NOHAR

Geste de savoir

J'aime ma stack !

14 juin 2020

Table des matières

Depuis le mois de janvier, je travaille sur un projet tout à fait exotique pour mon profil : je suis responsable de l'infrastructure et du backend d'un MMORPG indépendant. J'écris ce billet environ à mi-chemin de la sortie de la version pré-alpha, pour faire un point sur l'expérience que j'ai pu accumuler ces 6 derniers mois.

Comme le titre l'indique, j'ai décidé d'articuler celui-ci autour de la *stack* technologique que j'ai choisie et le cheminement que j'ai suivi pour ce faire. Car ce choix est le premier que l'on réalise lorsque l'on se lance dans un nouveau projet, quelle que soit son envergure, et que ses répercussions se ressentent au quotidien. Mais ce n'est évidemment pas le seul aspect sur lequel j'ai des trucs à dire !

Ce billet sera également l'occasion pour moi de vous montrer (ou vous rappeler) :

— Comment aborder un projet qui semble l'Everest avec des moyens limités ;
— La façon de gérer ses priorités dans une startup ;
— Ce que c'est **vraiment** que de concevoir ne serait-ce que le *backend* d'un MMORPG, et peut-être vous inciter à revoir vos ambitions à la baisse si vous comptiez vous lancer dans un tel projet « sur votre temps libre ».

1. Le contexte

Commençons par nous mettre d'accord sur ce dont je parle.

Je ne donnerai aucun nom dans ce billet.
Bien que j'adore ma boîte, mes collègues et mon projet, le jeu dont il est question ici n'a pas encore atteint une phase où l'on désire le faire connaître. De plus, ces informations ne

sont pas essentielles au propos.

1.1. Comment j'en suis arrivé là

L'année dernière, mon travail commençait à avoir réellement de moins en moins de sens à mes yeux, et cela m'a poussé à chercher un job qui me fasse rêver à nouveau. On pourrait résumer mes motivations aux éléments suivants :

— J'avais fait le tour de ma techno,
— Celle-ci montrait ses limites en termes de performances et de passage à l'échelle,
— J'avais bien trop peu de pouvoir décisionnel sur l'aspect technique du projet,
— Mon équipe était noyée sous les petits devs qui n'avaient aucun intérêt pour le produit ou ses utilisateurs, mais *faisaient plaisir* à tel ou tel gros client,
— La *vision produit* par un expert du métier est arrivée cinq ans trop tard, alors qu'il était devenu impossible, politiquement parlant, de tout péter et refaire,
— La boîte était globalement ralentie par une vérité toute bête : trop de projets pour trop peu de moyens humains, avec tous les problèmes organisationnels que cela implique.

Et c'est comme ça que j'ai sauté le pas pour intégrer mon équipe actuelle, une startup qui était alors en train de maintenir un petit jeu en *sandbox* sur mobile, axé sur la socialisation des joueurs. J'ai intégré cette petite équipe en tant qu'*ingénieur backend senior*, mais on verra plus loin ce que ça signifie.

Au mois de janvier, pendant une réunion, les trois fondateurs nous ont exposé notre nouveau projet. Nous allons réaliser un MMORPG à forte dimension narrative qui servira les valeurs de notre boîte. La suite n'est autre que ce qui se passe lorsque l'on expose un projet audacieux et séduisant à une équipe de gens motivés : on a remonté nos manches, refait du café, et on s'est mis au boulot !

1.2. L'équipe

Actuellement, notre équipe compte à peine une douzaine de personnes :

— Le CEO (président), qui possède la vision du jeu et travaille sur le *game design*,
— Le CFO (directeur financier), qui s'occupe également de la stratégie marketing, du *community management* et des ressources humaines,
— Le CTO (directeur technique), qui a également, bien entendu, les mains dans le code et le dernier mot sur les choix techniques,
— Un *game designer*,
— Deux artistes 3D,
— Un *tech artist*, qui s'occupe des aspects techniques du monde en 3D et développe des outils pour les artistes,
— Une équipe de trois *game developers*, chacun ayant ses domaines de prédilection,
— Et... moi, qui suis responsable de tout le *backend* et de l'infrastructure.

C'est ce que l'on peut réellement appeller une équipe **à taille humaine**, pour lancer un projet dont l'envergure nous pousse à porter plusieurs casquettes à la fois. Moi qui rêvais d'être autonome dans mes prises de décision, le moins que l'on puisse dire est que je suis servi !

1.3. C'est quoi, "le backend et l'infra" d'un jeu ?

Eh bien pour résumer, c'est **tout ce dont on n'entend absolument pas parler dans les tests**, mais qui est absolument vital au bon fonctionnement du jeu. Si vous préférez, c'est tout ce qui va permettre au *monde persistant* d'un MMORPG de... persister, justement.

Prenons quelques exemples concrets.

Vous créez un compte sur un jeu massivement multijoueur. Pour vous *authentifier* auprès de ce jeu, il y a besoin d'un service qui aille vérifier votre login et votre mot de passe, ou encore qui fasse le lien entre votre compte Google/Facebook/AppleID et votre identité dans le jeu. Ce service d'authentification fait partie du **backend**.

Vous commencez à jouer et créez votre avatar dans le jeu, en choisissant son sexe, son apparence, ses vêtements... ces données doivent exister dans une base de données pour rester persistantes. Là encore c'est un service du backend qui s'occupe de les gérer.

Comme il s'agit d'un jeu *massivement* multijoueur, vous imaginez bien que le monde n'est pas simulé par un seul serveur de jeu, mais toute une flotte de serveurs qui se partagent la simulation : pour adapter en temps réel le nombre de serveurs et la façon dont ils se partagent le monde du jeu en fonction du nombre de joueurs qui sont connectés et de leur répartition dans le monde, il y a là aussi besoin d'un service qui va provisionner des machines, lancer des nouveaux serveurs ou bien en arrêter, et plus généralement jouer les chefs d'orchestre pour maintenir la cohérence de la simulation : ici, il s'agit d'un service d'**infrastructure**.

Allez, un dernier exemple : lorsque vous vous connectez au jeu, il faut que "quelqu'un quelque part" dise à votre client sur quel serveur de jeu se trouve le point du monde où vous allez *spawner* après vous être authentifié, et ce "quelqu'un" est encore un service du backend, qui est perpétuellement au courant de l'état actuel de l'infrastructure (combien de serveurs, quel serveur simule quoi, quelle est son adresse…).

Cela commence à vous sembler démesurément complexe ? Je vous rassure, à moi aussi, et j'ai mis un certain temps avant de ne plus avoir le vertige. 🍊

Notez que le point commun entre tous ces exemples est que *si le backend et l'infra font correctement leur boulot*, les joueurs n'auront même pas conscience de leur existence, et ils seront happés, sans accroc, par l'univers du jeu qui s'anime comme par magie sous leurs yeux et leurs mains. Cette remarque est particulièrement importante, car il s'agit de savoir **trouver sa satisfaction** dans le fait de rendre possible l'impossible et de rendre des services vitaux aux développeurs du jeu tout en restant **invisible**.

2. Commencer petit

Comme je viens de vous le dire, le backend d'un MMORPG est un projet **gargantuesque**, qui rend des services extrêmement divers aux *game devs*. En fait, vous pouvez me croire sur parole ou bien tenter l'exercice par vous-même : plus on réfléchit à ce qui se passe dans un jeu en ligne, plus on trouve de choses totalement nouvelles à rajouter dans son backend.

Dans ce contexte, la question que l'on se pose en buvant notre premier café est évidemment la suivante : **par où commencer ?**

2.1. Les choix technologiques du départ

Comme je l'ai laissé entendre dans l'intro de ce billet, j'y ai répondu en commençant à réfléchir à la *stack* technologique que j'allais utiliser pour construire cet édifice, et deux de ces choix étaient pratiquement imposés :

— Tout comme le backend du précédent jeu, celui-ci serait développé en Go ;
— L'infra du précédent jeu était en cours de portage vers un cluster Kubernetes dans le cloud, et vu que le marché semble massivement se diriger vers ce genre de choses, on va garder cette direction.

Cela semble maigre, mais c'est déjà ça de posé, alors j'ai commencé par ouvrir un livre sur Kubernetes[1] pour me mettre dans le bain.

> On pourrait croire qu'engloutir immédiatement un livre revient à « se concentrer trop tôt et trop vite sur un détail technique ».
> Détrompez-vous.

Tout cet univers *Cloud Native* étant nouveau pour moi, cette lecture m'a permis de **dissiper le brouillard** et de me faire une idée de ce qui allait bientôt devenir mes préoccupations quotidiennes, tout en sachant que cela ne me renseignerait que sur la composition du mortier qui serait utilisé dans la construction de ma cathédrale. Sans pour autant répondre aux questions les plus urgentes pour le projet, cela m'a déjà permis de me rassurer en me donnant une première image mentale de mon travail : « je vais réaliser un système distribué et à haute disponibilité, à base de *conteneurs* et de *pods* et de *services* dans le cloud grâce à Kubernetes ».

Une fois mon livre reposé et mon rythme cardiaque revenu à la normale après la crise de panique initiale, j'ai pu commencer à réfléchir de façon plus *rationnelle* à mon problème.

2.2. Dépiler un "petit truc" et écrire un prototype jetable

Je ne vous étonnerai certainement pas en vous disant que je n'étais *pas le seul* à me poser des questions difficiles pour attaquer ce projet. Pour les *game devs* le début d'un nouveau projet consiste généralement en deux choses :

— S'assurer que les bases du *gameplay* sont là (la caméra, le personnage, les contrôles) et bien confortables,
— Avoir en tête un *Minimum Viable Product*, c'est-à-dire une version très minimaliste et dénudée du jeu qui en pose les bases, et qui réponde à la question *est-ce que ce jeu est amusant ?*

Pour autant, en ce qui concernait le backend et l'infrastructure, le CTO avait principalement deux questions qui lui trottaient dans la tête au tout début :

— Avoir au moins une vague une idée de la façon dont le jeu allait devenir "*MMO*", c'est-à-dire *scalé* sur plusieurs serveurs de jeu,

1. *Cloud Native DevOps with Kubernetes : Building, Deploying, and Scaling Modern Applications in the Cloud* , par John Arundel et Justin Domingus, chez O'Reilly.

— Savoir si le reste des choix techniques du backend allaient suivre le même modèle que le précédent jeu (une API REST/JSON/HTTP).

Dans ces conditions, le tout premier problème technique dont nous ayons discuté était de savoir *comment différents serveurs de jeu allaient pouvoir communiquer entre eux pour que la simulation reste cohérente*, et nous sommes vite arrivés à la conclusion qu'il devrait exister, à un moment donné, un service qui se chargerait de router des messages (des événements) entre les serveurs concernés.

On pourrait poser le problème de cette façon : si nous tenons pour acquis que les serveurs vont simuler chacun une *zone* du monde du jeu, *en partant de l'hypothèse que nous avons déjà résolu tous les problèmes techniques que cela implique* (car cette problématique est remise à plus tard), comment faire pour qu'une explosion ou un incendie de forêt, qui sont censés être visibles à des kilomètres à la ronde, puissent être vus par les joueurs du serveur B chargé de simuler une zone se situant à 500 mètres de celle du serveur A où l'événement se produit ?

Pour répondre à cette question, j'ai entrepris de coder un service de messagerie où l'on "s'abonne" à une zone rectangulaire du monde (celle que l'on simule), et où l'on envoie des événements ayant chacun une position et un rayon d'effet : chaque événement sera routé vers les serveurs qui simulent une zone d'où l'événement doit être visible.

?

Est-ce que ce service sera immédiatement utile aux *game devs* ? Pas du tout.
Est-ce que son code sera le même lorsqu'il sera vraiment question de distribuer la simulation ? Probablement pas.
Alors à quoi ça sert de commencer par ça ?

Eh bien déjà, **c'est une problématique simple ("au scope très réduit") et le cahier des charges est trivial à formuler**. À ce moment du projet, je commençais à peine à travailler avec cette équipe, et il est primordial de commencer par quelque chose qui s'énonce clairement, pour **s'assurer que l'on est capable d'en parler et de l'expliquer à n'importe qui dans la boîte**, y compris le CEO et les artistes qui n'ont pas du tout un profil technique. Il ne faut surtout pas négliger l'importance d'installer dès le départ une communication fluide avec tous les membres de l'équipe, et pour cela, la première chose à faire est de *s'assurer que vous êtes capable de faire comprendre à tout le monde ce sur quoi vous travaillez*, car c'est ainsi que les gens sauront venir vous trouver pour vous demander de l'aide quand ils en auront besoin, et qu'ils n'auront aucune réticence à le faire.

Ensuite, cela a été l'occasion de **faire des choix techniques** et de **me familiariser** avec ma stack, car la vraie question à laquelle je désirais répondre était celle de la techno à utiliser pour échanger des messages. J'avais mis en lice trois concurrents :

— Des *sockets* TCP tous simples, ce qui m'aurait ensuite probablement poussé vers du classique TCP pour les connexions persistantes, et du REST/JSON/HTTP pour mon API,
— ZerØMQ , qui m'avait toujours séduit pour son aspect "Lego", à savoir qu'elle permet de créer des architectures arbitrairement complexes en composant entre eux des *patterns* atomiques de communication,

— gRPC , un *outsider* dont je venais d'entendre parler dans mon livre sur Kubernetes (car il l'utilise en interne), qui semblait faire de plus en plus de bruit, et qui impliquait, par simple soucis de cohérence, de l'utiliser à la fois pour la messagerie et pour l'API.

En une semaine, j'ai essayé d'implémenter les trois versions de ce service, pour comparer les approches, la facilité avec laquelle j'ai fait le boulot, et les performances obtenues. Si je dis « *essayé* », c'est parce que j'ai renoncé à ZerØMQ en cours de route : je n'ai jamais réussi à faire fonctionner le *binding* en Go, et quand bien même, mon cas d'utilisation demandait déjà de taper dans les fonctionnalités exotiques de cette techno, qui ne prévoyait évidemment pas de *pattern* pour réaliser un *Publish/Subscribe* basé sur des collisions entre des cercles et des rectangles.

Une semaine, c'est le temps qu'il m'a fallu pour tomber amoureux de gRPC. Il s'agit bien sûr d'un amour rationnel et 100% justifiable d'ingénieur, pas d'un simple coup de foudre devant la première techno sexy que je vois passer, mais je vous expliquerai tout ça plus tard. À la fin de cette étape, j'avais fixé un nouveau choix technologique, et nous verrons que tout le reste de ma *stack* découle naturellement de celui-ci.

Pour l'heure, finissons-en avec la dernière étape du *bootstrap* de ce projet.

2.3. Réaliser un "tracer bullet"

Le terme *tracer bullet* a été rendu populaire par le célèbre livre *The Pragmatic Programmer* . Contrairement au *prototype* qui est un code jetable permettant de se familiariser avec un problème, le *tracer bullet* désigne une **micro-fonctionnalité** d'un système, que l'on implémente et intègre de bout en bout pour la faire fonctionner dans les mêmes conditions que la production. L'idée, ici, est réellement de poser la première pierre de l'édifice final, autour de laquelle on pourra construire tout le reste.

Concrètement, il ne s'agit pas de faire un *code qui marche "sur mon pc" dans des conditions de test* pour lever un verrou technique, mais plutôt **quelque chose qui ne fait trois fois rien, mais qui le fait rigoureusement bien, en production** :

— La fonctionnalité doit être versionnée, testée, configurée et déployée de la même manière que le système final,
— Elle expose des logs et des métriques compatibles avec le besoin d'*observabilité* d'un système en production,
— Elle tourne dans la même infrastructure que la prod,
— Etc.

Cette approche est connue pour résoudre le plus tôt possible les divers problèmes d'intégration auxquels on ne pense pas naturellement lorsqu'il s'agit de développer un nouveau système. Elle permet d'acquérir dès le départ une vision certes partielle, mais **profonde** et **réaliste**, du logiciel final, de son cycle de vie, de son processus de fabrication, et plus généralement de *toutes les couches* de son architecture.

Cette étape a duré pour moi plusieurs semaines, pendant lesquelles j'ai développé, testé, versionné, intégré, loggé, monitoré et déployé un service qui sert à exécuter des opérations CURLD sur une bête ressource `User`, composée au départ de quatre champs :

— Un identifiant (UUID) unique et immuable,
— Un pseudonyme unique,
— Une date de création,
— Une date de dernière modification.

Oui oui, juste ça. Ça suffisait *largement*, car cela m'a permis de fixer définitivement de nombreux choix techniques dont certains seront justifiés plus bas. Jugez par vous-même :

— Mon code serait versionné dans gitlab (la version *cloud* gratuite) et testé et publié via gitlab CI,
— Chaque service respecterait le 12-factor : la ligne de commande serait gérée par Cobra , et la configuration par Viper ,
— Ma base de données serait une base PostgreSQL *managée* par la plateforme Google Cloud,
— Les migrations du schéma de la BDD seraient exécutées via l'outil migrate et seraient embarquées statiquement dans le binaire de mon application,
— Je m'y connecterais grâce à l'ORM gorm , qui s'intègre à merveille avec gRPC,
— Mon service serait monitoré par Prometheus parce que c'est déjà pour ce système que Kubernetes expose ses propres métriques et qu'il existe un middleware tout beau que je peux intégrer en 3 lignes à mes serveurs gRPC,
— Mes logs seraient gérés avec la bibliothèque Zap développée par Uber, parce qu'il est trivial de les structurer pour les rendre lisibles dans la console de Google Cloud .
— Le tout serait testé à l'aide de testify , parce que c'est une surcouche bien pratique qui se greffe naturellement sur la toolchain standard de Go.

À la fin, ce tout petit service était déployé dans un cluster Kubernetes sur Google Cloud, distribué en plusieurs instances à travers plusieurs machines ("nœuds") et servi sur une IP publiquement accessible par un *load balancer*. Autrement dit, j'étais désormais *confiant* sur le projet, car le Cloud n'avait pour moi plus rien de nébuleux.

3. Sur les épaules de géants

Aujourd'hui, mon backend est en excellente forme, les services fleurissent à mesure que les *game devs* en ont besoin, et tout ce beau monde, y compris les serveurs de jeu, ronronnent tranquillement dans le Cloud.

Et surtout, je suis *serein*. Non pas que le système sur lequel je travaille ne soit finalement pas aussi complexe que prévu (bien au contraire !), mais je ne ressens *aucune force de frottement*, et je dois cette sérénité à ma *stack*.

Voici un aperçu de ce que je considère comme mes plus gros *wins*.

3.1. gRPC, c'est super-bon, mangez-en !

Comme à peu près tout le reste de ma stack, gRPC est une technologie Open Source créée initialement chez Google. Il s'agit d'une surcouche de Protobuf. Pour résumer son fonctionnement, regardez simplement ceci. Il s'agit de la *déclaration* du service que j'ai implémenté pour mon *tracer bullet* :

```proto
syntax = "proto3";
import "google/protobuf/empty.proto";
import "google/protobuf/timestamp.proto";

package user;

message User {
    string id = 1;
    string name = 2;
    google.protobuf.Timestamp created_at = 3;
    google.protobuf.Timestamp updated_at = 4;
}

message UserRequest {
    User payload = 1;
}

message UserIDRequest{
    string id = 1;
}

message UserResponse {
    User result = 1;
}

message ListUserResponse {
    repeated User results = 1;
}

service UserService {
    rpc Create(UserRequest) returns (google.protobuf.Empty) {}
    rpc Update(UserRequest) returns (UserResponse) {}
    rpc Read(UserIDRequest) returns (UserResponse) {}
    rpc List(google.protobuf.Empty) returns (ListUserResponse) {}
    rpc Delete(UserIDRequest) returns (google.protobuf.Empty) {}
}
```

Comme vous pouvez aisément le deviner, ce fichier résume *le format* des messages que l'on peut échanger avec ce service, ainsi que les *méthodes* du service lui-même. **Ce fichier définit un contrat** à partir duquel la magie peut commencer :

— Protobuf (et son extension gRPC) peuvent générer automatiquement le code du client et le *stub* du serveur pour ce service, *dans une palanquée de langages possibles* (Go, C++, C#, Python, Node, ...)
— Ce fichier définit un **contrat** que je passe avec les *game devs* : on se met d'accord dessus, et à partir de là, je n'ai plus qu'à implémenter les méthodes concrètes de mon service. Pour eux, il n'y a strictement *rien à faire*, car le client qu'ils peuvent utiliser depuis le code du jeu est automatiquement généré par la CI de mes services.
— Avec quelques annotations et une petite extension supplémentaire , sur un service

aussi banal que celui-ci et en considérant que `User` est une ressource dans une base de données gérée par gorm, **je n'ai même plus à écrire le code des méthodes** et peux me contenter de le générer automatiquement (avec bien sûr la possibilité de surcharger le comportement par défaut) et l'intégrer tranquilement.

Il faut savoir également que gRPC supporte des méthodes manipulant des *streams*, à savoir des connexions persistantes sur lesquelles on peut envoyer un nombre arbitraire de messages (comme des sockets, quoi). Dans ce cas d'utilisation, j'ai pu vérifier que le code auto-généré en Go *éclate littéralement en performances* tout ce que j'ai été capable de faire en manipulant moi-même mes connexions TCP de façon optimisée.

En résumé : c'est rapide, c'est propre, ça permet de communiquer sans ambiguïté et efficacement entre les équipes de développeurs, et surtout, *ça fait le plus gros du travail à notre place*. Utiliser gRPC constitue pour moi un gain de temps **monumental**.

3.2. Kubernetes : le système d'exploitation du Cloud

Vous expliquer comment fonctionne Kubernetes prendrait beaucoup plus de place qu'il ne m'en reste dans ce billet. Aussi vais-je me contenter de rester synthétique en vous encourageant à vous renseigner sur le sujet si cela vous intéresse. Pour comprendre Kubernetes, il importe de comprendre les enjeux techniques *réels* du Cloud. Une application, de nos jours :

— N'a pas le droit de s'arrêter de fonctionner (on veut qu'elles soient disponibles à 99.999%),
— Doit pouvoir passer à l'échelle instantanément : d'un seul serveur en heure creuse, nous voulons pouvoir passer, en quelques secondes, à une centaine pour absorber un pic massif de charge.

C'est exactement ça que Kubernetes fait pour nous. On lui décrit un service et le conteneur qui tourne derrière dans un fichier de configuration, on l'envoie en une simple ligne de commande, et *pouf* ! Il se débrouille pour créer en une poignée de secondes un déploiement qui colle au maximum à ce que l'on veut. Il surveille les programmes qui tournent, redémarrent ceux qui ne donnent pas de signe de bonne santé, augmente automatiquement le nombre de répliques s'ils commencent à devenir trop chargés, ou au contraire en élimine lorsque l'utilisation est trop basse... Ce n'est vraiment pas un hasard si l'industrie est en train de l'adopter en masse.

Les plus réfractaires d'entre nous pourraient croire que cela nous retire le contrôle sur ce qui se passe *sur les machines elles-mêmes*. Rien n'est moins vrai. Le fait de *monitorer* ce qui se produit sur les machines est un pré-requis indispensable à toute application qui tourne dans le Cloud, puisque c'est en partant de ces données que l'on peut prendre la décision de changer dynamiquement l'échelle de notre application. De toute ma carrière, jamais je n'ai contrôlé mon code *de si près*, *si tôt* et *si facilement* : il m'a suffi de déployer un serveur Prometheus dans mon cluster, et de l'utiliser comme source d'un grafana pour disposer de l'orgie de métriques produites par Kubernetes, allant de la simple consommation de CPU et de RAM, jusqu'aux métriques plus poussées (temps de réponse au 99e pourcentile, nombre de répliques actives en temps réel...).

3.3. Agones

Là, nous entrons plus dans la technologie *de niche*. Agones, c'est une bibliothèque née de la collaboration entre Google et Ubisoft, pour gérer dynamiquement des *flottes de serveurs de jeu* dans Kubernetes. En effet, un serveur de jeu ne se comporte pas tout à fait comme un serveur de *backend* :

— Les serveurs de backend sont généralement *stateless* et servent des requêtes unitaires, ce qui fait que l'on peut les coller derrière un *load balancer*, et si jamais un serveur tombe ou est arrêté, le service n'est pas interrompu pour autant : on le remplace et basta.
— Les serveurs de jeu, c'est autre chose : chaque serveur a son adresse à lui, propre. Il est hors de question de les placer derrière un équilibreur de charge, et du moment que des joueurs sont connectés dessus, il n'a *plus le droit* de s'arrêter, sans quoi cela ruinerait complètement l'expérience de jeu.

Pour faire tourner des serveurs de jeu dans Kubernetes, il y a donc besoin que celui-ci comprenne comment se comporte une flotte, comment la mettre à jour proprement, comment la faire passer à l'échelle... et c'est exactement ce qu'apporte Agones : une abstraction *custom* pour Kubernetes et une API pour la manipuler.

Autant vous dire que pour mon infrastructure, c'est une véritable aubaine. Même si une flotte de serveurs faisant tourner un MMORPG est généralement plus complexe à gérer que celle d'un jeu *eSport*, Agones a le mérite de me fournir tous les outils nécessaires pour que je puisse me concentrer sur les spécificités de mon projet, me déchargeant ainsi d'une quantité ahurissante de questions que j'aurais mis des mois à étudier sans cela.

En conclusion... j' ma stack, parce que grâce à elle, j'accomplis chaque jour depuis 6 mois des choses dont je ne me serais jamais senti capable avant de l'adopter.

Liste des abréviations

CURLD Create, Update, Read, List, Delete. 6