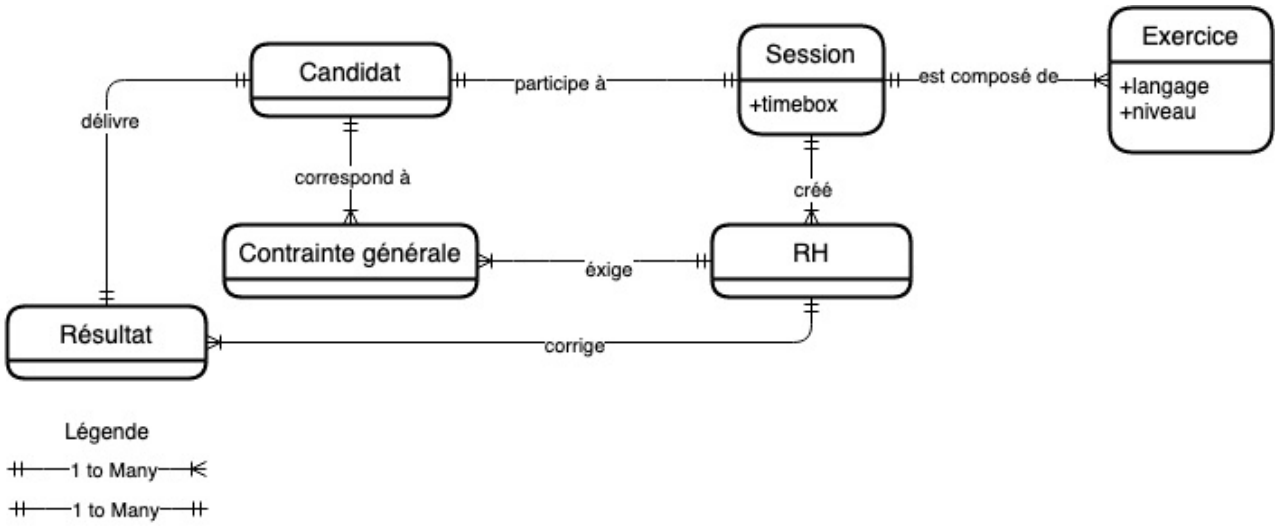
# Exercice 1 - Architecture

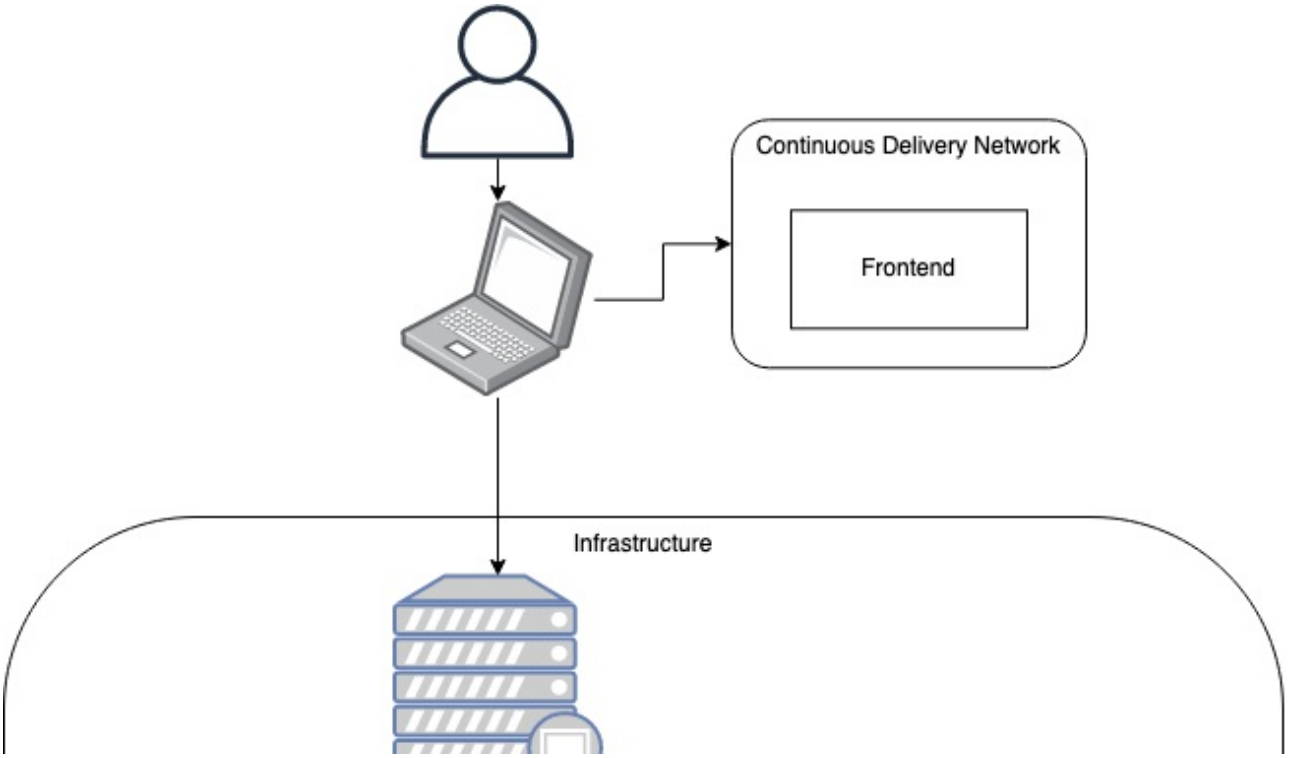
## Diagramme Entité-Relation

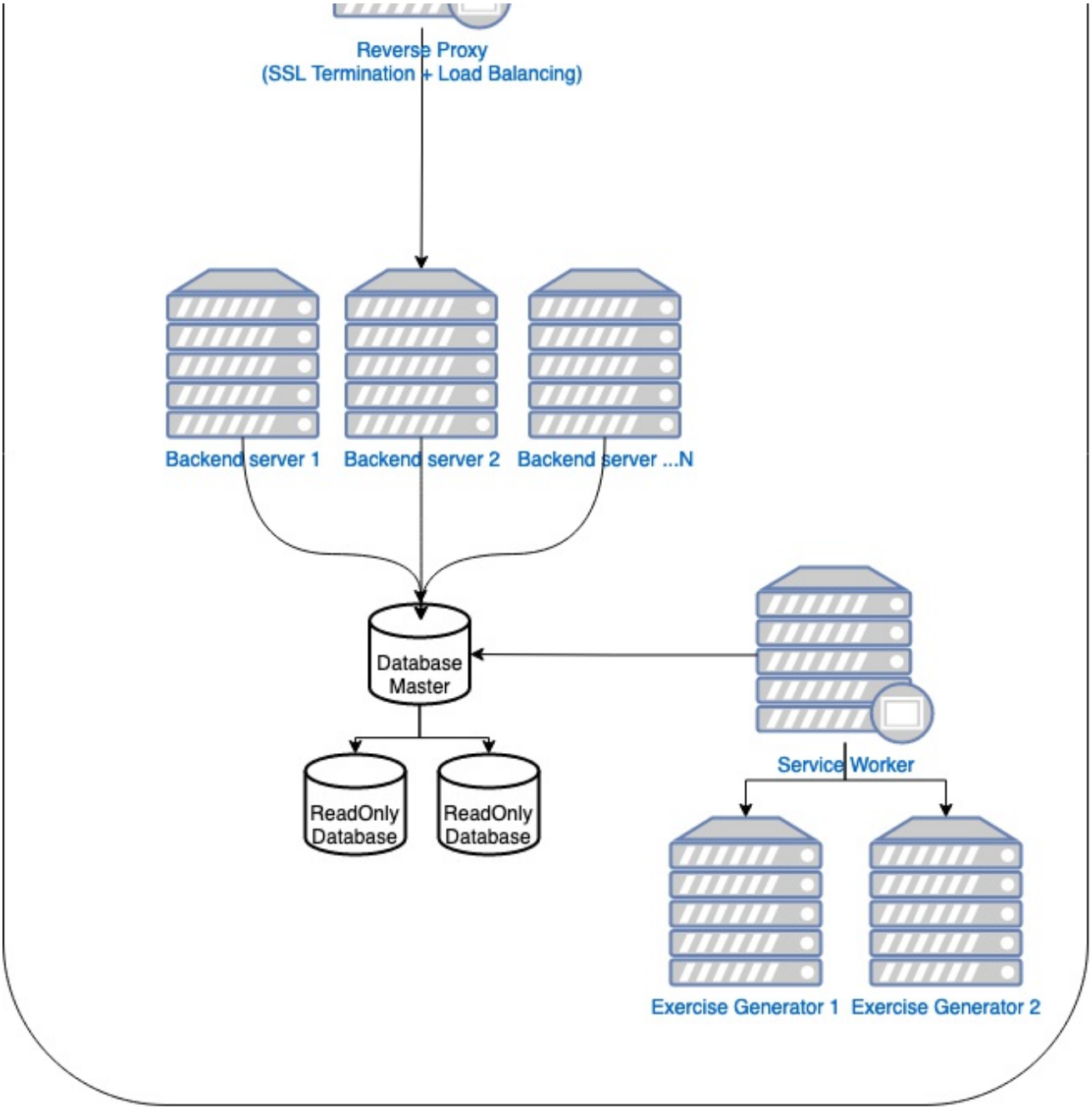



On a bien, ici, un schéma d'entité permettant les relations suivantes :

- un RH créé une Session
- un Candidat participe à une Session et délivre un Résultat
- un RH corrige les Résultats
- une Session est composée de plusieurs Exercices
- un RH exige différentes Contraintes Générales
- un Candidat peut correspondre à plusieurs Contraintes générales

## Diagramme d'architecture






## CDN

- On utilise un CDN pour héberger le front, car il s'agit d'un bundle statique. Il n'y a pas de besoin d'effectuer du server-side rendering.
- Comme le site peut être sujet à une forte charge en pic (sur 1/2 heure), on peut donc déléguer cette tâche à un hébergeur dédié comme OVH (qui est français) ou CloudFlare qui est le plus réputé en 2020.

## Frontend

- Angular est un framework complet et puissant qui permet de réaliser une application complexe et de générer un bundle sans se soucier des aspects techniques.
- React est un framework plus simple, qu'on peut considérer comme une librairie. Il constitue une très intéressante alternative car il est plus en vogue qu'Angular et permet donc de recruter plus facilement.

## Widget CLI

- TODO: faire faire ça côté frontend et pas côté backend. Trouver une techno qui permet ça.
- Voir https://pusher.com/tutorials/code-playground-react

## Reverse proxy

- Nginx est un très bon reverse proxy qui permet d'assurer la terminaison SSL et aussi de load-balancer les requêtes sur différents serveur.
- Une autre solution de reverse proxy serait HAProxy, ou même Apache, mais il est plus difficile de recruter des personnes compétentes sur ces technologies.

## Session

- Il y a un réel besoin identifié de gérer la montée en charge dû au type de business adressé. On peut donc écarter le cas du front qui peut être sous-traité, mais on doit tout de même protéger le backend.
- On peut utiliser une authentification classique, mais il peut être intéressant de mettre en place le JWT.
- Les JsonWebTokens sont la méthode la plus appropriée pour pouvoir scaler sur plusieurs backends en n'utilisant qu'un simple RoundRobin. Les données de sessions sont portées par le client. Ensuite, la connection HTTPS ou la signature du token permet de s'assurer que les données de session ne sont pas modifiées par un attaquant.
- Cette techno est utilisable que le serveur soit écrit en NodeJS, en Java ou autre.

## Backend

- NodeJS + Express est un couple de technologies de backend qui est en train de s'inscrire durablement dans l'écosystème web. On peut sans problème miser sur un tel modèle car on trouvera toujours des compétences dans ces domaines.
- En tant qu'alternative, on peut envisager Java pour les mêmes raisons : le recrutement
- Il est important que le backend soit sur une architecture dite "reactive" pour adresser la montée en charge. C'est pourquoi il est préférable de miser sur le monde JS / Java plutôt que PHP ou le concept n'est que très peu répandu.

## Générateur d'exercices

- Les exercices peuvent être longs à générer et coûteux en CPU et il y a un risque de devoir supporter une forte charge en pics (sur 1/2 heure). On souhaite donc pouvoir générer les exercices en avance et ne pas attendre que les candidats ne débutent une session (qui constitue le pic) pour les générer. De plus, les sessions sont timeboxées et le ressenti utilisateur ne doit pas en pâtir car c'est l'image de l'entreprise qui en souffrirait.
- On souhaite donc générer les exercices à la création de la session. C'est à dire que la charge sur le générateur interviendra lorsque les RH créent leur session, ce qui est moins risqué car (à priori) plus diffus dans le temps que lorsqu'un candidat débute une session (en plein pic de charge).
- Ce choix impose de stocker les exercices générés au lieu de les générer à la volée, mais puisqu'ils constituent le coeur de métier, il peut être intéressant de sécuriser ce point.
- Si la charge devenait trop lourde, alors on devrait pouvoir rajouter un serveur rapidement. C'est pourquoi je préconise de choisir tout de même une archi à N serveurs et de n'en prévoir que 2 dans un premier temps. L'important est de pouvoir rapidement monter en charge plutôt que de surdimensionner notre infrastructure.
- Chaque serveur est indépendant et créera un exercice à la demande du ServiceWorker.

## ServiceWorker

- Ce service scrute en permanence la BDD et lorsqu'il détecte qu'une session est créée et que les exercices sont à générer, alors il demande à un générateur d'exercice service de s'en occuper.
- On peut balancer la charge en utilisant la politique du round-robin qui est suffisante pour l'instant.
- Ce service est écrit en NodeJS car il utilisera les mêmes compétences et les mêmes process pour le développement que le backend (tooling, CI/CD, ...).

## Database

- Sur le principe du pattern CQRS (Command Query Responsibility Segregation), on va considérer qu'il y a plus fréquemment des requêtes de lecture que d'écritures. Rien que la scrutation permanente du ServiceWorker en est la preuve.
- On souhaite donc utiliser une technologie de BDD qui permettent d'utiliser plusieurs serveurs donc N pour l'écriture / réplication et au moins N+1 pour les opérations de lecture.
- On partira dans un premier temps 2 serveurs en lecture et 1 serveur en écriture.
- En choisissant MySQL, on obtiendrait 2 Slaves et 1 Master.
- En choisissant MongoDB, on obtiendrait un Replica Set avec un Primary Server et un Secondary Server.
- L'idéal est de se doter d'une personne compétente dans le métier de BDD qui a fortement évolué ces dernières années. D'expérience, j'ai pu constater qu'une équipe de dev sait généralement maîtriser son choix de techno BDD et sa structure de données, mais ne sait généralement pas comment optimiser les performances et la tolérance aux pannes. Comme il est crucial pour le Candidat de ne pas se retrouver frustré lors de son test timeboxé et qu'il peut y avoir des montées en charge en pic, il vaut mieux protéger ce point qui pourrait nuire à l'image de l'entreprise. Il y a donc 2 solutions : prendre un expert en prestation pour accompagner solidement le design de la solution BDD ou engager un Devops qui dispose déjà d'une expérience significative sur ce point (BDD à forte charge).
- Le format du résultat des sessions est à inspecter de très près pour définir s'il y a absolument besoin d'une BDD de type NoSQL. Le diagramme Entité-Relation a identifié qu'une base de données structurée pouvait suffire (à ce stade). On peut donc choisir MariaDB pour sa popularité, ou MongoDB si l'équipe est plus à l'aise.
- Le niveau de connaissances de l'équipe sur l'utilisation des librairies ORM/ODM peut influer le choix de l'équipe. Bien souvent, les équipent ont tendance à choisir MongoDB/Mongoose par effet de mode. Attention à ne pas se retrouver coincé par la suite si on souhaite versionner la structure des données, par exemple.
- PostgreSQL pourrait même être envisagé si l'équipe dispose des compétences car cette techno est plus proche du véritable métier de database que les 2 précédentes.

## Infrastructure

- On peut choisir d'héberger l'infrastructure en interne si on considère qu'il est crucial d'avoir la main dessus et qu'on dispose des compétences et des budgets pour assurer une tâche aussi lourde.
- Avec ce type de business, on peut considérer que la donnée n'est pas cruciale car il ne s'agit que d'exercices générés pour des candidats et les résultats leur appartiennent. On peut donc tout-à-fait externaliser les données car elles ne constitue pas une "proie" pour des attaquants ou des concurrents.
- Les sessions pouvant intervenir en dehors des heures de bureau, il peut être intéressant de s'appuyer sur un hébergeur comme OVH ou Online / Scaleway qui disposeront des moyens pour intervenir de nuit ou en week-end alors qu'en interne, il faudrait prévoir un programme d'astreinte pour protéger le business.
- L'OS a privilégier est Debian car il n'y a pas de besoin réel d'utiliser quelque chose de plus

moderne (et donc potentiellement moins stable). On peut toutefois s'orienter vers RedHat si on souhaite disposer d'un support très avancé lorsque l'équipe n'est pas suffisamment à l'aise en administration système ou si on risque d'avoir besoin d'expertises ponctuelles.

# Développement

## CI/CD

Si on considère que notre code est externalisable (la concurrence fait rage), alors on peut externaliser la CI/CD et le serveur Git chez Github. Sinon, alors on devra monter un Gitlab ou un Jenkins en interne et allouer une personne compétente au maintien / supervision de ce service.

## Tooling

L'équipe doit négocier quel niveau d'exigence elle souhaite en qualité de code (linter / build par branche, ...) et s'adapter donc d'elle-même sur ce point qui soulève généralement des débats houleux alors qu'il n'y a pas forcément lieu d'être.

## Testing

Il y a généralement 2 stratégies en testing automatique

1. Tester absolument tout pour sécuriser les développements à l'avenir (e2e/unitaires/intégration/automation). Effectivement, si le business devait croître exponentiellement ce serait plus rassurant d'être dans une situation comme celle-ci dès le départ mais c'est extrêmement coûteux et cela ajoute de l'inertie au projet.
2. Tester d'abord l'essentiel pour atteindre rapidement un niveau idéal de qualité. Dans cette démarche, on doit identifier le minimum de scénarios utilisateurs qui permettent de couvrir le MVP (Minimum Viable Product). Ensuite, on doit identifier ce qui est automatisable sur ces scénarios. Par exemple, si on est confronté à une grande combinatoire sur l'un des scénarios, alors il peut être pertinent d'automatiser ce point en premier (ex: Browserstack peut aider à tester sur une appli web sur plusieurs plateformes). C'est la stratégie que je préconise car elle protège le business et le coeur de métier d'abord, et elle n'empêche pas de consolider la qualité par la suite.

## Remarques générales

- Si on couple NodeJS/Express pour le backend et React pour le frontend, alors on peut employer NextJS pour se faciliter le travail. NextJS est un framework permettant de structurer facilement une application NodeJS/Express si on souhaite choisir React comme technologie de Frontend. Cela pourrait constituer un bon guide. Attention toutefois à la jeunesse du projet. Il faut pouvoir changer de technologie si une nouvelle "mode" survient à l'avenir et bloque les recrutements.
- Si on souhaite une alternative à NodeJS pour le backend et le Service Worker, on peut tout à fais choisir Java qui est plus lourd à héberger, mais qui permettra de toujours pouvoir trouver des personnes compétentes pour le faire. Python peut aussi être une solution, mais la laborieuse migration de Python2 vers Python3 a provoqué un précédent historique qui mitige le choix de Python pour l'avenir.

# Conception du générateur d'exercice

Puisqu'un ServiceWorker est chargé d'aller scruter la BDD, le générateur d'exercices en C n'aura pas besoin de connexion à la BDD. Le ServiceWorker s'en chargera et appellera directement notre générateur d'exercices les données d'entrée suivantes :

- langage
- niveau

Le Générateur d'exercices aura probablement besoin d'un des trois générateurs de nombres aléatoires du noyau Linux. Des 3 API noyau, on préférera celle de /dev/urandom car il n'est jamais bloquant et que la génération d'exercices n'est pas un sujet crucial en sécurité. Ainsi, on pourra garantir un temps d'exécution pour chaque génération d'exercices et s'engager sur une structure simple :

- pas de multithreading
- chaque générateur constitue un processus dédié
- on peut dédier un processeur par générateur d'exercice (voir une machine virtuelle par générateur)

Ainsi, la conception du générateur d'exercices est assez simple et maintenable. On peut tout simplement prévoir un main qui récupère les 2 paramètres d'entrée : langage et niveau et ensuite, appeler le générateur pour dédié au langage voulu, avec un fichier par langage. La structure des fichiers et fonctions peut ressembler à quelque chose comme ceci :

```
// main.c
int main(int argc, char *argv[]) {
 // Simple options parsing with getopt from POSIX (#include <unistd.h>)
}

// utils/utils.c
int some_utility_function(...) { ... }

// generators/python.c
char* generate_exercice(int level) { ... }
// generators/java.c
char* generate_exercice(int level) { ... }
// generators/bash.c
char* generate_exercice(int level) { ... }
// generators/ada95.c
char* generate_exercice(int level) { ... }
```

On évitera de s'engager dans une architecture objet en C++ car l'expérience montre qu'il est difficile de maintenir des applications lorsque l'archi se complique alors que la problématique est simple. L'architecture en modules (dossiers / fichiers / fonctions) semble suffisante à ce niveau.

# Références

- Diagrammes générés avec https://app.diagrams.net/
- Architectures reactives : https://www.reactivemanifesto.org/

Conversion Markdown vers PDF :

```
npx markdown-pdf README.md
```