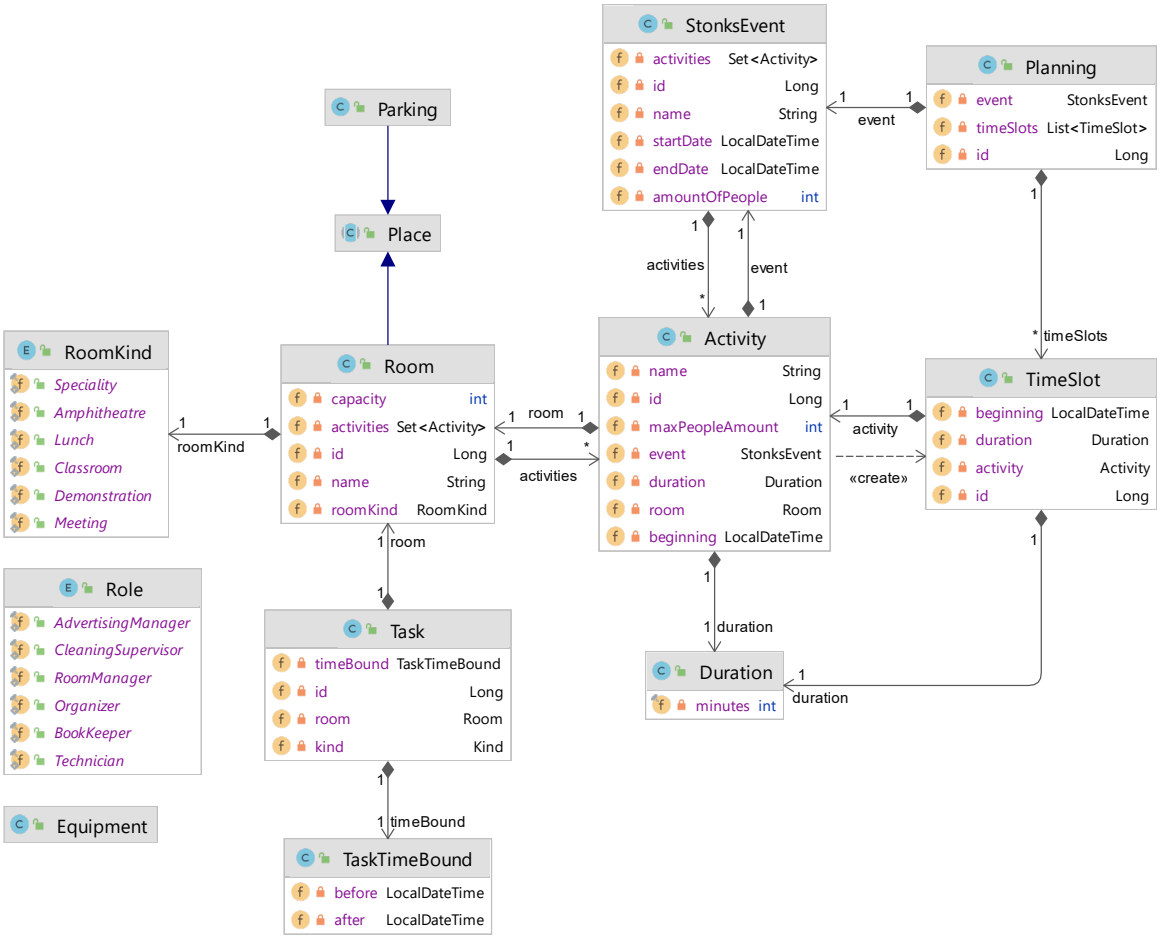
Nous avons choisi cette architecture en fonction des connaissances que nous avons acquises en cours d’ISA. La répartition est sous forme de composants qui communiquent entre eux, ces composants sont liés à leurs objets métier. Ils sont utilisés grâce aux endpoints donnés par les contrôleurs.

Le premier aspect de notre implémentation concerne les tâches planifiées. Le contrôleur *TaskController* est relié au composant métier *TaskManager* qui permet l'édition de ticket, utilisé surtout pour indiquer les salles qui devront être nettoyées après l'événement. Ce composant est lié au room manager pour pouvoir ajouter les périodes de nettoyage sur le planning des salles. Ensuite le *RoomController* est relié au *RoomManager*, ce composant s’occupe de gérer les salles et tout ce qui leur est associé tels que la taille, où le type de salle. Il est lié au *RoomPlanning* qui gère le planning et la disponibilité des salles. Chaque salle est liée à un planning dès lors qu’elle est utilisée pour une activité dans un événement. Ensuite, le *StonksEventController* est relié au *StonksEventManager* qui est le composant métier responsable de créer et d’administrer les événements. Attention, ce composant n’a absolument rien à voir avec une notion d'événement de micro-service ou autre déclencheur. Il s’agit d’un nom choisi car en lien avec le sujet et dont la seule signification est de désigner le composant de création des événements du PolyEvent. Enfin, le *ActivityController* est relié au *ActivityManager* qui est un composant en charge de la création d'activité.

Chaque élément contient un nom unique (qui correspond à son identifiant du côté utilisateur). Un événement est un ensemble d'activités créé. Chaque activité disposera d’une salle, d’un nombre maximal de personnes présentes. Lors de la création de l'événement, l’organisateur planifie les dates de début et de fin, le nom et le nombre maximal de personnes attendues. Il peut ensuite ajouter des activités avec une date de début et une durée comprise dans l'intervalle des dates de l'événement. Le but des activités est la représentation d’une fraction d’un événement pour simplifier la réservation de salle. Tous les composants sont reliés à notre base de données pour la persistance des informations, comme détaillé dans un paragraphe plus bas. Enfin, le dernier aspect de notre implémentation concerne le système de mail automatique, qui est notre service externe. Il remplace la banque, car nous préférons envoyer à l’organisateur et au gestionnaire comptable de l’école les devis de prix pour les événements. Autrement dit, notre application est un assistant pour la création des événements de l’école mais n’est pas directement utilisé pour le payement de ces derniers. Bien entendu il s’agit d’une piste d’amélioration qui pourrait être abordée.

III/ Objet métier

1. Diagramme de classes



2. Pourquoi de tels objets

Notre projet à été articulé autour de ses différents composants métier. Hors, tous ses composants ont besoin de leurs objets métier associés. Les objets les plus importants dans notre projet sont les objets *Room*, qui contiennent tout ce qui est en lien avec les salles. L'objet *Activity* contient lui, l'ensemble des données caractérisant une activité, telles que son nom ou la date de début et sa durée. L'activité est liée à une salle pour disposer d'un espace pour la réaliser, mais elle est aussi liée à notre objet *StonksEvent*, qui regroupe l'ensemble des activités pour un même événement. L'événement dispose également d'une date de début et de fin. Tous ces objets communiquent avec notre objet *Planning* utilisant lui-même l'objet *TimeSlot* pour stocker nos dates et nos plannings. L'objet *Task* permet de stocker les tâches et les tickets envoyés par les différents organisateurs pour organiser le nettoyage des salles après les événements. Enfin, les différentes énumérations permettent de placer des mots compréhensibles pour les utilisateurs.

IV/ Capacités d'évolutions

1. Une gestion du matériel

C’est probablement l’un des plus gros ajouts que nous aurions pu implémenter dans notre projet. En raison d’un manque de temps dû à l’implémentation de la JPA, nous n’avons malheureusement pas pu pleinement intégrer cette fonctionnalité. En effet, nous disposons bien d’un composant et de son objet associé pour la gestion du matériel, mais il n’est pas relié aux composants cruciaux tels que le choix des salles en fonction du matériel déjà disponible. Par manque de temps, nous avons donc dû nous résoudre à ne pas implémenter cette fonctionnalité. C’est par conséquent une perspective d'amélioration certaine de notre projet et serait bien entendu notre priorité d’ajout si nous disposions de plus de temps. Pour rentrer dans les détails par rapport à l’implémentation possible, nous pourrions inscrire dans une salle le matériel disponible, puis inscrire dans une réserve (salle spéciale avec seulement du matériel) l’ensemble du matériel disponible. Tout cela en ayant une gestion d'éventuels achats de matériel supplémentaire, que l’on pourrait regrouper pour optimiser l’utilisation des salles en fonction de la place disponible, mais aussi du matériel ainsi que limiter le plus possible l’achat de nouveaux objets pour réduire les coûts d’organisation de l'événement.

2. Un ajout de salle virtuelle

Depuis la pandémie de covid-19, l’organisation d'événements regroupant beaucoup de monde est devenue plus difficile et il est bien plus courant de réaliser une partie de ces événements en ligne sous forme de réunion avec plusieurs personnes. On y trouve des stands virtuels et d’autres sous-réunions permettant aux utilisateurs de pouvoir participer à un événement sans prendre de risque de propager le virus. Nous pourrons pleinement ajouter cette fonctionnalité, en ajoutant des salles virtuelles, jumelées ou non avec des salles réelles permettant aux usagers d’assister aux conférences et ce même si le nombre maximum de spectateurs est atteint. De plus, ces salles virtuelles rendraient l’accès à l'événement plus facile et permettrait d’étendre grandement le nombre de visiteurs concernés tout en contournant la contrainte du déplacement. L’ajout de cette fonctionnalité passera par l’utilisation d’un service externe bien connu dans les universités, la plateforme de réunion Zoom. Nous aurions donc un ensemble contrôleur et composant métier dédié à la création et la planification des réunions zoom. Le but ici n’est pas de réécrire tous les composants de zéro, mais plutôt de relier nos nouveaux composants métier à nos anciens de façon intelligente, tel que la réutilisation des méthodes de notre composant planning qui existe déjà et pourrait bien entendu créer les plannings associés à une salle virtuelle. Ces plannings, lors de leur création, déclencherait automatiquement la création de réunion zoom.

3. Une cafète intégrée à l'événement :

Un dernier ajout possible pour notre implémentation serait éventuellement l’ajout d’un système de gestion de la cafétéria. Dans le cadre de cette fonctionnalité, nous ferons appel à un service externe chargé de faire le lien entre une éventuelle commande et les différents traiteurs qui fourniront les ressources. Notre implémentation en elle-même consisterait à donner dans un premier temps le choix à l’organisateur de sélectionner un ou plusieurs menus types pour son événement tel que des biscuits apéritifs, des boissons, etc… Puis, en fonction du nombre de personnes attendues, le système réaliserait un devis provisoire. L’organisateur pourra alors relire le devis, le corriger en fonction de l’estimation du nombre de personnes et ajuster les coûts en conséquence. Puis la validation du devis le rend alors définitif. Un mail

sera alors envoyé automatiquement au service extérieur lors de la validation définitive de l'événement, qui se chargera de réaliser la commande auprès des traiteurs. Cependant cet outil ne permet pas de réaliser des commandes auprès de restaurant ou autre pour d'éventuelles demandes particulières. Il ne pourra donc pas commander de plat sur mesure ou autre sauf si une modification est apportée par l'ajout d'un nouveau menu depuis le front end.

V/ Implémentation de la persistance

La persistance est gérée grâce à la *Java Persistence API* (JPA), en s'appuyant sur le pilote *Postrges* permettant de stocker de façon totalement transparente et structurée les données sur une base de données PostgreSQL. Cela a plusieurs avantages. Premièrement, on peut redémarrer le backend pendant le développement sans perdre les données qu'on y avait enregistrées, puisqu'elles sont sauvegardées dans une base de données séparée. Deuxièmement, cela permet lors de la mise en production de pouvoir dupliquer le container du backend pour répondre à un pic de requêtes. De la même façon, on peut faire tourner la base de données dans un cluster de plusieurs containers de façon à avoir un système robuste. Nous avions initialement pensé utiliser PostgreSQL pour la mise en production et H2 pour les tests où une base de données est nécessaire. Après plusieurs tentatives de setup et de combats acharnés contre les technologies impliquées, nous avons jeté l'éponge et décidé de toujours utiliser PostgreSQL.

Au niveau de l'implémentation, nous stockons à ce jour dans la base de données les objets suivants : *Event*, *Activity*, *Room*, *Planning*, *Task* et *TimeSlot*. Tous ces objets ont un ID auto-généré et sont liés par diverses relations ***ManyToOne***, ***OneToMany*** et ***OneToOne***. Toutes les relations sont chargées de manière *LAZY* car aucune raison valable ne nous a poussé à modifier ce comportement par défaut. Les activités sont reliées aux événements en utilisant le mode *cascade.ALL* puisqu'on ne souhaite pas garder des activités sans événements associés. En revanche, la relation *Room* - *Activity* ne l'est pas puisque nous voulons garder une certaine indépendance entre les salles et les activités. Enfin, les classes *Duration* et *TaskTimeBound* sont annotées ***Embeddable*** puisqu'il n'y a pas besoin de les stocker dans une table en tant qu'entité à part entière.