Polympic – Projet PS7

EQUIPE AL2
PASSAGE A L'ECHELLE
EXTENSIBILITE ET INTEROPERABILITE

BISEGNA DAVID
DEMOLLIERE COME
MONTOYA DAMIEN
PERES RICHARD

Année universitaire 2019/2020

Table des matières

Introduction

Les Jeux Olympiques à Paris auront lieu en 2024 et rassembleront un grand nombre de personnes venant du monde entier. Paris se veut exemplaire en mettant l'accent sur la sécurité et l'organisation.

C'est sur ces aspects que notre application, Polympic intervient. Nous voulons mettre en avant les mouvements de foule afin que des organisateurs d'événements puissent prévenir d'éventuels incidents. Toute personne pourra donc utiliser l'application Polympic afin de se localiser.

De plus, une API ouverte est disponible pour que n'importe quelle entité puisse suivre des personnes spécifiques afin d'analyser leurs déplacements ou même que n'importe quelles agences de sécurité se recensent afin qu'elles soient prévenues lorsqu'un incident est recensé dans leur zone d'intervention.

Enfin, nous souhaitons inciter les utilisateurs à utiliser notre plateforme en mettant en avant des événements partenaires que les organisateurs peuvent créer et tenir à jour. Bien sûr d'autres agences d'événements pourront recenser leurs événements au sein de Polympic afin d'enrichir son contenu.

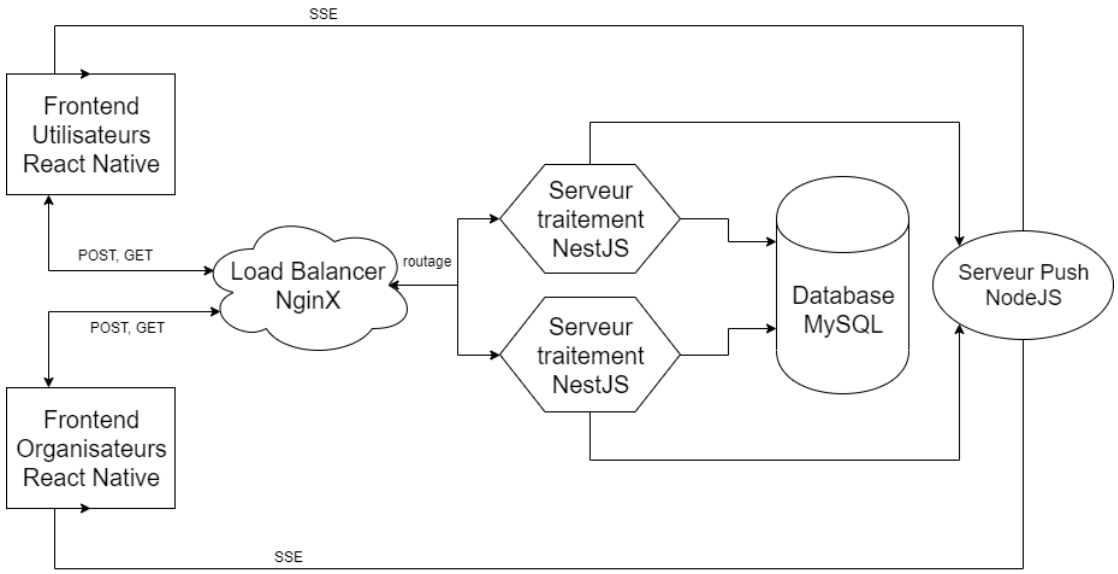
Défis associés au projet

LES DEFIS DU PASSAGE A L'ECHELLE

Une des attentes fondamentales du client concerne le passage à l'échelle de l'application et donc le maintien des performances peu importe le nombre d'utilisateurs ou de services externes connectés en simultané. Pour réaliser cet objectif nous avons défini que le critère principal serait la minimisation du temps de réponse moyen aux utilisateurs. Notre application se doit donc d'être capable de répondre le plus rapidement possible aux demandes de tous les utilisateurs en un temps minimum. Pour correspondre aux attentes des clients nous nous sommes donc assurés que les trois points suivants soient respectés.

Permettre aux utilisateurs d'échanger des données rapidement

L'objectif principal de l'application étant de pouvoir suivre la position des utilisateurs, nous avons dû mettre en place une certaine architecture pouvant supporter plusieurs milliers de traitement de données en parallèle. Après avoir effectué diverses recherches nous avons appliqué la solution suivante :



Le schéma d'architecture du projet est plutôt simple à visualiser. Nous avons en partie client des utilisateurs disposant d'un smartphone et possédant l'application Polympic développée en *React native*. Pour assurer leurs requêtes, les clients contactent le *load balancer* qui se charge alors de les répartir sur les différents serveurs de traitement. Le *load balancer* a donc pour but de router les clients et ainsi déléguer les calculs à d'autres serveurs pour répartir au mieux la charge de travail.

Dans cette configuration nous pouvons constater la présence de deux serveurs de traitements, ici développés via *NestJS*. En réalité nous pouvons en instancier autant que l'on veut dans la limite des ressources de la machine pour ainsi améliorer les performances et accueillir plus d'utilisateurs.

Nous avons également mis en place un serveur de push qui permettra d'envoyer aux utilisateurs des données en temps réel, dans notre cas la position des utilisateurs aux organisateurs et la modification des événements à tout le monde. Suite à nos recherches nous avons choisi d'utiliser une méthode de connexion SSE pour envoyer les données aux utilisateurs. En effet, pour récupérer les positions et les événements, l'application se met sur écoute et seul le serveur envoie des données. Nous avons donc une connexion unidirectionnelle. En mettant en place une socket, la connexion aurait été bidirectionnelle mais utilisée seulement dans un sens ce qui n'aurait donc pas été pertinent.

Lorsque les serveurs de traitement se lancent, le serveur de push se connecte également par SSE à tous les serveurs de traitement lancés. Ensuite, lorsqu'un utilisateur ou un organisateur ouvre l'application, cette dernière se connecte au serveur de push via une requête GET. Le fait de connecter le serveur de push aux serveurs de traitement permet de déléguer le traitements et calcul des données de l'envoi de ces dernières aux utilisateurs.

De ce fait, grâce à cette architecture, lorsqu'un utilisateur met à jour sa position un POST est donc envoyé au *load balancer* qui lui va redistribuer cette requête à l'un des n serveurs de traitements, en se basant sur l'algorithme round robin. Ensuite cette requête sera transmise grâce à une requête de type SSE vers le serveur de push (que l'on appellera Publisher) et enfin ce serveur retransmet cette position associée à l'utilisateur à tous les organisateurs connectés.

La vérification du passage à l'échelle

Le passage à l'échelle étant une attente importante pour le client nous avons cherché plusieurs solutions pour vérifier qu'en cas de forte utilisation le système serait tout de même en mesure de traiter les demandes des utilisateurs dans un délais raisonnable.

Durant notre sprint 1, nous avions une architecture différente de celle présentée ci-dessus. En effet, c'était une architecture client-serveur simple qui permettait une connexion via des requêtes et des sockets. Les requêtes avaient le même but que dans notre architecture finale, pourvoir récupérer les listes d'utilisateurs et d'organisateurs lors du chargement de l'application mais également de permettre aux services externes d'accéder à l'API. Les sockets quant à elles permettaient d'envoyer les positions pour la partie utilisateur, et de récupérer ces mêmes positions pour les organisateurs, ce qui était très pratique pour avoir cet effet temps réel au sein de l'application.

Pour pouvoir tester les requêtes nous avons dans un premier temps utilisé *Gatling*. Ce logiciel nous a permis d'effectuer des stress test et a démontré que notre système pouvait accueillir aux alentours de 500 requêtes en simultané avec un temps de réponse de moins d'une seconde. Pour tester les sockets nous n'avons cependant pas réussi à configurer *Gatling*. Nous avons donc utilisé un *node* module, *Artillery* qui nous a permis d'effectuer les stress test. Cet utilitaire a démontré que l'architecture pouvait supporter 500 sockets en simultané et avait un temps de réponse d'environ une demi seconde. Les sockets étaient donc plus performants lorsqu'il y a beaucoup d'échange entre le client et le serveur. Cependant, cette architecture s'est révélée assez problématique pour le passage à l'échelle. En effet, le problème des sockets est qu'elles sont figées entre un serveur et un client donc on se serait retrouvé avec un système se rapprochant d'une *sticky session* avec un *load balancer* jouant plutôt un rôle

d'orchestration, en indiquant à quel serveur le client devait se connecter. Donc si par la suite on voulait mettre en place une gestion dynamique des serveurs, il aurait fallu à chaque fois que l'on souhaite répartir la charge indiquée, et donc déplacer les utilisateurs d'un serveur à un autre leur indiquer de couper la connexion avec la socket actuelle et en créer une nouvelle avec le nouveau serveur en question. C'est pourquoi, nous nous sommes plutôt tournés vers une architecture plus classique avec un *load balancer* mais basé sur une API Rest.

Pour tester cette nouvelle architecture nous avons utilisé l'utilitaire Locuste qui permet contrairement à *Gatling* de voir les tests en temps réel et non pas seulement à la fin. Ces stress tests ont confirmé l'amélioration des performances avec la nouvelle architecture qui permet maintenant grâce à 3 serveurs de traitements de répondre à 30.000 requêtes en simultané en moins de 300 millisecondes en moyenne. Les tests sont effectués sur l'une de nos machines personnelles et donc on a potentiellement moins de performance que ce que l'on pourrait avoir avec un serveur spécifique.

Permettre à l'API d'interagir avec des systèmes externes

Grâce à notre architecture finale nous pouvons permettre aux services externes d'accéder à notre API de manière fluide en supportant plusieurs milliers de requêtes en simultané. En effet, tout service voulant envoyer ou recevoir des informations devra passer par le load balancer qui va les répartir vers un serveur de traitement. Ce principe étant le même que pour les utilisateurs de l'application, l'API externe restera donc aussi performante que le système interne.

Dans le cas de l'API externe, la seule différence avec le système interne est que cette dernière ne répond pas avec le serveur de push mais directement au client en retour de requête.

LES DEFIS DE L'EXTENSIBILITE ET D'INTEROPERABILITE

Le but de cet axe est de rendre notre projet extensible, en autre terme, utilisable et modulable pour de nombreux services externes. Pour l'interopérabilité, il faut mettre en avant la communication entre notre système et des systèmes externes, on peut envoyer des données vers un système externe et nous pouvons recevoir des données venant de systèmes externes.

Permettre à des services extérieurs de se connecter à notre application

Afin de permettre à des services d'intervention tel que les Pompiers ou le Samu d'intervenir facilement en cas d'incidents nous avons fait en sorte que ces derniers puissent se recenser, en passant par l'API, en tant que personnes à prévenir en cas d'urgence.

Pour ce faire nous avons donc mis une route HTTP spécifique de type POST sur laquelle chaque service de sécurité peut envoyer leur donnée tel que leurs localisations. De plus, ce service devra nous indiquer la route HTTP sur laquelle il souhaite recevoir les divers incidents. Nous stockons donc toutes ces informations dans notre base de données.

Ainsi à chaque incident, nous prévenons toutes les agences de sécurités par le biais d'un POST sur leur route spécifiée au préalable. Par la suite, lors de futur sprint, nous permettrons à ces services de nous donner une zone d'intervention ainsi nous préviendrons que les services

intervenant dans la zone de l'incident. Cet aspect touche à l'extensibilité puisqu'on permet à n'importe quel service externe de se connecter à nous en nous proposant leur propre route sur laquelle il souhaite recevoir les informations. Cependant, si une erreur survient au moment du POST, par exemple si l'adresse est introuvable, alors nous n'effectuons juste pas ce POST. On pourrait supposer que par la suite un système de mails soit mis en place pour prévenir le service en question lorsqu'on rencontre un problème avec la route qu'il nous a spécifié.

Nous avons effectué le même principe pour des services qui souhaiteraient suivre un utilisateur spécifique durant une durée spécifique afin de pouvoir la surveiller par exemple. Il suffit à ces services de demander à suivre une personne en passant son identifiant, la durée du temps de suivis en minutes (nous avons fixé un maximum à 30 minutes) ainsi que la route sur laquelle nous allons pouvoir envoyer les informations lors du déplacement de l'utilisateur. De ce fait, à chaque fois que l'utilisateur fait envoie un déplacement alors le service le suivant recevra par le biais d'un POST, sur la route informé ultérieurement, la nouvelle position de l'utilisateur.

Des services extérieurs doivent pouvoir nous envoyer du contenu

Nous souhaitons avoir beaucoup de données concernant la localisation de personnes afin de permettre aux organisateurs d'avoir une vue plus concrète du nombre de personnes présentes et de leurs mouvements. Pour ce faire, nous permettons a d'autre service, utilisant des données de localisation de leurs utilisateurs, de nous envoyer leurs données de manière régulière afin que nous puissions par la suite retransmettre ces données au divers organisateurs.

De plus, nous proposons le même principe pour les événements afin que les utilisateurs et organisateurs puissent consulter des événements venant de source externe. Actuellement, nous n'avons pas mis en place le fait que la création ou la modification d'un événement, venant d'un service externe, apparaissait dans des temps raisonnables sur le page des utilisateurs, c'est donc à l'utilisateur de demander à voir les événements externes afin de pouvoir consulter qu'elles sont les événements au moment de la demande. Dans des sprints futurs, nous ferons, comme pour les événements créés par les organisateurs, ils seront mis à jour par le biais du serveur Publisher directement lors d'une modification.

Ces ajouts de données se font par le biais de notre API ouverte. Il suffit d'effectuer une requête de type POST vers la route correspondant au type de données que l'on souhaite ajoutés.

Bien sûr, nous souhaitons pouvoir permettre à de nombreux services de nous envoyer leurs données et ce de manière concurrente. C'est pourquoi, notre architecture supportant un nombre important de requêtes se doit d'être aussi ici performante. Nous avons donc testé cette spécificité par le biais de stress tests en simulant en nombre important de données envoyé par plusieurs services en parallèles.

Nous avons également mis en place une route ouverte permettant de signaler un incident à une position précise pour ne pas être obligé de passer par l'application Polympic pour pouvoir signaler un incident. Ainsi, cet incident sera retransmis à tous les services d'intervention recensés. Il s'agit également ici une route de type POST.

Récupérer du contenu venant de services extérieurs

Nous nous comme un peu moins penché sur cet aspect lors des deux premiers sprints mais il pourra être ajouté lors de futur sprint. Actuellement, nous utilisons une API externe : *Google Map*, nous permettant d'afficher la carte sur laquelle nous représentons les utilisateurs ainsi que les événements.

Cependant, nous aimerions bien lors de sprint futur mettre en place la possibilité d'aller chercher du contenu, par exemple événementielle, directement chez des services externes en utilisant leur API. Cela impliquerait donc de mettre en place des patrons de conceptions de type Adapter permettant d'adapter les données qu'on reçoit en donnée interprétable par nos services et stockable dans notre base de données. Ceci nous permettrait d'être plus extensible afin d'avoir seulement l'adapter à modifier si le service sur lequel nous nous branchons change le format de leurs données.

Analyse critique du produit

ANALYSE DE NOTRE ARCHITECTURE

Notre architecture finale nous permet d'avoir un passage à l'échelle au niveau des serveurs de traitement. On peut donc allouer autant de serveur que l'on veut par rapport au budget, aux ressources allouées et à la quantité de requête en parallèle que l'on souhaite pouvoir supporter. Cependant, actuellement nous rencontrons un goulot d'étranglement au niveau du serveur de push qui actuellement est seul et donc peut potentiellement se retrouver saturé même si le traitement qu'effectue ce serveur est minime, puisqu'il fait que de la retransmission. Cela pourrait surtout se ressentir sur les événements, puisque le serveur de push doit retransmettre les modifications événementielles à tous les utilisateurs, si on a un nombre d'utilisateurs trop important pour ce dernier le temps de latence peut devenir assez important.

Par la suite, on pourrait donc mettre en place un deuxième load balancer au niveau du serveur de push qui ferait la même chose que pour les serveurs de traitement mais cette fois ci avec les serveurs de push, on aurait ainsi un passage à l'échelle sur cet aspect-là.

Un autre goulot d'étranglement pourrait se situer au niveau de la base de données. Actuellement, on a qu'une seule base de données pour la production qui est partagé pour tous les serveurs de traitements. Cette dernière peut donc être également un point de blocage puisqu'elle admet un seuil au niveau des requêtes qu'elle est capable de traiter parallèlement. Il faudrait donc aussi ici trouver une solution pour pouvoir être encore plus sujet au passage à l'échelle avec potentiellement une base de données spécifique à chaque serveur. Cependant, cela pourrait potentiellement poser des problèmes au niveau des données partagée. Par un exemple si un événement est stocké dans une base de données d'un serveur spécifique et que depuis un autre serveur on veut avoir les infos de cet événement il faut être en mesure de pouvoir aller le récupérer.

Nous avons donc un système capable de lancer beaucoup de serveurs de traitement. Cependant, le nombre de serveurs à lancer se fait au moment du lancement global, actuellement nous ne pouvons ajouter ou éteindre des serveurs dynamiquement mais ce que nous pourrions faire par la suite est d'améliorer ça en optimisant les ressources allouées. Pour ce faire, il faudrait que le *load balancer* allume des serveurs que si on en a l'utilité, en cas de pic d'activité par exemple, et en inversement en éteindre si le nombre de requêtes à traiter est suffisant bas. Cela nous permettrait de réduire les coûts ainsi que d'alléger le système. De plus, ça n'impacte pas notre architecture actuelle.

ANALYSE DE L'API MISE A DISPOSITION DES SERVICES EXTERNES

Notre API mise à disposition des services externes est plutôt complète puisqu'elle permet de nombreuses fonctionnalités comme signaler un incident, se recenser en tant que service de sécurité, suivre un utilisateur pendant un temps donné ou même nous envoyer des données utilisateur ou événementielles. De plus la documentation associée disponible sur la route /api générée grâce au plugin *swagger* permet aux services ou développeurs voulant

utiliser notre api de savoir comment l'utiliser et quels types de données sont attendus ou reçus.

Cependant, sur l'aspect d'extensibilité, comme vue précédemment, nous aurions pu plus mettre cet aspect en avant en allant chercher nous-mêmes des données chez des services externes et ainsi mettre en place des Adapter pour pouvoir facilement suivre et s'adapter aux modifications que le service sur lequel nous nous branchons pourrait apporter. Actuellement on dispose des couches *controller*, *service* et *repository* mais il faudrait qu'avant de rentrer dans la couche du service on rentre dans une couche adapter qui s'occupe donc d'adapter les données pour que le service puisse les traiter facilement.

Une autre idée d'amélioration concernant les API mais aussi le passage à l'échelle serait de faire en sorte que toutes les requêtes venant de l'API externe soient traitées de manière moins prioritaires que celles venant de notre système afin de prioriser nos utilisateurs et leurs conforts dans l'utilisation de notre application. On pourrait même dissocier la partie API externe de la partie API interne avec des serveurs de traitement différent et un *load balancer* qui leur est propre mais avec quand même une base de données partagées pour pouvoir récupérer les données entre les différents serveurs.

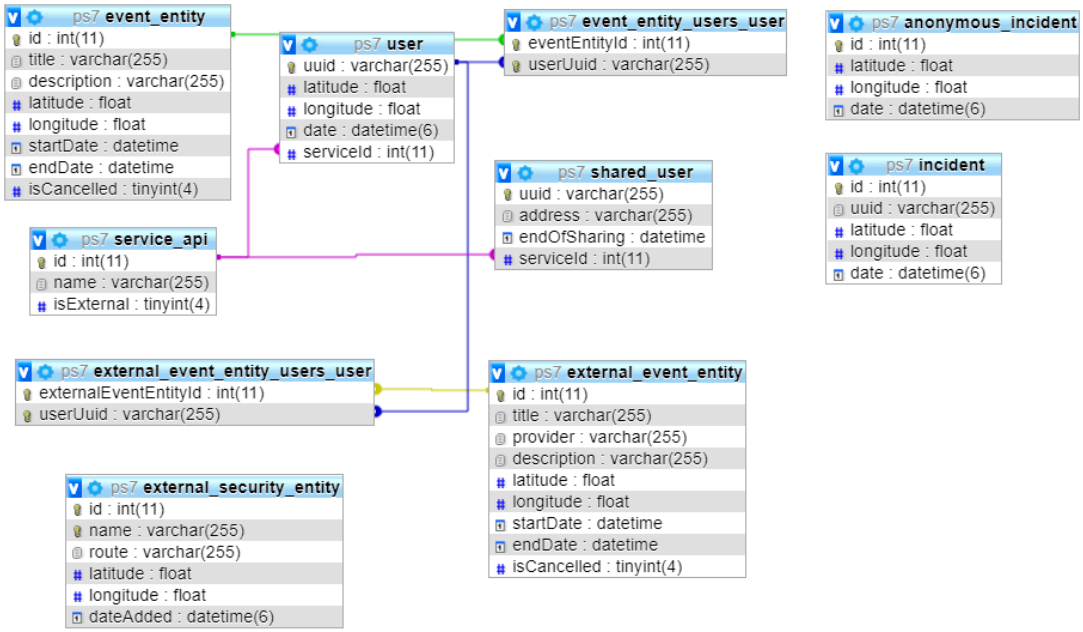
Positionnement scientifique

PS6

Si nous avons pu mettre en place rapidement le backend de Polympic c'est grâce au projet de semestre 6 que nous avons eu l'année dernière. En effet, dans ce projet nous avions développé une application ayant pour backend un serveur node.js et un frontend e n *Angular*. PS6 nous a donc appris à mettre en place un serveur node.js mais également le développement en utilisant le langage *TypeScript*. Bien qu'ayant développé la partie frontend de Polympic *en React Native*, le côté backend lui propose un serveur node.js avec des fonctionnalités similaires que celui mis en place lors du projet PS6.

BASE DE DONNEES

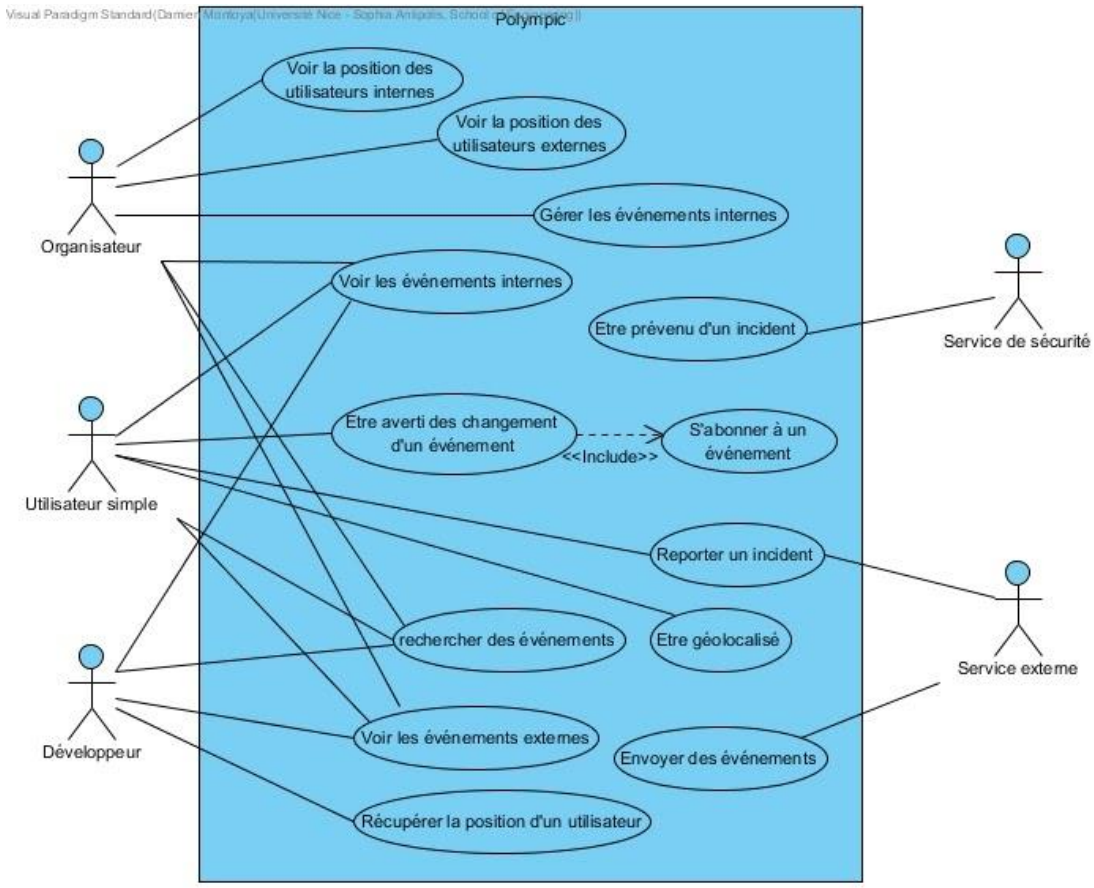
Le cours de base de données suivi en première année nous a été utile pour imaginer la structure de stockage pour Polympic. Nous avons également pu mettre en pratique les requêtes *SQL* étudiées en cours pour récupérer les données voulues. La base de données finale est composée des tables suivantes :



La table *event_entity* correspond aux événements internes à Polympic qui sont ajoutés par les organisateurs. La table *external_event_entity* correspond aux événements externes auxquels Polympic peut accéder. De même pour les tables user et *shared_user* qui correspondent respectivement aux utilisateurs internes et externes. Concernant la gestion des incidents, nous avons établi deux tables, une pour les incidents ajoutés depuis l'application et donc possédant l'identifiant de l'utilisateur concerné ainsi qu'une table pour les incidents dits "anonyme", ceux émis par les apis externes.

CONCEPTION LOGICIELLE

Concernant cette matière, cette dernière nous a aidé dans un premier temps pour effectuer un diagramme de cas d'utilisation. Ce diagramme a permis d'établir les limites du projet et les interactions que les utilisateurs pourraient effectuer. Nous avons également effectué plusieurs diagrammes d'architecture pour arriver au diagramme final montré précédemment. Enfin, cette matière nous a aidée pour la gestion du GitHub et la création ainsi que la maintenance du kanban. Nous avons également pensé à la mise en place d'un pattern adapter dans un troisième sprint pour pouvoir récupérer les événements de plusieurs api différentes.



GESTION DE PROJET

Tout au long du projet nous avons retrouvé des concepts préalablement étudiés en cours. En effet, l'élaboration de sprint et des réunions hebdomadaires avec le client et avec l'équipe de développement est l'un des aspects du cours. La mise en place d'un kanban via GitHub et la détermination du temps de travail pour une tâche sont également des concepts qui nous ont été utiles.

Conduite de projet

SPRINT 1

Nous avons abordé ce projet en commençant par le découper autour de nos axes. Pour ce faire nous avons établi un grand nombre de stories sur lesquelles nous avons défini des critères d'acceptation ainsi que des stories points afin d'évaluer leur difficulté. Puis nous avons évalué à l'aide de notre client et de la méthode *MoSCoW* lesquelles semblaient les plus pertinentes pour lui et les avons placés dans notre Sprint 1. Le premier sprint fût assez compliqué à instancier car nous n'avions pas interpréter le projet de la bonne manière, nous avions pensé que l'aspect événementielle était plus l'aspect du projet le plus important à mettre en avant cependant c'était vraiment l'aspect localisation qui était pertinente pour notre client. Nous avons donc dû recréer des stories en meilleure adéquation avec notre client puis garder les précédentes en tant que *backlog*.

Pour réaliser une story nous créons des tâches, que l'on associe à la story référente et nous indiquons sur cette tâche comment la réaliser et ce qui est attendu techniquement. Chaque tâche comporte des labels pour les repérer et est assignée aux personnes chargées de la réaliser. De plus, nous communiquons également beaucoup à l'oral afin de s'entraider et de se tenir à jour sur ce qui est en train d'être fait.

Durant ce sprint nous avons réalisé 107 points de story.

SPRINT 2

Pour établir les stories à réaliser durant le sprint deux nous avons pu récupérer une bonne part de notre *backlog* réalisé lors du sprint précédent notamment sur les événements.

Nous avons passé une bonne partie de ce sprint à revoir notre architecture *back-end* afin de pouvoir plus facilement répondre au besoin du passage à l'échelle qui était moins mis en avant lors du sprint précédent.

Tout comme le sprint précédent nous avons établi des tâches relatives à chaque stories que nous avions à faire.

Durant ce sprint nous avons réalisé 106 points de story. Il faut savoir que pendant ce sprint nous avons passé une bonne partie du temps à faire des recherches, c'est également un aspect à prendre en compte durant le développement du projet, c'est pourquoi certaines stories valent beaucoup de point car elle implique d'apprendre de nouvelles choses avant de pouvoir la réaliser. En effet, durant ce projet beaucoup de notions nouvelles ont été abordées. Le découpage des tâches et le partage des connaissances était donc un aspect très important du projet.

A la fin de ce sprint nous avons également défini ce qu'il pourrait être fait lors de futur sprint et notamment lors du sprint 3 en prenant soin de mettre d'avoir tout autant de travailler à réaliser dans le sprint 3 proportionnellement au travail effectué dans les deux premiers sprints.

REPARTITION DU TRAVAIL

Dans ce projet nous étions tous très investis, c'est pourquoi le travail a pu être répartis de manière homogène. On s'accorde donc tous la même note en termes de travail personnel fourni pour ce projet.

- BISEGNA David : 100 points
- DEMOLLIERE Côme : 100 points
- MONTOYA Damien : 100 points
- PERES Richard : 100 points

Conclusion

Pour conclure, ce projet fût pour nous tous une expérience très enrichissante. En effet, il nous a amené à utiliser des technologies nouvelles et nous a inculqué une méthode de travail efficace. Même si nous ne sommes pas encore des experts avec un projet scrum nous en avons appris beaucoup et serons capable de réutiliser nos connaissances acquises pour de futurs projets. De plus, ce projet nous a permis de découvrir plus en profondeur la filière architecture logicielle et ainsi acquérir des compétences qui pourrait nous être utile pour la suite dans cette filière.