CSC5002 - Rapport sur le micro-projet: VLibTour Application

Eloi Besnard - Joshua Randria

October 26, 2022

Abstract

Ce compte-rendu présente le Proof Of Concept technique du back-end d'une application nommée VLibTour. Cette dernière sera dédiée à la visite de Paris à vélo pendant les Jeux Olympiques 2024 à Paris. Les technologies Entreprise Java Beans pour la gestion des entités et la persistence, RabbitMQ pour la communication et d'architectures REST sont notamment utilisées pour mener à bien ce POC.

Sommaire

1 Introduction

Les Jeux Olympiques de Paris 2024 approchant, un POC est demandé pour le back-end d'une application permettant la visite de Paris à vélo. Les visites sont appelées des *tours*, qui sont modifiables par les utilisateurs. Chaque tour consiste en une succession de *POI* (Point of Interest). Les consignes du POC indiquent que les *tours* et *POIs* doivent être modifiables à souhait. Les clients utilisant l'application auront le choix de créer des groupes de visite ou bien d'en rejoindre, afin d'agrémenter leur découverte de Paris.

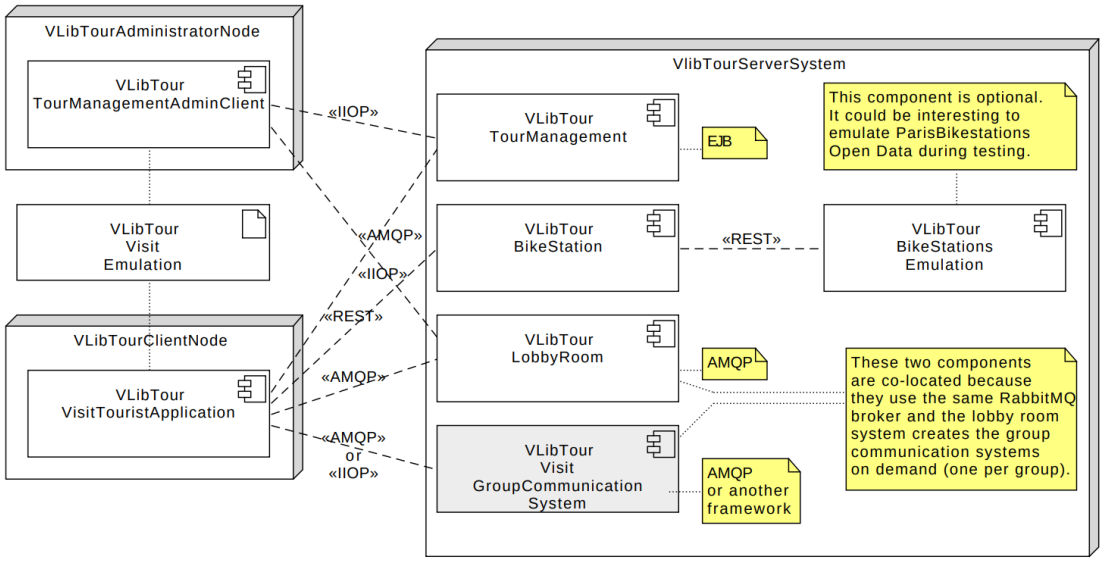
Un projet initial est fourni et est à implémenter pour parachaver le POC. Ce projet contient plusieurs parties qui sont indépendantes. L'intégration de ces différentes parties relève des critères du POC.

Les différentes parties traitent tout d'abord de l'organisation des entités ($POIs$, $tours$,...) et de leur persistence (base de données), à travers $TourManagement$. Elles s'appuient ensuite sur $EmulationSystem$ pour la gestion des positions. La partie $CommunicationSystem$ intervient dans l'envoi et la réception de messages notamment avec une architecure $publish/subscribe$ que nous définirons plus loin. Enfin le *Lobbyroom* permet la création de groupes de touristes voulant effectuer une visite groupée. On peut également rejoindre ces groupes à travers le *Lobbyroom*.

La partie 2. Architectures utilisées rentre dans les détails techniques de chaque implémentation réalisée pour ce POC. La partie 3. Exigences extrafonctionnelles sont simplement des pistes et réflexions sur des exigences présentes en dehors du cahier des charges principal.

2 Architectures utilisées

L'architecture générale du projet est la suivante :



Architecture générale du système

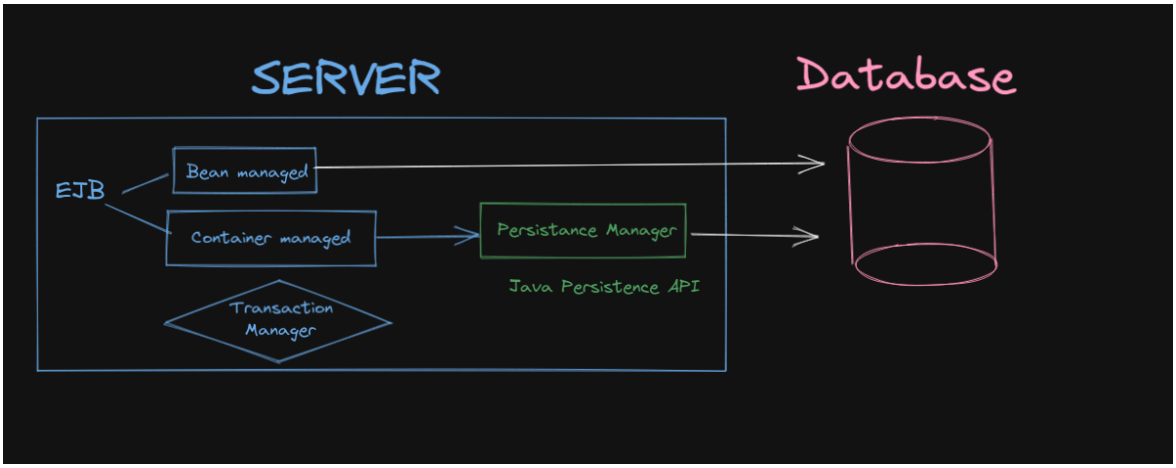
Les différents composants sont détaillés ci-dessous.

2.1 Tour Management System

Dans ce module on se concentre sur la création et la mise en relation des POIs et des tours. Pour cela on définit les règles ou la "business logic" grâce au modèle de composants fournit par Java à savoir les EJB (Enterprise Java Beans). Les différents $EJBs$ sont ensuite mis dans un EJB container qui est deployé dans le server $glassfish$. Cela permet d'avoir une structure standart, garantissant une réutilisation des composants. De plus les sessions sont stateless, ce qui est pertinent puisqu'on lit les données persistantes, et cela améliore également la scalabilité car on ne conserve pas l'état suite à un appel.

Les entités ainsi deployées sont les POIs et les tours ainsi que les fonctions qui les relient comme par exemple $createTour$, $createPOI$, $getTour$,$addPOIToTour$ ainsi que leur liens spécifié avec des $@Id$ ou $@ManyToMany$.

On utilise stocke enfin ces données dans la database en utilisant la JavaPersistence API (JPA).

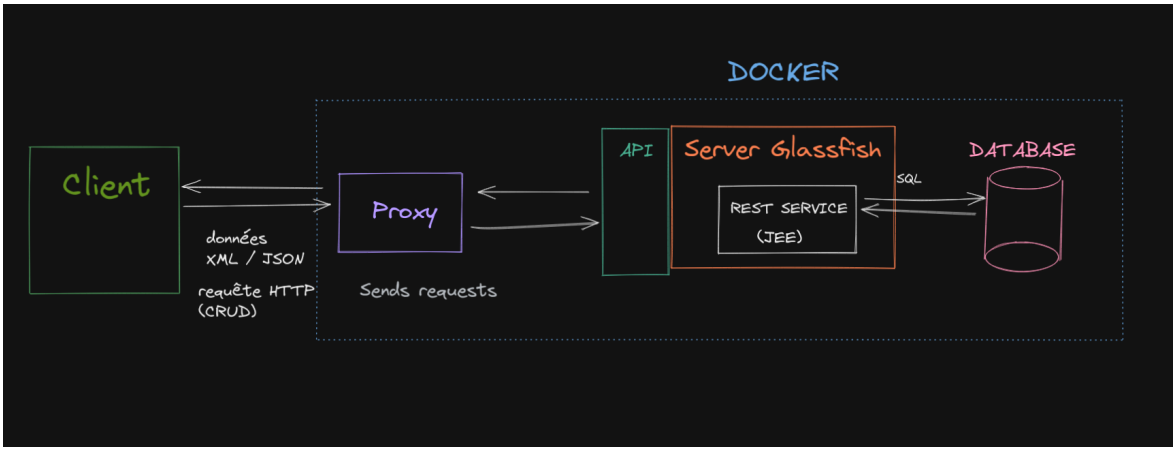


Architecture du Tour Management System

2.2 Emulation System

Ce module permet d'obtenir des positions comme "currentPosition" ou "nextPOI" par rapport à un utilisateur et ainsi de le situer sur la map. En effet, on effectue des requêtes $HTTP$ au server via l'API $REST$. C'est également dans le server que sont initialisés toutes les positions possibles ainsi que les enchaînements entre chacunes d'elles.

Les requêtes sont effectuées à travers un proxy qui implemente les fonctions désirées dans l'interface client, il fait l'intermédiaire en reformattant et transmettant ces requtes au serveur. Le format est par exemple la spécification de la méthode $.get()$ ou encore l'indication du chemin pour accéder à la donnée que l'on cherche, par exemple ("$visitemulation/stepInCurrentPath/" + user$).

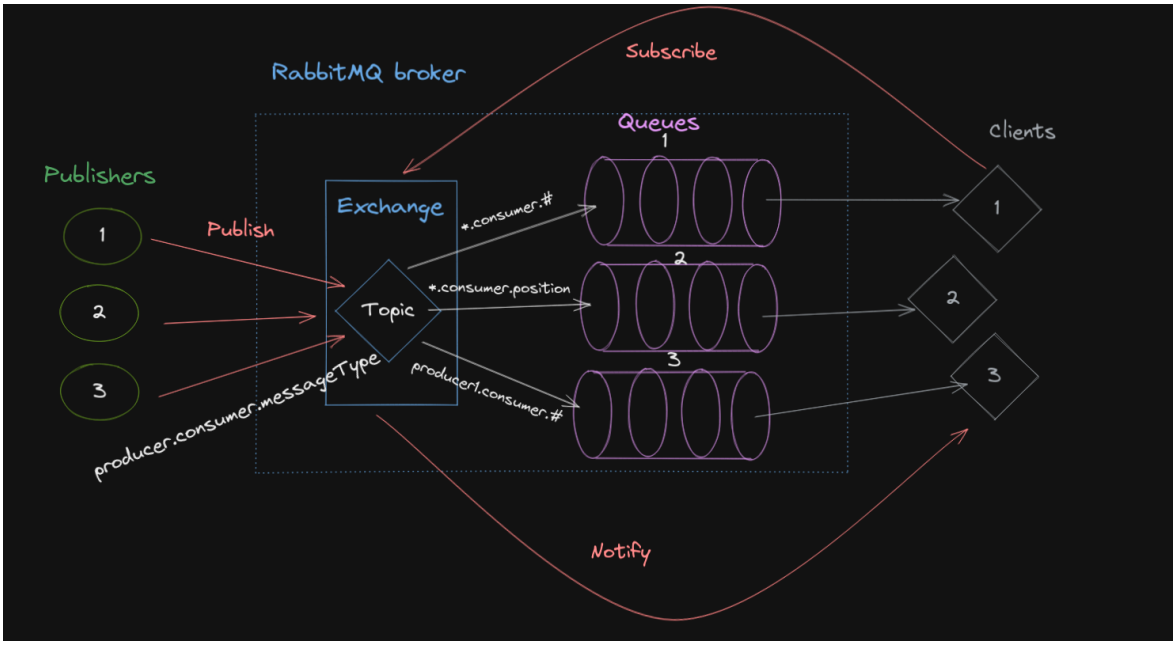


Architecture de l'Emulation System

Les méthodes de l'interface seront ensuite appelées depuis VLibTourVisitTouristApplication dans une boucle while qui permettra de se déplacer de positions en positions, POIs en POIs jusqu'à la fin du tour.

2.3 Communication System

Dans cette partie on s'occupe de la communication entre utilisateurs. En effet, on souhaite qu'à chaque évenement du type $stepInCurrentPath$, $StepsInCurrentVisit$, etc, l'information soit transmise à tous les autres utilisateurs de façon à ce que chacun puisse voir la position de tout le monde en temps réel. On s'appuie sur un topic-based Distributed Event-Based System ($DEBS$) donc avec un système d'abonnement et de publication pour envoyer et recevoir les messages. Pour cela on utilise le standart $AMQP$ que l'on implemente grâce au serveur $RabbitMQ$. On définit ainsi les fonctions pour les producers et consumers à savoir publish et consume, dans un proxy ($VLibTourGroupCommunicationSystemProxy$) auquel les clients accéderont via leur application (ce lien étant fait dans $VLibTourVisitTouristApplication$).



Architecture du Communication System

Pour la sélection et le format des messages on a donc définit la forme suivante pour les $bindingsKeys$ et $routingKeys$: "producer.consumer.messageType". Or ici on souhaite faire un broadcast, donc pour tout le monde, de position. Avec cette simplification on obtient la forme "$userId.all$.". Ce message est envoyé au broker $AMQP$ qui va ensuite distribuer le message à toutes les queues qui sont abonnées à ce type de message. Ces messages seront ensuite consumés au fur et à mesure via la méthode $handleDelivery$. Dans notre cas on s'intéressera à la récupération du producer et la position pour afficher cette position sur la map de l'utilisateur associé.

Les messages sont stockés dans la queue jusqu'à ce que l'utilisateur se connecte à l'application, ce qui permet d'un côté au producer de publier sans dépendre de la disponibilité du consumer, et de l'autre côté au consumer de recevoir les messages dès qu'il est prêt. Le traitement des messages se fait au fur et à mesure ce qui permet de réguler la charge de travail de l'application.

2.4 Lobby Room

Ce module permet aux utilisateurs de communiquer avec le lobby room pour créer et/ou rejoindre un tour/groupe. Les communications sont assurées de manière similaires à celle utilisée dans CommunicationSystem c'est à dire grâce à $RabbitMQ$ et l'architecture $AMQP$.

Cependant plutôt qu'un topic-based system, on utilise un server RPC pour Remote Procedure Call, afin d'avoir un échange direct entre le client et le lobby. On créé ainsi un $jsonRPCClient$ géré par le proxy, qui fait son appel dirigé dans la queue associé au lobby qui une fois traitée, renvoie sa réponse à l'exchange qui transmets directement au client.

En plus de l'ULR spécifique renvoyé par le lobby, on ajoute un système d'authentification, en guise de couche de sécurité, en utilisant un virtual host qui possède un identifiant unique, garantissant l'unicité du groupe, auquel les clients doivent se connecter en utilisant l'url envoyé et le mot de passe généré. Cela se fait via $rabbitmqctlset_permissions$, qui spécifie les user ainsi que leurs mot de passe

3 Exigences extrafonctionnelles

3.1 InterOperabilité

L'interopérabilité est une condition essentielle de la partie MiddleWare. Premièrement, c'est l'existance de nombreux autres langages de programmation et systèmes d'exploitations qui impose de l'interopérabilité. L'envoi de données peut différer à travers des protocoles réseaux différents ou des standards d'encodage variés (*little Endian*, *big Endian*). Pour y faire face, il existe de nombreuses méthodes. D'abord, écrire son code de manière $top - down$. Le but est d'avoir une vision globale des interactions entre chaque module et puis progressivement rentrer dans le détail des implémentations. Ensuite, de manière plus précise, il s'agit de garder les structures de données relativement simples. Ainsi, elles auront beaucoup moins de mal à être réutilisées du côté client. S'imaginer également que les types qu'on utilise peuvent ne pas être casté comme on l'aurait voulu par manque de précision sur un autre support. Il est aussi important de rester vigilant sur l'utilisation de variable pouvant être *null*. Dans notre implémentation, en respectant le standard AMQP, cela garantit un minimum d'interopérabilité que nous aurions perdue sans ce standard.

3.2 Mise à l'échelle

Les serveurs étant stateless, il n'y a pas besoin de repliquer les états des sessions des clients (EJB par ex). Ainsi, cela permet à n'importe quel serveur de traiter des requêtes sans condition d'"historique". Glassfish effectue également l'affectation aux CPU en fonction des requêtes, ce qui permet d'assurer de la mise à l'échelle.

4 Conclusion

Nous avons donc programmé les différentes parties de ce projet et réalisé l'intégration de ces dernières.

Le POC décrit dans ce compte-rendu est fonctionnel et les architectures qu'il implémente lui confèrent un minimum d'interopérabilité et de mise à l'échelle.

L'intégration est une partie délicate qui nécessite un effort de synthèse et une aisance avec l'ensemble du projet. Cela s'est révélé être un bon exercice pour prendre du recul sur le code et le voir sous un paradigme global.

Des améliorations à apporter à ce projet seraient d'implémenter les conditions extrafonctionnelles qui pourraient lui apporter de la robustesse et de la sécurité. Egalement un front-end pourrait permettre à l'utilisateur de ne pas se perdre et dans l'appli et dans Paris.