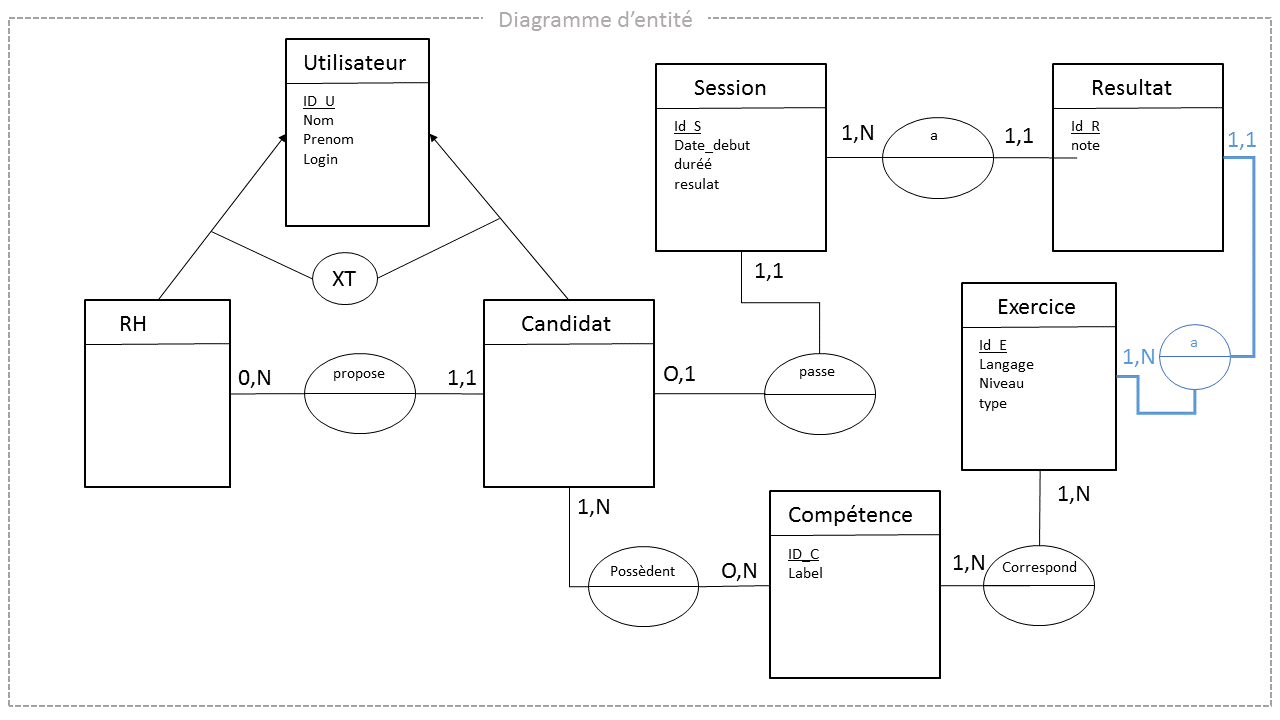
Réponse exo AbBeal

[disclaimer] Pour moi la partie générateur de test en C était un peu flou, je suis donc partie un fonctionnement proche de www.codingame.com.

Coding game est une application SaaS qui permet à des développeurs de coder l’IA d’un jeu, l’interface sous forme d’IDE permet de coder et de lancer les tests unitaires. Les langages de programmation sont variés (python, c, c++, java, JavaScript, etc) et le code est compilé côté serveur. De plus leur plateforme permet à des recruteurs de proposer un challenge et de tester les développeurs afin de de les recruter par la suite, ce qui ressemble un peu à l’exo.

# Diagram d’Entité



1. RH et Candidat héritent d’Utilisateur
2. Une RH a zéro ou plusieurs candidats
3. Un candidat peut avoir qu’une seul RH
4. Un candidat peut passer une session
5. Un candidat peut avoir une ou plusieurs compétences (ceux sont les contraintes dans les énoncés :« peut intégrer du web », « peut développer une app mobile »)
6. Chaque compétence correspond à une série d’exercices
7. Un exercice peut correspondre à une ou plusieurs compétence
8. Une session a plusieurs résultats
9. Un résultat appartient à une seule session
10. Un résultat est relié à un exercice (en bleu ici car le lien est une règle de gestion et il n’est pas modélisé en BDD pour éviter un cycle)

# Descriptifs des technologies utilisées

Pour créer un application SaaS, même s’il est possible de le faire en mode « bar métal », il est plus intéressant de le faire sur une plateforme cloud de type PaaS Ou IaaS.

## 1. Plateforme



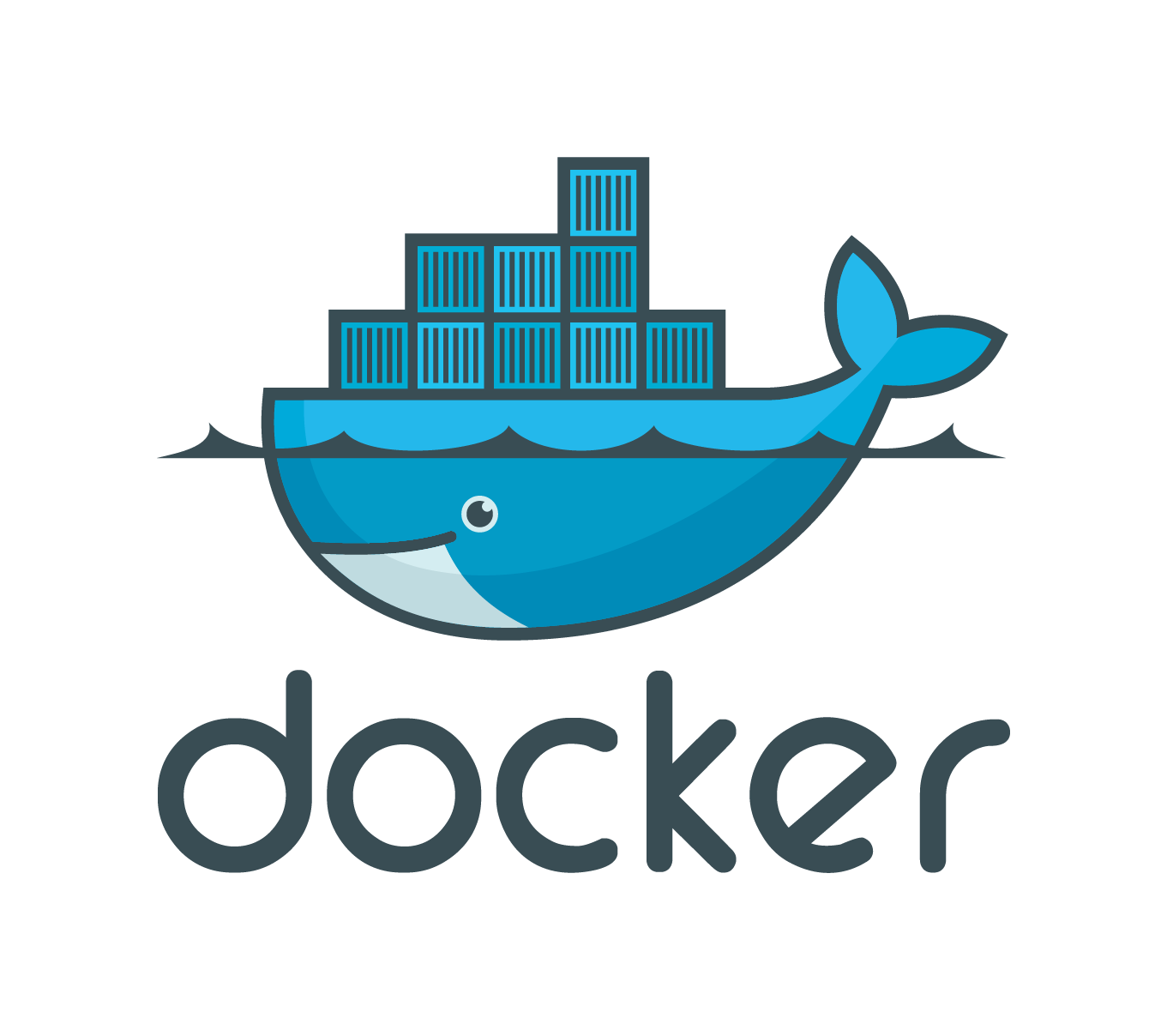
L’utilisation de Kubernetes permet déployer et gérer les différents briques contenus dans des containers dockers.

Ce déploiement peut se faire aussi bien sur:

1. Du baremetal avec CoreOS
2. Un Openstack
3. Un Cloud public comme Google App Engine ou Amazon ECS

Une fois configuré on peut alors facilement changer de fournisseur de Cloud ou même faire un cloud Hybride, par exemple si le cloud interne est saturé on peut alors déborder sur un cloud public.

Kubernetes permet de scaler automatiquement, il réduit ou augmente le nombre de container en fonction de la demande. C’est peu contraignant à mettre en place mais une fois cette étape franchie, le déploiement de nouvelles versions de l’application est rapide.



L’ensemble des briques logiciel qui suivent seront encapsulées dans des containers dockers.

Docker permet dans cette architecture l’isolation des ressources et un déploiement faciliter des applications.

## 2.Front

### 

### Serveur web statique

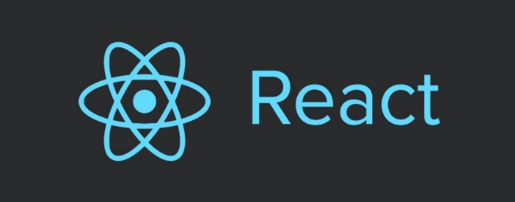


Nginx en frontal permet de servir les fichiers statiques, de faire le load balacing sur les Middles et d’avoir un premier niveau de cache.

Il est ultra performant, en effet il peut servir des dizaines de milliers de requêtes avec une empreinte ram/CPU minimal.

### Application Web front

Première solution : ReactJS + Redux + React routeur



Deuxième solution : AngularsJS + Twitter Bootstrap



ReactJS a très bonnes performances de rendu. ReactJS est une libraire, ce n’est pas une Framework comme AngularsJS, du coup il faut ajouter des briques supplémentaires et les intégrer.

D’un autre côté AngularsJS a beaucoup de fonctionnalité et évite de se poser trop de questions lors de la conception (choix de la lib de routage, etc.).

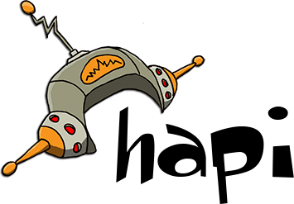
## 3.Middle

Première solution :



Spring boot est « cloudready » avec par exemple des facilités de gestion de session ou l’utilisation des variables d’environnement dans les properties. Il peut contenir un Tomcat embarqué pour faciliter son déploiement.

Deuxième solution :



Nodes js avec Hapi pour les API REST(solution simple et légère) , plus accessible pour des développeurs non expérimenté qu’avec l’univers Springs.

NodeJS , comme nginx, utilise le reactor pattern ce qui lui permet très performant.

Dans les deux solutions, on utilise les Websockets pour faire communiquer le front avec les résultats des tests qui sont calculés de manière asynchrone sur la plateforme. Ainsi dès qu’un résultat est calculé, on peut mettre à jour la session du côté du candidat.

Pour l’authentification, Spring boot ou Hapi ont des plugins simples d’authentification, et par la suite si on a besoin d’une authentification plus complexe, ils ont tous les deux un support d’Oauth2.

Ce qui est intéressant si l’on veut font switcher sur un serveur SSO de type CAS ou sur une authentification basée sur un tiers comme LinkedIn, Google, Facebook, etc.

## 4.Solution de stockage BDD

### Première solution :



MongoDB est une BDD NoSQL orienté document.

Elle se couple bien avec docker et Kubernetes. Elle est facilement scalable et performante.

### Deuxième solution :



PostgreSQL est une base de données sql robuste et performante, assez complète en termes de fonctionnalité.

Les avantages d’une basse de données NoSQL en général, c’est qu’elles sont mieux adaptées au scale, et donc au cloud computing.

Cependant de nos jours, des bases de données comme Postgress, ont des systèmes de synchronisation et réplication efficace qui permettent un scale acceptable, mais plus complexe à mettre en place.

Dans le cadre cette exercice, l’un comme l’autre donneront des performances assez satisfaisantes pour les besoins.

Dans d’autre cas, cela dépend vraiment du besoin, du type de données stockées et du besoin d’intégrité des données.

Dans le cas d’un site e-commerce, je recommanderai une bdd sql et dans le cas d’un réseau social plutôt une bdd no-sql.

Dans notre cas, les avantages ou inconvénients seront plus visibles au niveau du développement.

En méthode agile, utiliser une base de données SQL comme Postgress peut vite devenir contraignant.

En effet la mise à jour des schémas BDD peut devenir contraignant en cas d’itération courte avec de nombreuses modifications sur les données alors qu’avec une BDD NoSQSL comme MongoDB, on n’a pas besoin de s’en occuper.

Le fait que MongoDB soit une BDD orienté document qui stocke les données sous forme de JSON est un plus quand on a une Stack front et middle en JavaScript. Les données ont alors un format très proche de celui utilisé par le front et le middle.

Il faut aussi prendre en compte les compétences dans son équipe, en effet le gap d’apprentissage est plus élevé sur Postgress que sur MongoDB, notamment au niveau optimisation et clustering.

D’un autre côté, une base de données SQL oblige à modéliser correctement la structure de données et donc éviter certains problèmes de conception qui se verront en amont du projet.

## 5.Focus sur la gestion des sessions dans un environnement Cloud.

### Première solution

Pour gérer les sessions, on peut utiliser les sticky session de NGINX qui est capable de forwarder la requête vers le bon middle.

Nginx possèdent un avantage intéressant pour répondre à la problématique ; le rechargement à chaud, c’est-à-dire qu’il est capable de recharger ses configurations, tout en continuant de servir des requêtes et en n’arrêtant pas les requêtes en cours.

Cet avantage permet en cas d’ajout/suppression de serveurs middle de modifier ces configurations sans coupure ou dégradions de service.

Pour automatiser la modification des configurations, on peut utiliser NGINX Plus qui le fait nativement cependant cette solution est payante.

Sinon, on peut faire un peu de développement pour faire cette automatisation.

Cette solution est élégante, simple à mettre en place et permet d’être agnostique au niveau de la solution middle.

On répond bien à la problématique d’une infrastructure élastique mais si un middle s’arrête suite à une surcharge ou un problème, la session est perdue.

### Deuxième solution

On peut gérer les sessions au niveau du middle, on utilise alors un redis ou un Memcache distribué pour que n’importe quelles sessions soient accessibles aux différents Middles.

Spring boot a d’ailleurs un adaptateur Redis plutôt bien fait pour ce cas de figure.

Dans le cas de l’utilisation d’HAPI pour la couche Middle, la session est entièrement stockée dans le cookie, donc la session n’a pas besoin d’être gérer lors d’un ajout ou d’une suppression de serveur middle, en d’autres termes le scale horizontal est géré nativement, sans avoir besoin de se préoccuper de la gestion des sessions.

Dans le cas où le nombre d’informations en session dépasse la capacité du cookie alors on peut utiliser HAPI avec redis comme solution de stockage de session.

## 5.Focus sur l’intégration de la partie C

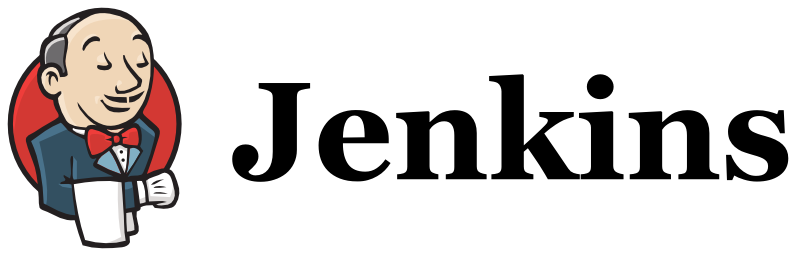
### Première solution

Soit le générateur de test en c est assez rapide (< 1s) dans ce cas on peut l’encapsuler dans une API faites avec NodeJS et un binding du code c.

### Deuxième solution

Si la génération des tests est assez longue, alors il est préférable de le mettre dans un orchestrateur comme Jenkins pour éviter une surcharge/surutilisation de la plateforme.

Pour se faire, on créer un build Jenkins, ce build Jenkins peut alors être lancé via les API Rest Jenkins.



Comme dans cette solution, les générations des tests des candidats sont faites de manière asynchrone avec Jenkins, il est intéressant d’ajouter un Bus de communication pour améliorer la résilience de la plateforme.

Cette brique sert à faire la communication entre le middle applicatif, Jenkins et les différents builds.

Pour le bus de communication, on peut utiliser en autre :

* RabbitMQ qui est facile à mettre en place.
* ZeroMQ plus compliqué à mettre en place mais avec de meilleures performances.





Je commencerai avec un RabbitMQ, puis suite à des benchmarks de la plateforme, si les besoins s’en font sentir, on peut switch sur ZeroMQ.

## 6.Monitoring de la plateforme

Kubernetes permet de monitorer les différents containers de la plateforme. Cependant pour améliorer la qualité de service, on peut utiliser Nagios qui permettra de faire de la surveillance applicative et faire efficacement l’exploitation.

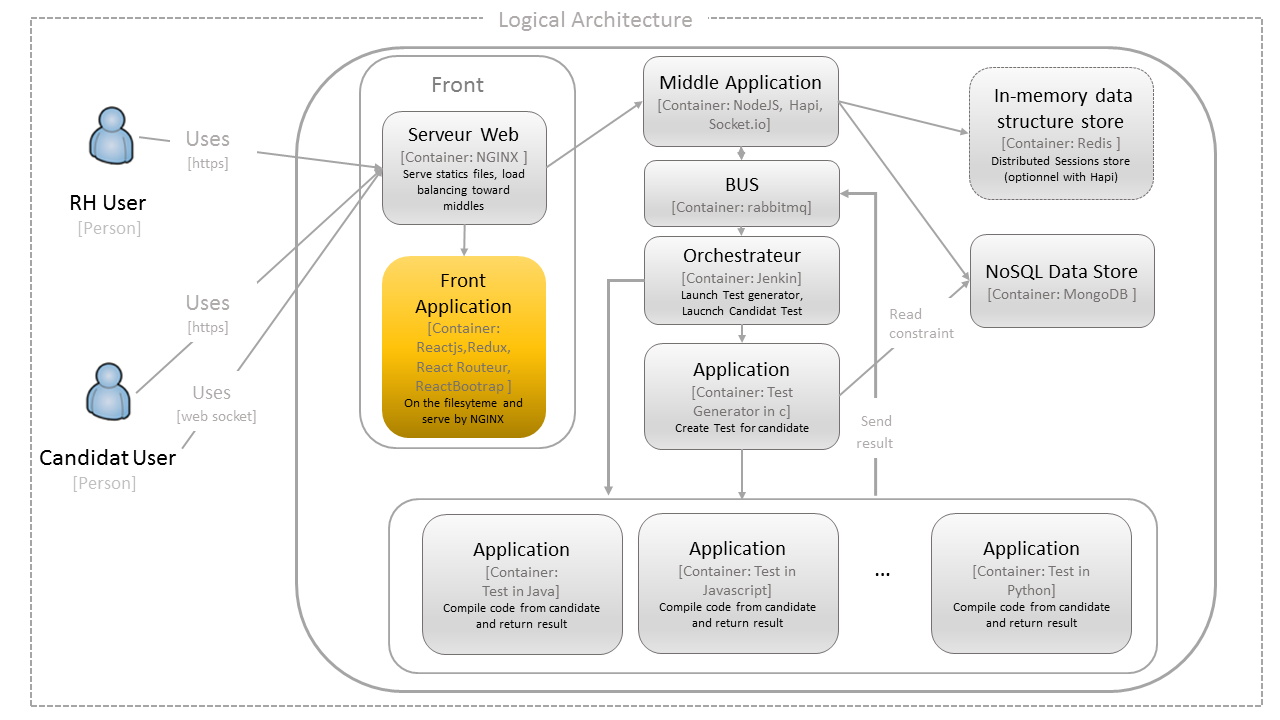
## 7.Gestion du pic de charge

Les benchmarks de la plateforme vont permettre par extrapolation de savoir le nombre d’instances dockers nécessaires lors du pic de charge. Pour assurer lors de ce pic, on peut via Kubernetes prédéfinir le nombre d’instances minimal à l’heure H.

Le fait que l’application soit scalable par les technologies utilisées et sa plateforme cloud vont permettent d’absorber une arrivée massive de requêtes.

Néanmoins pour avoir une solution de secours au cas où l’infrastructure interne soit saturé, on peut mettre en place un mécanisme de débordement, qui renverra la charge en trop sur un cloud public, grâce à Kubernetes.

# Diagramme d’architecture



Ce diagramme montre la solution que je retiendrai au niveau des stacks logiciel.

C’est un schéma logique qui montre l’interconnexion entre les diverses briques par contre il ne montre pas les redondances de container automatiquement faites par Kubernetes.

L’ensemble des solutions retenues permet d’avoir une architecture scalable, performante et résiliente.

On pourrait l’améliorer en proposant une architecture micro service en découpant le middle applicative en plusieurs briques, ce qui permettrait d’améliorer sa résilience mais complexifiera les interconnections.

# conception de l’algorithme C

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

Pour construire l’algorithme en c, je suivrai ces étapes :

1. Recueille du besoin
2. Description des fonctionnalités en utilisant la méthode BDD(Behavior Driven Development => given,when, then) pour éviter des incompréhensions avec le client
3. Diagramme de séquence
4. Identification d’algorithme existant
5. Création de pseudo code
6. Calcul de la complexité
7. Recherche de librairies existantes
8. Choisir la structure de données la plus appropriée
9. Recherche de design pattern approprié
10. Création des prototypes des fonctions dans le .h
11. Création des implémentation vide
12. Création des tests unitaires
13. Implémentation
14. Ajout de cas plus complexe à gérer

Suivant les cas, toutes les étapes ne sont pas nécessaires ou pas dans le même ordre.