Architecture - Équipe E - Insurance

Shiyang Huang - Mohamed Chennouf - François Melkonian - Lucas Matteo

Personas

Céline est une utilisatrice de BlaBlaMove. Elle souhaite transporter ses affaires dans une nouvelle ville. Elle n'a pas d'assurance et souhaite trouver son contrat avec le service que propose BlaBlaMove.

Julien est un utilisateur de BlaBlaMove. Il n'a rien à déménager mais fait souvent de longs trajets qu'il rentabilise en transportant les affaires d'autres personnes. Il n'a pas d'assurance et souhaite trouver son contrat avec le service que propose BlaBlaMove.

Jean est un utilisateur de BlaBlaMove. Il utilise régulièrement le service pour transporter mais aussi déménager. Cependant il a déjà une assurance avec une agence hors du système de BlaBlaMove et souhaite l'utiliser pour ses déplacements.

Hubert est un assureur. Il propose des contrats pour les utilisateurs de BlaBlaMove. Il a souvent des nouveaux contrats ou des modifications à apporter aux contrats déjà existants et souhaite le faire facilement. Il souhaite aussi être contacté dès qu'un de ses clients a effectué un trajet.

Scénarios

#1

- Celine souhaite faire transporter son ordinateur et sa commode vers sa nouvelle ville.
- Céline interroge les types de contrats d'assurance avec des critères.
- N'ayant pas d'assurance alors que celle ci est obligatoire, elle décide de souscrire à un contrat d'assurance.
- Elle trouve Julien et Jean qui effectuent le trajet qu'elle souhaite.
- Jean décide de prendre l'ordinateur de Celine et a déjà sa propre assurance.
- Julien décide de prendre ordinateur de Celine et décide de souscrire à un contrat d'assurance sur BlaBlaMove.
- Julien et Jean effectuent le transport.
- Un compte-rendu est envoyé à l'assureur de Julien, Jean et Celine.

#2

- Hubert souhaite proposer un nouveau contrat.
- Il fournit les informations de contrat nécessaire sur la plateforme BlaBlaMove.

- Hubert aimerait ensuite vérifier que son nouveau contrat est bien une assurance de vol d'objet électronique. Il souhaite donc afficher sur le système tous les contrats d'assurance de vol.
- Les utilisateurs peuvent choisir ce contrat quand ils décident de souscrire à un contrat.

User Story

Celine

Je souhaite rentrer mes objets dans le système
afin qu'il soit visible par Julien. (mock)

Je souhaite trouver un trajet
afin de transporter mes affaires. (mock)

Je souhaite souscrire un contrat pour un ensemble d'objets
afin d'être assuré.

j'interroge les types de contrats d'assurance offerts aux déménageurs
afin de trouver quel contrat est le mieux adapté pour moi.

Je souhaite savoir si mon contrat est valide sur un trajet spécifique
afin d'être assuré sur tous les trajets.

Je souhaite marquer un trajet comme fini
afin que le rapport soit envoyé à mon agence.

Julien

Je souhaite rentrer mon trajet dans le système
afin qu'il soit visible par Céline. (mock)

Je souhaite trouver des objets à transporter
afin de gagner des points. (mock)

Je souhaite souscrire à un contrat
afin d'être assuré.

J'interroge les types de contrats d'assurance offerts aux transporteurs
afin de trouver quel contrat est le mieux adapté pour moi.

Je souhaite savoir si mon contrat est valide sur un trajet spécifique
afin d'être assuré sur tous les trajets

Je souhaite marquer un trajet comme fini
afin que le rapport soit envoyé à mon agence

Hubert

Je souhaite ajouter un nouveau contrat
afin de divertir mes offres.

Je souhaite mettre à jour mes contrats
afin de m'adapter à de nouvelles situations.

Je souhaite classer mes contrats.
afin que les clients puissent consulter mes contrats en fonctions de leur type.

Je souhaite recevoir les comptes rendus des trajets de mes clients
afin de m'occuper de mes clients.

Jean

Je souhaite que mon contrat d'assurance externe au système soit reconnu
afin d'être assuré par une agence externe à BlaBlaMove (mock).

Architecture

L'idée de notre architecture est de découpler la partie métier liée au trajet et à l'assurance. La partie assurance doit être orthogonale à la logique du trajet. Hormis la partie subscription à un contrat et l'ajout de contrat, la logique métier de l'assurance repose sur le monitoring de la logique métier du trajet. Si l'utilisateur respecte toutes les conditions posé par le contrat auquel il a souscrit alors il sera assuré. Le monitoring vise donc à s'assurer que l'utilisateur respecte ces conditions.

Pour assurer ce découplage, les deux logique métier communiquent par un queue de message. Ainsi dès qu'une informations est mise, un message est envoyé dans la queue de message. Ce message est ensuite consommé pour mettre à jour la partie assurance de l'application.

Pour l'instant, la partie assurance va soit valider le trajet qui lui est envoyé, soit envoyer une notification de fin de trajet à la compagnie conrrespondante.

Composants

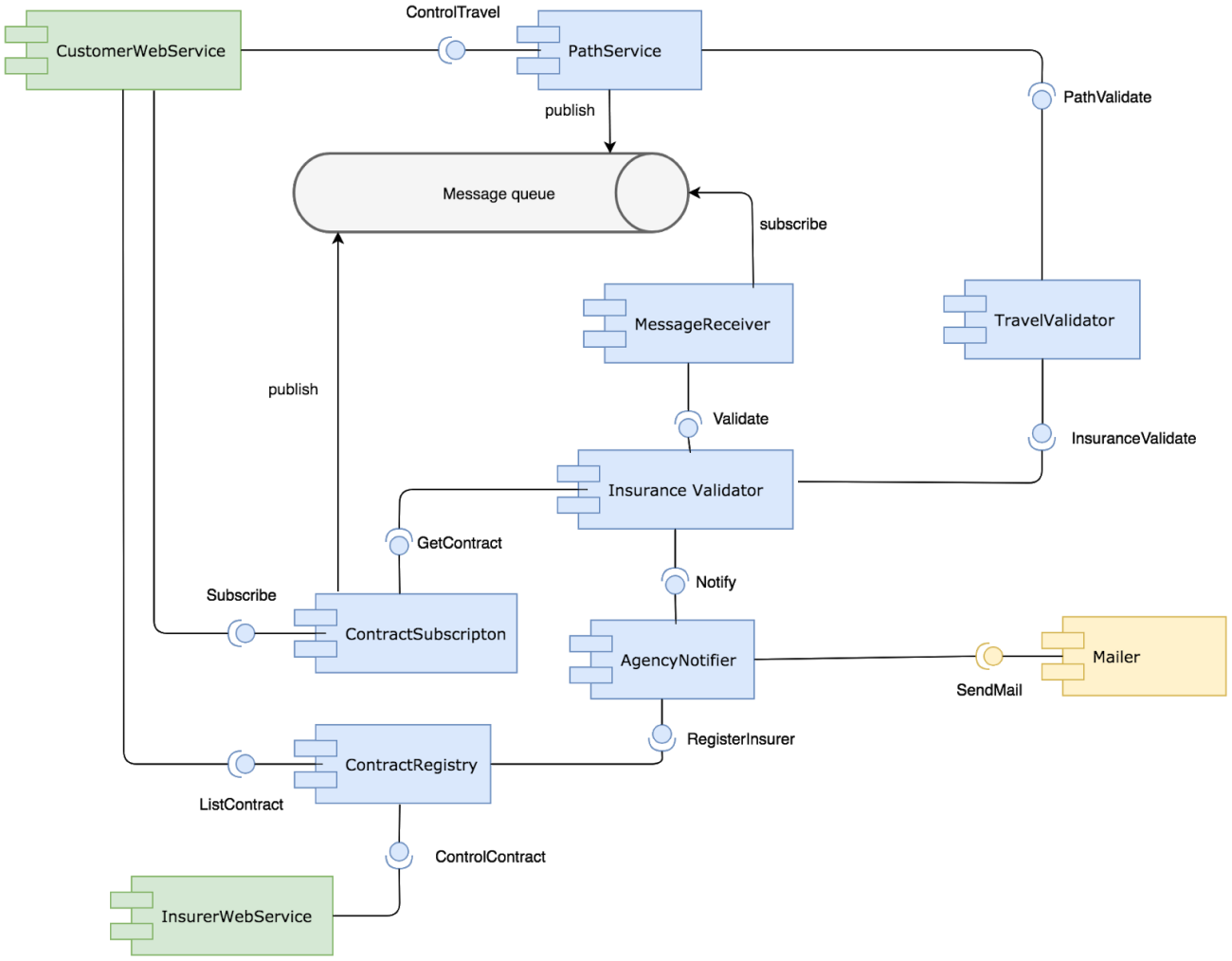


diagramme de composant

Contract Instance

Le module Contract Instance s'occupe des contrats actifs entre l'assureur et les clients. Il propose une interface *Subscribe* qui permet à Celine, Julien et Hubert de renseigner les assurances qu'ils choisissent. Lorsqu'un voyage finit, les différentes informations seront reçu depuis l'interface *FinishContract*. Le module s'occupe de transmettre le compte rendu du déplacement aux différentes assurances via l'interface *Notify*.

Contract Registry

Ce composant a pour rôle de lister les contrats proposés par les agences trié par catégorie. Lorsqu'un nouveau contrat lui est soumis, il va transmettre au composant *Agence Notifier* la façon dont celui-ci doit le contacter.
Les contrats que manipule ce composant sont différents des contrats de *Contract Instance* car ce sont ici les propositions faites par les agences, alors que dans *Contract Instance*, on parle plutôt d'une souscription à un contrat.

Agence Notifier

Ce module a la responsabilité d'informer les assurances du déroulé des trajets. Pour cela, chaque informations sont transmises avec l'interface *RegisterInsurer.* Le module proposera grâce àl'interface *Notify* un moyen de signaler au composant que l'agence doit être contacter.

Path Service

Ce module s'occupe de simuler le trajet des transporteurs et gérer les points associer à ces trajets. Étant donné que ce module est nécessaire au système mais n'est pas la partie principale de notre variante de projet, il sera *mock*.

Les composants *Client* et *Insurance Client* sont les clients de notre système.

Message Consumer

Ce composant va consomer les messages envoyer dans la *message queue*, en fonction du message il va le rediriger vers le composant adéquat. Pour l'instant il ne le redigire que vers *Insurance Validator*

Insurance Validator

Ce composant va se charger de valider ou d'invalider un trajet en fonction des contraintes associées aux participants du trajet. Si le trajet est valide alors une notification est envoyée.

Travel Validator

Il fait office de pont entre la logique métier de trajet et celle d'assurance. Une fois que les deux ont validés le trajet, alors le trajet est validé.

Interface

ControlTravel

```
Travel createTravel(String customerName, String departure, String destination);
Travel addItemToTravel(Item item, String travelId);
List<Travel> findTravel(String departure, String destination);
List<Travel> findTravel();
Travel findTravelById(String travelId);
Travel chooseTravel(String transporterName, String travelId);
void finishTravel(String travelId);
```

PathValidate

Travel pathValidate(Travel travel);
Travel pathInvalidate(Travel travel);

InsuranceValidate

Travel insuranceValidate(Travel travel);
Travel insuranceInvalidate(Travel travel);

Validate

void validate(Travel travel);

Subscribe

ContractSubscription subscribeToContract(Customer customer, Contract contract);
void cancelSubscritpion(ContractSubscription subscription);

GetContract

Collection<ContractSubscription> getContract();
Collection<Contract> getContractByCustomer(Customer customer);

HandleContract

Contract addContract(Type type, String description, String mail);
Contract updateContractDescription(int id,String description) throws
NoSuchContractIdException;
void clear();

ListContract

Collection<Contract> getContractByType(Type type);
Contract getContractById(int id) throws NoSuchContractIdException;

Chemin REST

POST /travels	Création d'un travel
GET /travels	Lister les trajets possible
GET /travels/:travelId	Trouver un trajet
PUT /travels/:travelId	Ajout d'un objet à un trajet
DELETE /travels/:travelId	Notifier de la fin d'un trajet

GET /contracts	Lister les contrats
GET /contracts/:typr	Lister les contrats par type
POST /contracts	Souscrire un utilisateur à un contrat
GET /subscribe	Lister les souscription
POST /subscribe	Signer un contrat
PUT /subscribe	Annuler une souscription
POST /cont	Ajouter un contrat
PUT /cont	Mettre à jour un contrat

Technologies

Pour l'instant, nous nous dirigeons vers un stack technologique java c'est-à-dire JEE, tommee, JPA... similaire à ce que nous avons utilisé l'année dernière en ISA.

- On connaît déjà
- On n'a pas de gros point de complexité nécessitant une technologie particulière.
- Les composants EJB amènent une clarté et un découplage (dû à l'injection de dépendance) qui est intéressant dans notre cas.

JEE : -> contraintes de codage en plus, par contre on se dégage la responsabilité de faire communiquer nos serveurs, on gagne les transactions, (ça peut etre compliqué à gérer à la main)

-(DOCKER pour que nos composants EJB puissent s'exécuter sur n'importe quelles machines)
-(Gattling pour nos tests de charge)
- Peut-être une librairie pour nos scénarios, sinon des requêtes avec curl

Répartition

On voit 2 manières de faire :

- par composant :

Nous avons 4 composants, chacun de nous aura la responsabilité d'un composant. Dans le but que tout le monde puisse voir tous les modules, chacun de nous fera des tests d'intégration sur les composants qu'il n'a pas développé.

- par story:

On a une livraison dans 1 moins, pas de nécessité de livrer des features en continue.

Changements apportés

Changement des contrats d'assurances

Nous avons changé notre façon de représenter notre modèle assurance en y ajoutant des polices d'assurance qui sont des clauses de contrat. Cet ajout nous sera utile pour comparer des polices assurance et ainsi trouver l'assurance la plus adéquate aux recherches des clients.

Comparaison entre polices d'assurance

Nous prévoyons de mettre en place un comparateur qui à travers le choix d'une configuration de clauses nous retourne un prix. Ce comparateur sera visible à travers une interface graphique qui propose une liste de contrat que l'on peut sélectionner. A la selection d'un contrat, il est possible de cocher les clauses que l'on souhaite avoir et ainsi récupérer le devis en temps réel.

Simulateur externe

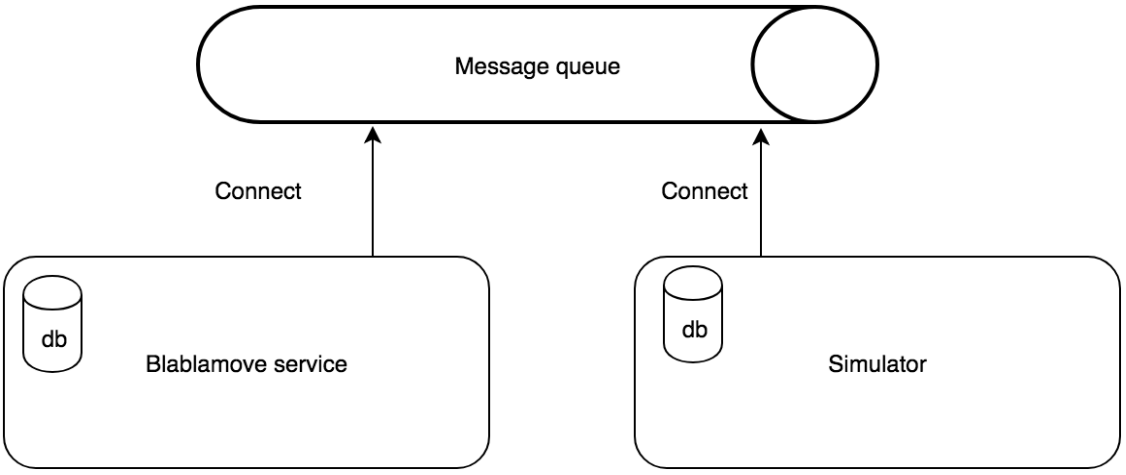
Nous avons à implémenter un simulateur au travers duquel un utilisateur peut entrer les détails de son voyage. Le simulateur devra proposer l'assurance la mieux adaptée.

Pour cela, nous avons décidé d'utiliser la queue de message déjà présente dans notre architecture. Le comparateur va se connecter à cette queue et écouter les événement d'ajout et de suppression d'assurance. Ainsi il va avoir à sa disposition la liste de contrat nécessaire à la sélection de la meilleur garantie.

Une alternative est de transmettre la liste de contrat par requêtes directes entre le service principale blablamove et le simulateur. Le soucis de cette solution est qu'elle crée du couplage entre les deux services et que le simulateur est totalement dépendant du services principales.

Une autre solution encore est de récupérer directement la liste des contrats par requête sur la base de donnée de blablamove. Cependant cela ne ferait que déplacer le couplage et un changement de structure de la base de donnée impacterait tous les services.

L'avantage de notre solution est donc l'indépendance des deux services. Le simulateur possédera sa propre base de donnée et fonctionnera sans que le service blablamove soit joignable.



Schémas de l'intéraction entre le service BlaBlamove et le simulateur

Changement de langage

Nous avons de décidé de ne plus développé en J2EE et de remplacer le code java déjà présent. Nous le faisons pour gagner 3 points.

Pour remplacer J2EE nous avons choisie de développer en Typescript. C'est langage qui se transpile vers du javascript et qui rajoute du typage par rapport au javascript vanilla.

En choisissant Typescript on obtient donc tous les avantages du développement javascript et de Node.js. A savoir un moteur qui peut supporter des montées à charges de part son système d'entrée sortie non bloquant. Mais aussi une communauté active qui permet d'avoir accès à beaucoup de librairie différente.

Le typage qu'il apporte nous permet en effet de garder les interfaces déjà élaboré et les avantages de la programmation par interface. On peut ainsi conserver le découplage entre nos composants.

L'environnement Node.js ne fournit pas autant d'outils que Java EE pour le développement de serveur web. Il faut donc choisir des librairies externe afin de palier à cela. L'environnement npm comporte de nombreux modules dont on ne peut pas vérifier la qualité, pour cela nous nous documentons avant de les utiliser.

Nous avons utilisé Express.js pour définir l'interface réseau de notre service. Il nous permet de mettre en place une interface REST assez rapidement. C'est une libraires relativement vieille et bien documenté. De plus elle est utilisé par des entreprises et des applications professionnelles comme Netflix.

Comme ORM nous avons choisie typeORM. Cette librairie s'interface avec différentes base de données, notamment MySql, et permet la définition des données par annotation dans un style similaire à JPA.

Nous avons fait le choix de garder la même structure pour la base de donnée SQL qui a été extrait de notre implémentation en J2EE. Cela nous permet d'avoir accès au système de transaction qui est important lors de la gestion des souscription des clients et la gestion des assurances.