Maxime Flament
Théo Frasquet
Lucas Matteo
Jehan Milleret

Rapport d’architecture pour le MVP de PolyEvent

Introduction to Software Architecture

Equipe C

2017 - 2018

Introduction

Ce rapport présente notre proposition d'architecture pour le produit minimal viable (MVP) de PolyEvent, qui est un logiciel dédié à la facilitation de la création d'évènements, et à la gestion de la logistique nécessaire au bon déroulement de l'évènement.

Dans une première partie, nous présenterons et détaillerons notre conception du MVP de PolyEvent.

Dans une seconde partie, nous expliquerons et justifierons nos choix de conception. Enfin, nous discuterons les avantages et limites de notre conception, et évoquerons d'autres solutions possibles que nous aurions pu choisir de mettre en place pour réaliser ce MVP.

Vous trouverez, dans le dépôt *main*, nos diagrammes en format *PNG* et en plus grande résolution que ceux présents dans ce rapport.

Architecture du MVP

Cas d'utilisation de haut niveau

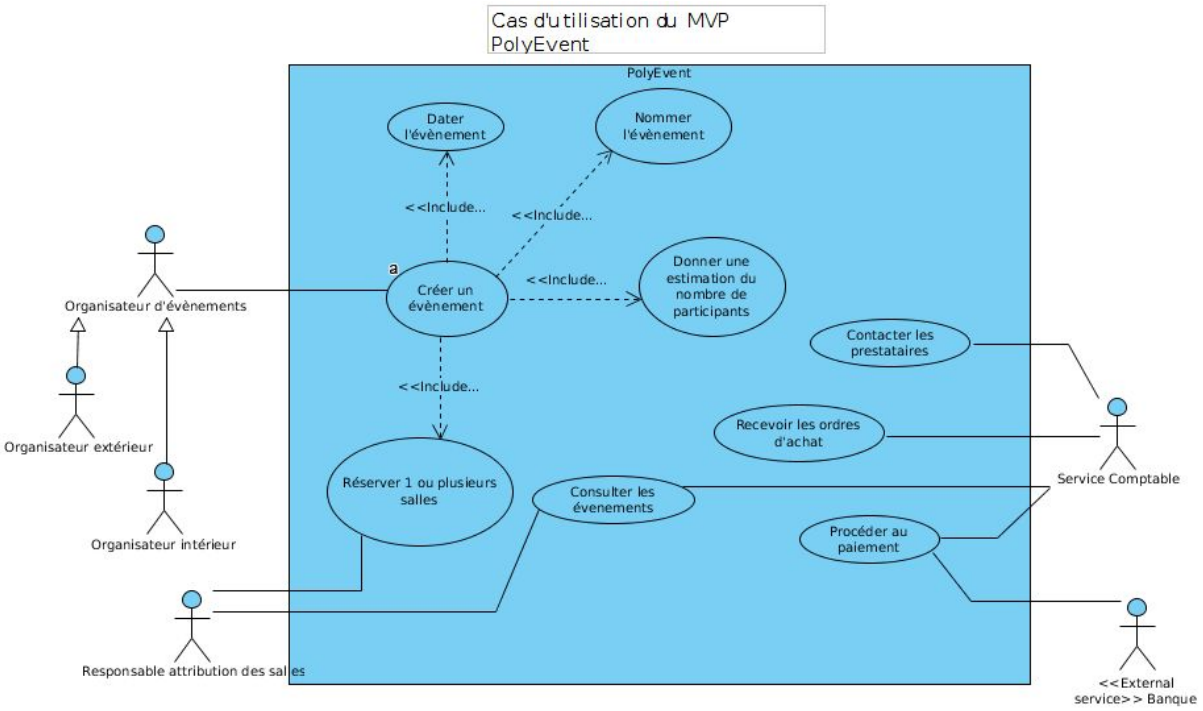

Figure 1 - Cas d'utilisation de haut niveau du MVP PolyEvent

Dans la première proposition d'architecture, nous avions choisi de nous concentrer sur la création simple d'un événement, pour que le processus de gestion d'événement puisse être, bien que minimal, réalisé du début à la fin. Cependant, il nous a été reproché que nos cas d'utilisation ne permettent pas d'exprimer l'ensemble des besoins minimaux des différents personas, et sont fermés à l'extension. Nous avons donc ajouté de nouveaux cas, permettant de régler ces deux problèmes. Ainsi, un organisateur a la possibilité de créer un événement en spécifiant un nom, une date et le nombre de participants. Si il a des salles particulières qu'il a l'habitude de réserver il peut les choisir en particulier, sinon les salles sont choisies automatiquement en fonction de leur capacités.

Une fois l'événement créé, le responsable du service comptable reçoit les ordres d'achat correspondant aux prestations et aux salles demandées pour l'évènement nouvellement créé. Le service comptable pourra ainsi contacter les différents prestataires

pour répondre aux besoins de l'organisateur, et procéder au paiement de ces dits-prestataires, grâce au service de paiement externe connecté à notre système.

Enfin, le responsable des attributions de salles recevra la liste des salles fournies par l'organisateur, ou bien choisies par notre système, et pourra, ou non valider la requête, en fonction de la disponibilité des salles.

Diagramme de Classes Métier

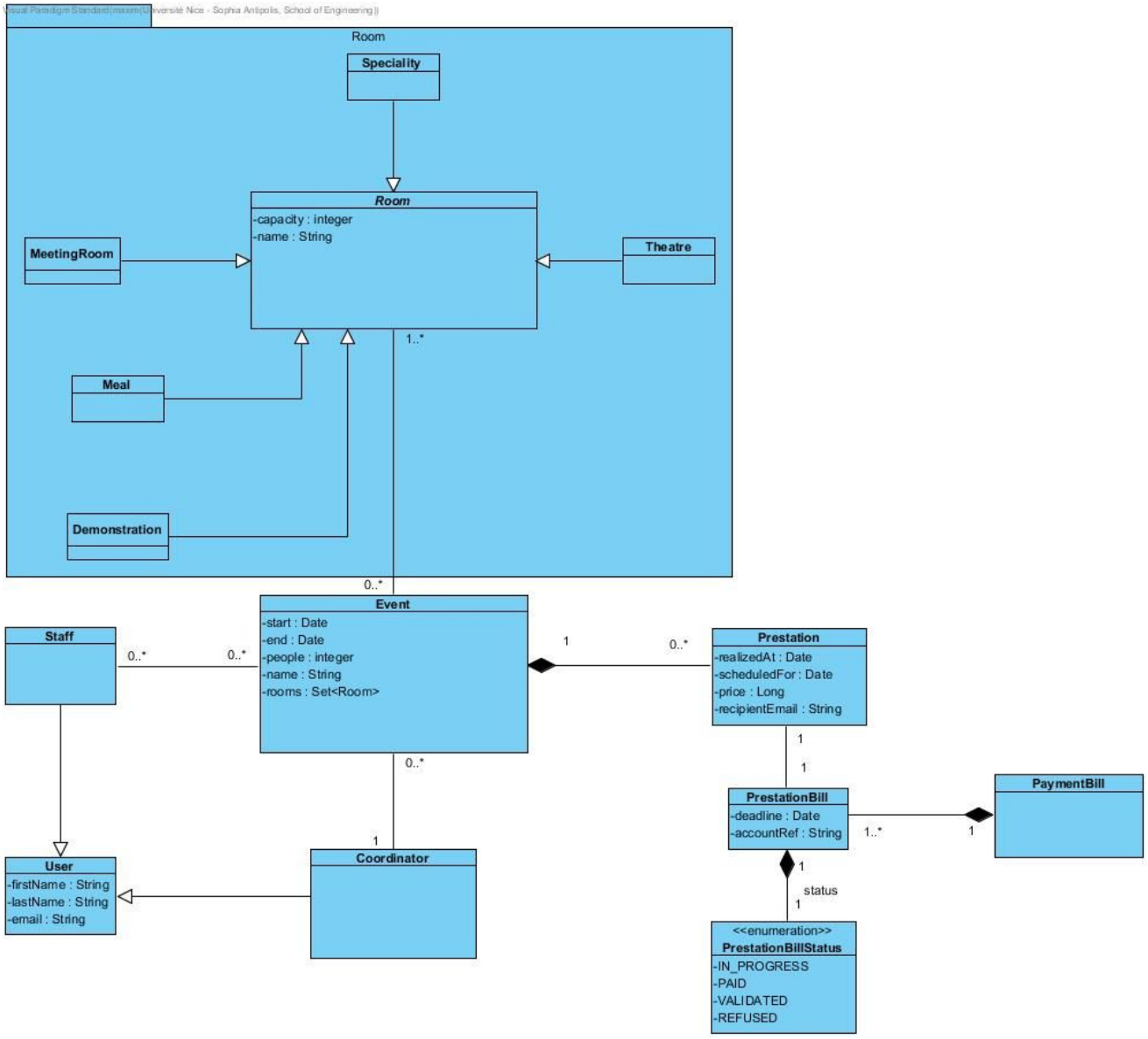

Figure 2 - Diagramme de classes métier du MVP PolyEvent

Le diagramme de classes ci-dessus nous permet de réaliser les fonctionnalités décrites dans la figure 1.

Nous avons choisi de représenter les différentes salles comme une hiérarchie père-fils, puisque les salles ont des attributs et comportements communs (elles ont un nombre de places, un identifiant/nom, elles peuvent être occupées, disponibles) mais ont également des attributs qui peuvent différer (par exemple, un amphithéâtre a nécessairement un projecteur et des enceintes, tandis qu'une salle n'en a pas forcément).

Nous avons choisi une conception similaire pour la représentation des utilisateurs du système, i.e., les différents personas décrits dans la figure 1, puisqu'ils possèdent des informations communes, mais également des rôles différents et des visions des données/fonctionnalités différentes. Un organisateur (*Coordinator*) d'évènements voit les évènements qu'il crée, alors que le service comptable, ou encore le responsable des salles font parti du staff (*Staff*) de l'évènement, puisqu'ils sont là pour permettre à ce dernier d'être organisé.

En ce qui concerne les paiements et les prestations demandées par l'organisateur pour l'évènement nouvellement créé, nous avons décidé d'associer à la création de l'évènement les demandes de prestations de l'organisateur. Ces demandes ne constituent en rien des prestations devant être réalisées, puisqu'il est nécessaire de les facturer avant d'affirmer qu'elles auront lieu. C'est pourquoi les requêtes de prestation sont transmises au service comptable, qui peut générer les factures associées, et définir un statut de traitement de la facture (*PrestationBillStatus*), afin de fournir un suivi d'avancement pour l'organisateur et le staff. Une fois cette facture générée, la prestation est validée et le paiement pourra être effectué une fois la prestation effectuée.

Diagramme d'interfaces

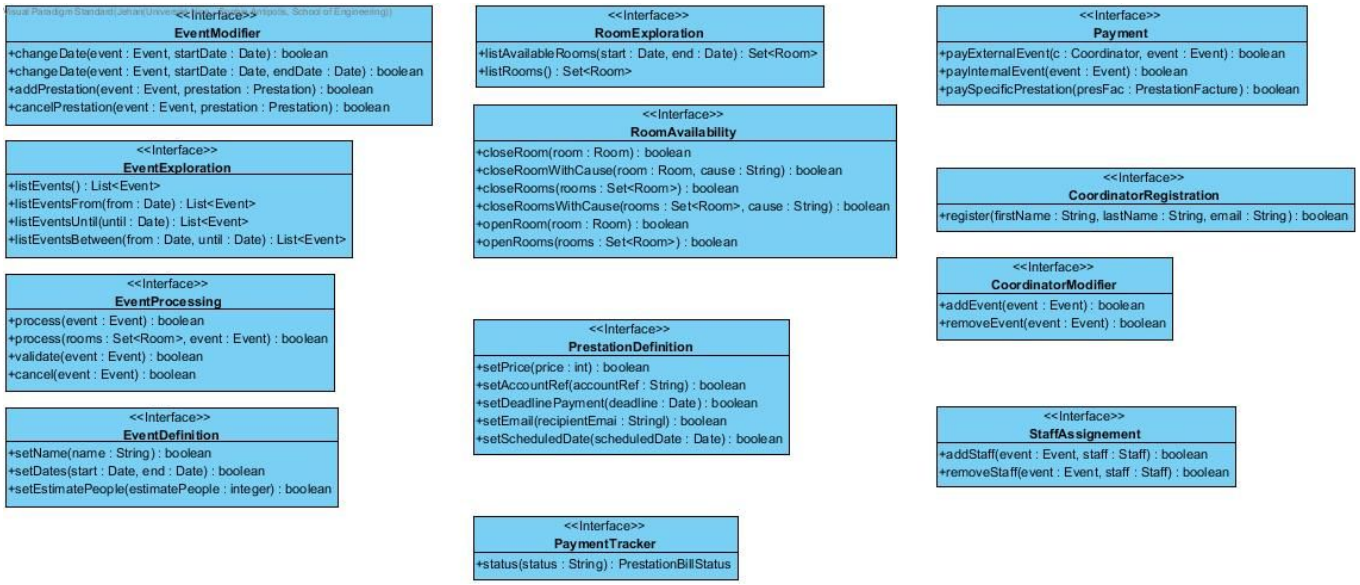

Figure 3 - Interfaces fonctionnelles du MVP PolyEvent

Nous fournissons une liste d'interfaces que nos composants pourront utiliser pour communiquer. Pour chaque appel, nous avons décidé de renvoyer un booléen ou une collection de données (qui peut être nulle). Nous avons fait ce choix afin d'avertir lorsqu'un problème est survenu lors du traitement de la requête. Dans le futur, nous voudrions réaliser un objet *Resultat* qui nous permettra de renvoyer un message de retour précis et les données s'il y en a dans un format universel comme un JsonObject.

Tout d'abord nous avons les interfaces "classiques" de création que sont *EventDefinition*, *PrestationDefinition* et *CoordinatorRegistration*, qui nous permettent respectivement d'ajouter des évènement, des prestations, et des coordinateurs dans le système.

Nous avons ensuite une interface *EventExploration* et *RoomExploration* qui nous permettent de récupérer toutes les informations concernant les évènements et les salles.

CoordinatorModifier est le point d'entrée pour le coordinateur afin de créer ou de supprimer un évènement qui lui appartient dans notre système.

L'interface *EventProcessing* permet de valider l'ajout d'un évènement dans le système. Une fois l'évènement créé par le coordinateur, il est possible de donner uniquement les capacités voulues ou directement les salles que le coordinateur veut

réserver. Lorsque cette phase est passée, le responsable d'attribution des salles va alors réserver les salles. Une fois réservées, le coordinateur valide ou annule l'évènement selon les réservations faites. Un évènement annulé n'est pas totalement supprimé du système comme avec l'interface *CoordinatorModifier*, mais uniquement annulé pour le moment. Il pourra être repris plus tard avec les mêmes informations.

StaffModifier nous permet alors d'ajouter ou de retirer des personnes du staff, tout en sachant que certains rôles sont obligatoires (Service comptable et responsable d'attribution des salles).

Dernier interface lié directement aux évènements, *EventModifier* permet de faire des modifications sur l'évènement. Il comprends également les éventuels ajouts ou suppression de prestations demandées par le coordinateur.

RoomAvailability permet de réserver ou de libérer des salles par le responsable d'attribution des salles, avec une cause s'il le juge nécessaire.

Payment met en avant les méthodes permettant de payer les évènements, qu'ils soient réalisés par des intervenant extérieurs ou intérieurs. Dans le cas d'un intervenant extérieur, un des prestataires est forcément Polytech, qui recevra le paiement pour avoir loué ses salles. La deadline pour le paiement sera alors la seule qui sera avant l'évènement. *PaymentTracker* nous permet alors de changer le statut de la facture (En progrès, payée, puis validée ou refusée selon le retour de confirmation du paiement).

Diagramme de Composants

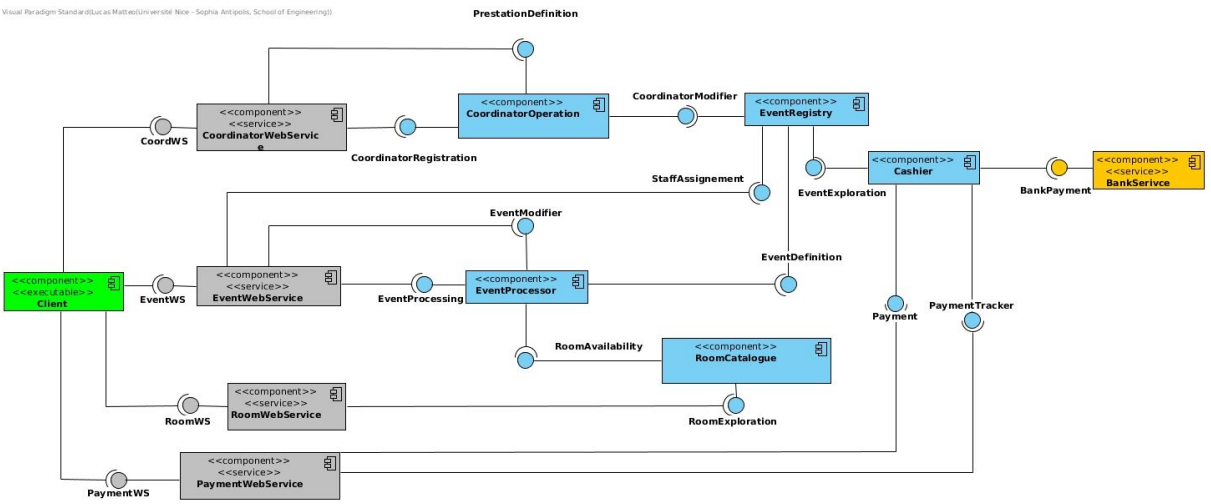

Figure 4 - Diagramme de composants du MVP PolyEvent

Tout d'abord on peut remarquer que nous avons décidé de représenter plusieurs web service, chacun destiné à des utilisateurs ou usages différents. Le *RoomWebService* est fait pour le responsable de l'attributions des salles, ou encore le service de paiement pour le service comptabilité.

Nous avons divisé notre application en 5 composants. Le composant *CoordinatorOperations* expose des interfaces manipulant les objets *coordinator* et *prestation*. Nous avons choisie de rassembler ces deux interfaces sur le même composant car ce sont des fonctionnalités offertes aux coordinateurs d'événements. C'est lui qui fera le choix d'engager ses propres prestataires ou de laisser faire polytech.

La gestion des *event* est séparé sur deux composants differents. Le composant *EventRegistry* a pour responsabilité d'initialiser un *event* dans le système lorsqu'il est créé et de fournir le catalogue des événements présent dans l'application. Le composant *EventProcessor* a lui pour responsabilité "l'organisation" de l'événement. C'est lui qui choisira les salles à attribuer à chaque évènement par exemple en utilisant l'interface *RoomAvailability*. C'est lui qui jugera si l'évènement peut-être fait ou non. C'est lui qui porte l'interface *EventModifier* car si un évènement est modifié, un changement de date par exemple, celui-ci devra être *process* une nouvelle fois.

Nous avons isolé toutes les opérations liés aux paiements dans le composant *Cashier* qui utilise un service externe pour effectuer les opérations bancaires. Il utilise

l'interface *EventExploration* afin de pouvoir trouver les prix des évènements et prestations associées.

ORM

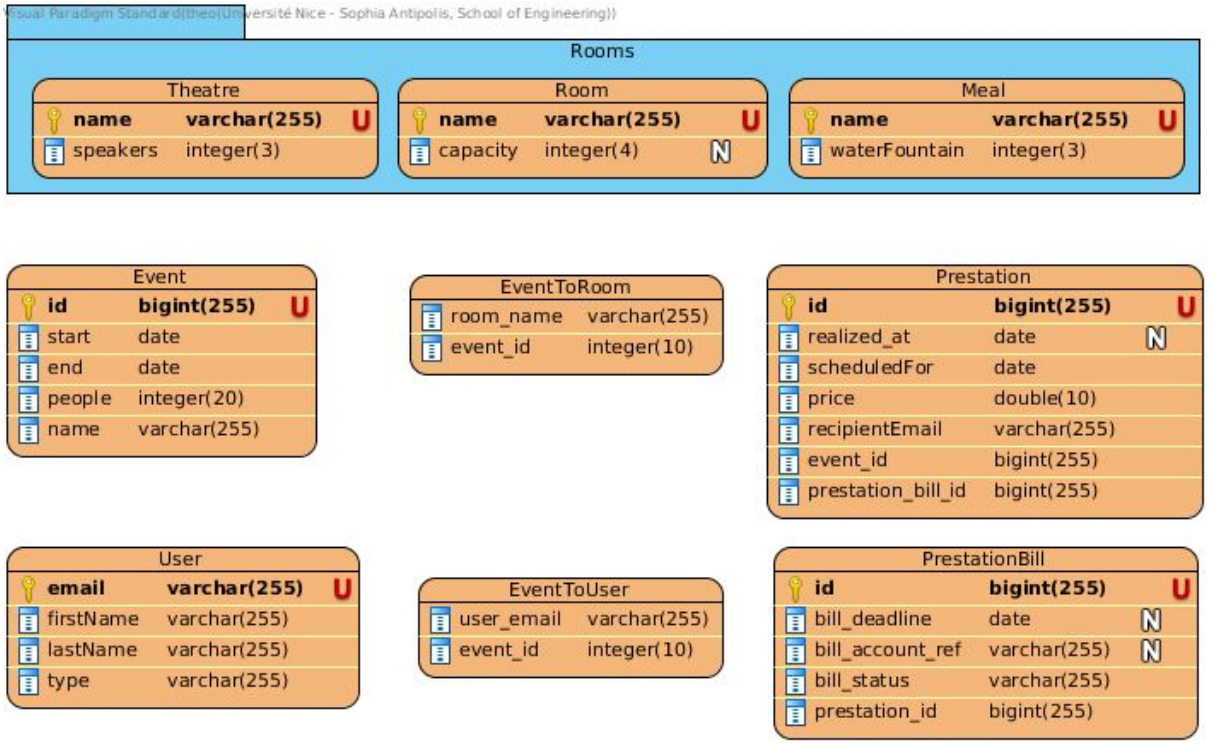

Figure 5 - ORM du MVP PolyEvent

Afin de garantir la persistance des données, et pour s'assurer un passage simple d'objets J2E à des entités relationnelles (et inversement), nous avons réalisé l'ORM visible dans la figure 5.

Nous avons dû faire des choix de représentation des objets du diagramme selon deux aspects :

- l'héritage
- les associations

Les ensembles de classes utilisant l'héritage sont les classes *Room* et *User*. Nous avons utilisé 2 "modèles" connus d'ORM pour traduire cet héritage en relations.

Room : nous avons choisi le modèle “class-table inheritance” pour plusieurs raisons. Tout d’abord, les différents types de salles ont des attributs en communs, et des attributs propres au type. Nous avons défini que la clé primaire des tables *Room** est le nom de la salle, puisque nous supposons ces noms uniques (il serait très peu pratique de devoir chercher des salles, utilisées par n’importe qui, autrement que par leur nom). Etant donné que le nombre d’attributs des différents types de salles est relativement grand (au moins une dizaine), nous ne voulions pas surcharger une unique table en utilisant le modèle “single-table inheritance”, et de ce fait, réduire les performances de recherche, insertion et modification lors d’une requête sur cette table. Enfin, nous avons décidé de ne pas utiliser le modèle “concrete-table inheritance”, afin de ne pas dupliquer des informations dans toutes les tables de salles. Ce modèle aurait pu être intéressant puisqu’il aurait permis de définir des contraintes différentes sur des données communes, comme par exemple la non nullité sur la colonne projecteurs pour les amphithéâtres, et la possible nullité sur cette colonne dans les salles de classe.

Ce modèle présente plusieurs avantages comme de bonnes performances pour la récupération de données, un allègement des tables de données. Cependant, il est moins performant que les autres modèles lors de l’écriture de données, puisqu’il peut être nécessaire de modifier plusieurs tables à la fois (parent et fils).

User : nous avons choisi le modèle “Single-table inheritance”, car nos différents utilisateurs ont, dans la version du MVP, toutes les opérations en commun. Tous les utilisateurs peuvent être des organisateurs d’évènements, et/ou faire partie d’une équipe de staff. De plus, nous n’avons, à l’heure actuelle, défini aucun champs spécifique à un type d’utilisateur en particulier. De ce fait, nous l’avons choisi car il répondait à notre problème de manière simpliste.

Pour ce qui est des associations, nous avions eu à faire un choix pour les associations *Room* ↔ *Event* et *Event* ↔ *User*, *Event* ↔ *Prestation*, et *Prestation* ↔ *PrestationBill* et *PrestationBill* ↔ *PrestationBillStatus*.

Room ↔ *Event* et *Event* ↔ *User* : dans ces deux cas, les associations sont de type *plusieurs-à-plusieurs*, ainsi, nous avons choisi d’utiliser une table d’association, qui est la manière la plus économe, en mémoire, de lier deux entités relationnelles ayant une relation *plusieurs-à-plusieurs*. De plus, cette association respecte la forme *3NF* des bases de données relationnelles.

Event ↔ *Prestation* : ici, nous avons une relation *un-à-plusieurs*. Tout comme pour le cas précédent, nous n'avions qu'un seul choix, utiliser une *Foreign Key* dans la table *Prestation* qui fait référence à l'*id* du *Event* associé.

Prestation ↔ *PrestationBill* et *PrestationBill* ↔ *PrestationBillStatus* : pour ces deux derniers cas, nous avons des associations *un-à-un*. Nous avions deux choix, soit regrouper les deux tables en une, soit utiliser à nouveau des *Foreign Key* afin de lier ces deux tables. Nous avons choisi la deuxième solution, pour trois raisons. D'abord, elle respecte le principe de *responsabilité unique*. Ensuite, utiliser deux tables permet d'obtenir de meilleurs performances, puisque le jeu de données à traiter est moins grand. Finalement, cela nous permet d'instancier ces deux classes séparément, le couplage est réduit.

Diagramme de Déploiement

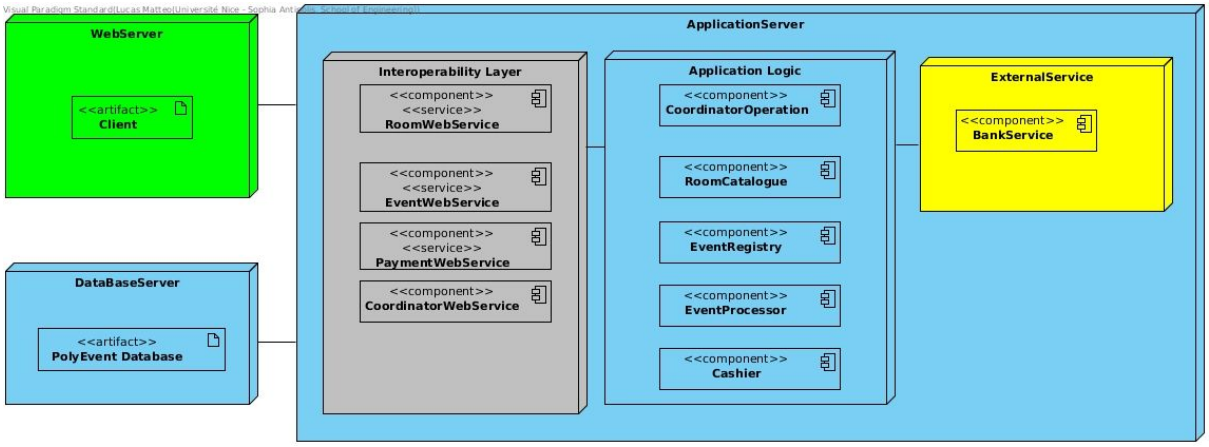

Figure 6 - Diagramme de déploiement du MVP PolyEvent

Notre diagramme de déploiement est composé de 3 *layers* ou *nodes*, qui sont la présentation, le domaine, et la source de données. Nous avons choisi de découper le domaine en trois sous parties, représentées et décrites dans le diagramme de composants, et contenues dans le noeud *ApplicationServer*. Ces 3 domaines travaillent ensemble pour réaliser les fonctionnalités client, et ont chacun un rôle spécifique :

- L'*interoperability layer* a le rôle de façade, pour la communication entre le *front* (*artifact Client*), et offre des méthodes simples pour répondre aux besoins client. De plus, cette couche de l'application suit une architecture *SOAP*, ce qui permet de fortement lier le client à la façade de l'application, et de ce fait, garantir que le client

peut toujours utiliser les fonctionnalités du système, et a toujours la **dernière version** de ces dites fonctionnalités.

- Le sous-noeud *Application Logic* correspond à la partie *EJB* de l'application. Cette couche permet d'isoler la partie complexe du système.
- L'*ExternalService* permet de définir une façade pour des services externes comme une API de paiement. L'isolation de ce "type de composant" offre la possibilité d'intégrer de nouveaux services à l'application, de manière transparente et sûre, pour le reste du système.

Le découpage en noeuds, puis en sous-noeuds de notre système nous permet de tester plus facilement l'ensemble du produit. De plus, le déploiement de PolyEvent est rendu plus simple, du fait que différentes parties du système peuvent être déployées indépendamment du reste des noeuds de l'application.

Avantages, limites, et autres solutions possibles

Avantages

Nous avons limité l'architecture du projet pour qu'il reste un MVP mais l'avons fait de tel façon à ce que notre application soit directement utilisable, tout en étant facilement extensible, par rapport aux autres fonctionnalitées annoncées par le client. L'organisateur peut directement ajouter des personnes du staff selon ses besoins. Ainsi, le responsable logistique pourra utiliser le système pour être au courant des différents évènements annoncés, avec les salles nécessaires pour préparer leur signalisation et prendre contact avec l'organisateur pour le matériel. Le responsable des évènements aura le même accès et pourra préparer la diffusion tout en prenant contact avec le service de comptabilité pour organiser les frais.

De plus, comme nous avions beaucoup de suppositions possible grâce au cadre donné (Polytech), nous avons pu faire de nombreuses suppositions pour avoir un modèle performant. En particulier, notre ORM repose sur des présomptions, comme par exemple le nombre de salles limitées. Si ces suppositions s'avèrent fausses par la suite, il faudra néanmoins repenser à d'autres représentations des héritages.

Limitations

On voit que le composant *EventProcessing* expose et utilise beaucoups d'interfaces et risque d'en utiliser de plus en plus à mesure que l'application se complexifie. Il sera donc peut-être nécessaire de lui retirer certaines interfaces comme *EventModifier* pour éviter qu'il devienne un "GodComponent".

De plus, la classe Event, présente dans notre diagramme de classes métier, comprend également beaucoup de dépendances, ce qui risque d'en faire une godclasse à terme. Afin de remédier à cela, au moment du développement de l'application, nous devrons penser à introduire de l'abstraction ou des design pattern pour pallier à ce problème, et réduire le nombre de responsabilités et de ce fait la complexité de cette classe.

Bibliographie

1. [Internet]. 2018 [cited 28 February 2018]. Available from: https://stackoverflow.com/questions/7296846/how-to-implement-one-to-one-one-to-many-and-many-to-many-relationships-while-de
2. [Internet]. 2018 [cited 26 February 2018]. Available from: http://www.softwareresearch.net/fileadmin/src/docs/teaching/WS11/SE/SE_lecture5.pdf
3. [Internet]. 2018 [cited 17 February 2018]. Available from: https://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf
4. [Internet]. 2018 [cited 18 February 2018]. Available from: http://www.cse.chalmers.se/edu/year/2015/course/EDA222/Documents/Slides/Software_Architecture_IH.pdf
5. [Internet]. 2018 [cited 12 February 2018]. Available from: https://www.tutorialspoint.com/software_architecture_design/introduction.htm
6. [Internet]. 2018 [cited 28 February 2018]. Available from: https://dzone.com/articles/how-to-handle-a-many-to-many-relationship-in-datab