



ÉCOLE
D'INGÉNIEURS
PARIS-LA DÉFENSE

Application pour le Cloud

Développement d'application Cloud

Nicolas PICARD
Henri DUHAMEL
Johan VAN DER SLOOTEN
Maxence CLUSAZ

Décembre 2019

Table des Matières

1	Architecture	2
2	API & Client	3
2.1	Utilitaires	3
2.2	Partie utilisateur	4
2.3	Partie analyste	5
2.4	Partie administrateur	6
3	Conclusion	7

1 Architecture

Notre application devant être hébergée sur le cloud, nous avons décidé de réaliser son architecture en séparant les données de l'UI. En effet, de nos jours, de nombreuses applications utilisent un client léger qui réalise des appels AJAX vers un API REST. Puisque nous utilisons MongoDB qui utilise le JavaScript, nous avons décidé de nous placer dans la continuité et d'utiliser Node.js (Express.js pour être exact) pour l'API ainsi que React.js pour le client. Le tout est containerisé à l'aide de Docker, afin d'avoir le même environnement pour chaque application. Nginx étant un serveur assez simple d'utilisation et efficace pour la mise en cache, nous avons décidé de placer le build (html statique) du client derrière ce serveur. Vous trouverez ci-dessous notre schéma d'architecture de l'application. Les Dockerfile ainsi que le fichier de configuration Docker Compose sont disponibles en annexe.

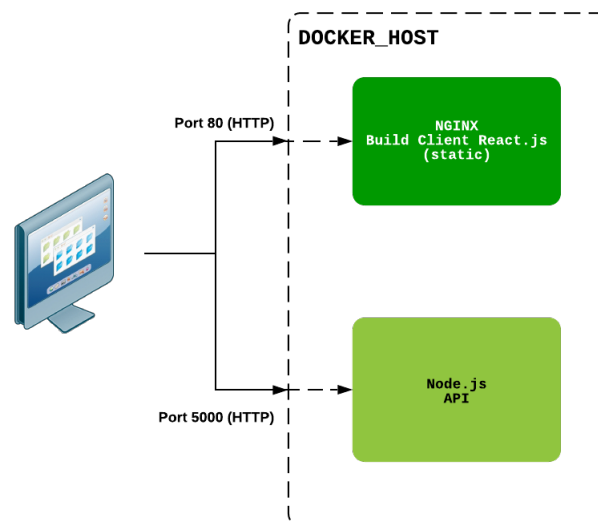


Figure 1: Architecture de l'application

2 API & Client

2.1 Utilitaires

Afin d'éviter la répétition de code au sein de l'API, nous avons créé des méthodes génériques permettant de réaliser les opérations les plus courantes sur la base de données, soit find, aggregate et mapReduce. Vous trouverez ci-dessous en exemple notre méthode permettant de réaliser un aggregate depuis une pipeline, puis de renvoyer les résultats dans une fonction de callback.

```
exports.aggregate = (pipeline, callback) => {
  this.connect((err, client) => {
    if (err) return callback(err, null);
    const db = client.db(dbName);
    const collection = db.collection(collName);
    collection.aggregate(pipeline, (err, results) => {
      if (err) return callback(err, null);
      results.toArray((err, docs) => {
        if (err) return callback(err, null);
        return callback(null, docs);
      });
    });
  });
};
```

Figure 2: Utilitaire pour agrégation

Après avoir défini chacune des requêtes sous forme de route, nous arrivons aux routes correspondantes:

n°	description	route
1	Obtenir sa fiche de paie de l'année actuelle	/employees/:emp_no/salaries/last
2	Obtenir les employés travaillant dans son département	/departments/:dept_no/employees
3	Obtenir le manager d'un département	/departments/:dept_no/managers
4	Obtenir le nom, prénom des derniers employés arrivés dans son département	/departments/:dept_no/employees/latest
5	Obtenir le salaire moyen par département	/departments/stats/salary
6	Obtenir tous les employés qui ont le même titre	/employees/titles
7	Obtenir l'age moyen des employés pour chaque département	/departments/stats/age
8	Obtenir la répartition du genre des employés/managers par département	/departments/stats/gender?type={manager}
9	Obtenir le nombre de documents par shard	/admin/stats
10	Obtenir le poids moyen d'un document par shard	/admin/stats
11	Obtenir le statut des shards	/admin/shards
12	Obtenir la répartition des chunks	/admin/chunks
13	Obtenir des détails sur les chunks	/admin/chunks/details
14	Obtenir des informations sur l'hôte (uptime, etc.)	/admin/status
15	Obtenir des informations sur les connexions ouvertes	/admin/connections
16	Obtenir des informations système sur l'hôte (distribution, kernel, memory)	/admin/host

Figure 3: Route et requête respective

2.2 Partie utilisateur

Avant toute chose, nous avons dû définir un thème pour notre application et un style. Après avoir vu la grande adoption du Material Design de Google dans le web, l'utilisation de celui-ci s'est parue évidente due à sa simplicité. L'existence de bibliothèques de composants React nous a simplifié le développement car nous avons utilisé Material-UI (<https://material-ui.com/>), un framework React permettant l'utilisation personnalisée de composants suivant ce design.

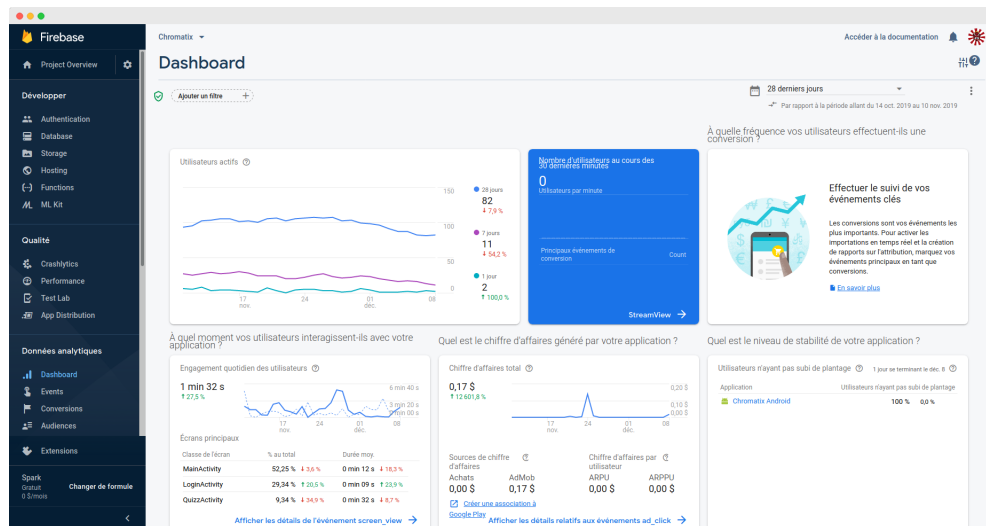


Figure 4: Dashboard Firebase utilisant le Material Design

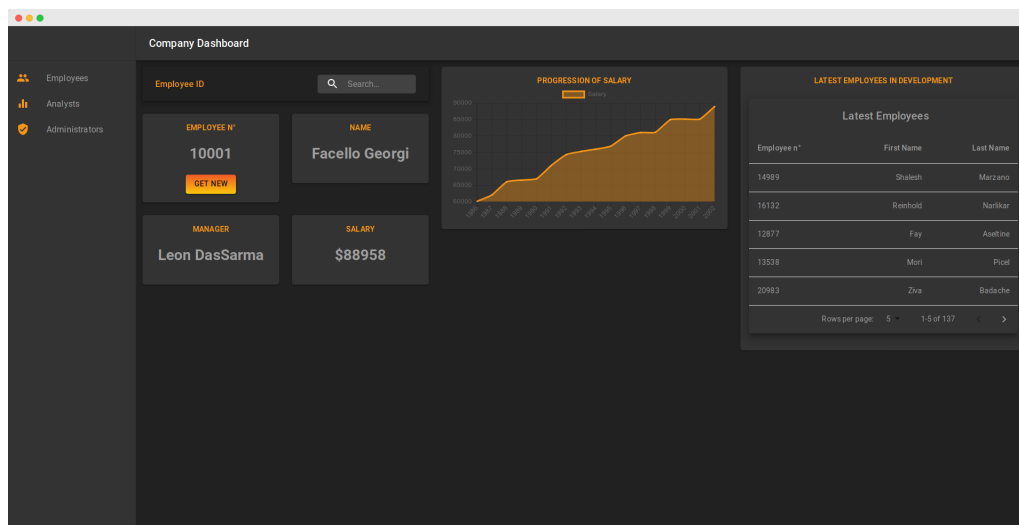


Figure 5: Dashboard utilisateur

Nous avons développé la partie utilisateur du dashboard de telle façon à ce qu'elle rassemble les informations les plus importantes pour un employé. L'employé en question pourra y trouver son numéro d'employé, ses informations personnelles telles que son salaire actuel, le manager de son département, les derniers employés arrivés dans son département, mais encore la progression de son salaire depuis son arrivée dans l'entreprise. Nous avons également ajouté un bouton permettant d'obtenir un nouvel employé aléatoire (de façon à pouvoir jouer un peu avec le dataset). On peut également chercher un employé par son numéro. Ces dernières options sont avant tout présentes pour des utilisateurs comme nous, afin de pouvoir observer les informations de différents employés, et non de donner la possibilité à un employé d'obtenir des informations confidentielles sur ses collègues.

2.3 Partie analyste

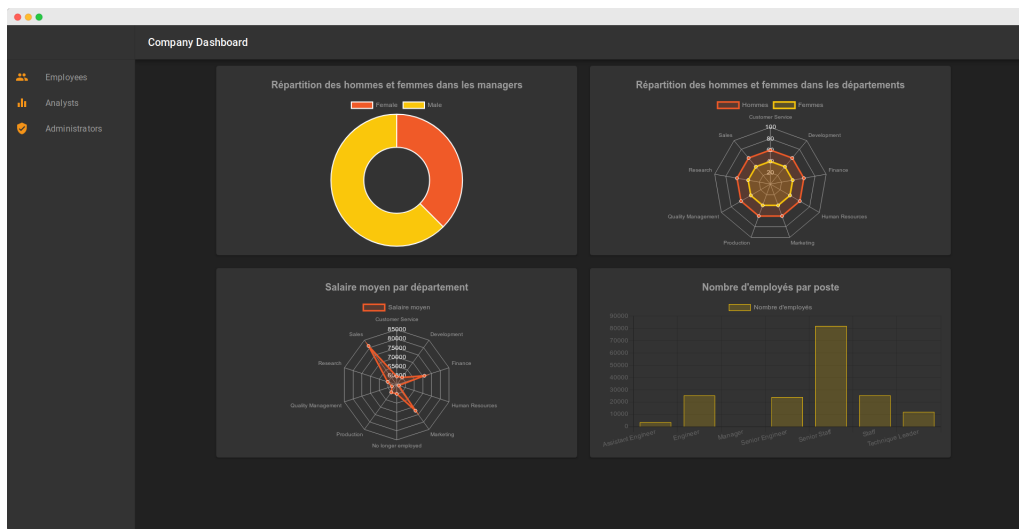


Figure 6: Dashboard analyste

La partie analyste du dashboard a elle été pensée pour rassembler les informations pertinentes aux départements ou aux employés en général, de façon à obtenir des statistiques sur la répartition du genre parmi les départements et les managers. Un analyste y retrouvera également le salaire moyen par département, ainsi que le nombre d'employés occupant le même poste. Malheureusement pour nous, notre jeu de données étant "faux", on obtient des résultats similaires pour le pourcentage d'hommes et de femmes à travers chaque département. Nous avons également développé un graphique représentant l'âge moyen par département, mais il s'est avéré qu'il n'était pas pertinent d'afficher ces informations. En effet, nous obtenions des valeurs très très proches (différence d'environ 0.02 ans par département), ce qui n'était pas utile selon nous.

JSON	Données brutes	En-têtes
Enregistrer	Copier	Tout réduire Tout développer
		Filtrer le JSON
▼ data:		
▼ 0:	department: "Customer Service" age: 60.91	
▼ 1:	department: "Development" age: 60.9	
▼ 2:	department: "Finance" age: 60.92	
▼ 3:	department: "Human Resources" age: 60.94	
▼ 4:	department: "Marketing" age: 60.9	
▼ 5:	department: "No longer employed" age: 60.92	
▼ 6:	department: "Production" age: 60.94	
▼ 7:	department: "Quality Management" age: 60.94	
▼ 8:	department: "Research" age: 60.9	
▼ 9:	department: "Sales" age: 60.92	

Figure 7: Résultat de la requête pour obtenir l'âge moyen par département

2.4 Partie administrateur

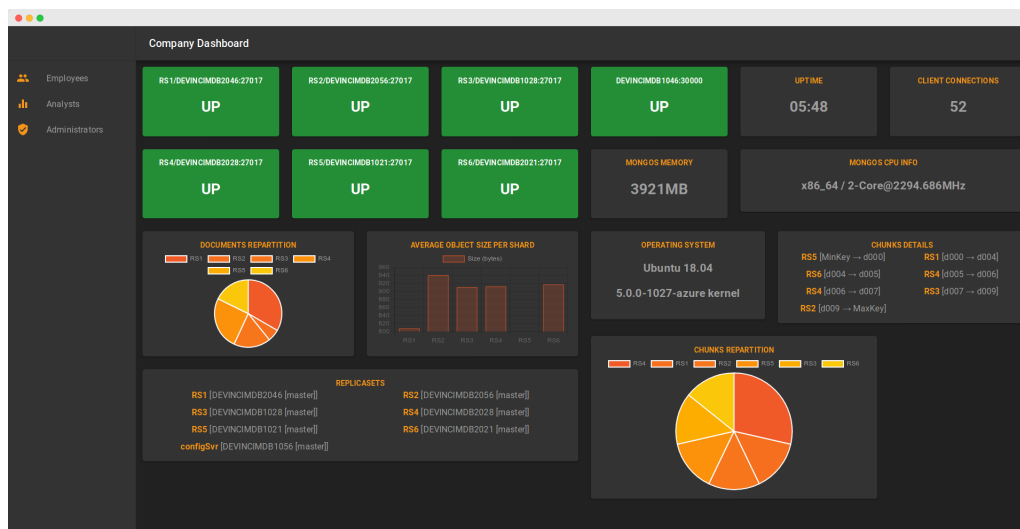


Figure 8: Dashboard administrateur

La partie administrateur étant destinée à obtenir une vision périphérique du statut du cluster, nous avons immédiatement pensé à Grafana (<https://grafana.com/>), une solution de monitoring open-source. Nous avons ainsi implémenté de multiples routes sur l'API permettant d'obtenir des informations relatives à chaque host et à son statut. On peut ainsi savoir lorsque chaque host est up ou down, connaître des informations relatives au uptime de la machine mongos, sa mémoire, son architecture CPU, sa distribution. D'autres routes ont elles été réalisées afin d'obtenir des informations sur la répartition des documents et des chunks dans les shards, afin de réaliser les graphiques ci-dessus, et de voir la répartition des documents dans les shards à travers les informations relatives aux intervalles de la clé de sharding.



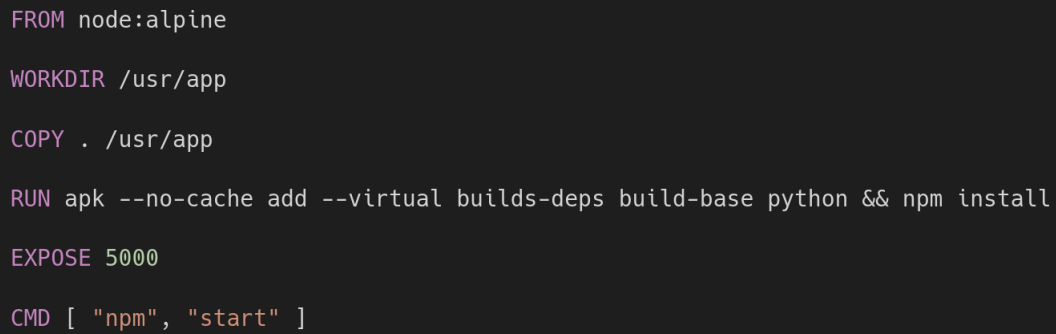
Figure 9: Dashboard Grafana

3 Conclusion

Chaque partie de ce projet nous a montré les difficultés à mettre en oeuvre une solution NoSQL de qualité à partir d'une base de données relationnelle. Nous avons mis un peu de temps à comprendre comment créer un cas d'usage pertinent, et les requêtes respectives. Cependant, dès la partie Dénormalisation, nous avons compris les enjeux du passage au NoSQL et amélioré notre cas d'usage. Cette partie a été pour nous la plus difficile, d'autant plus que celle-ci se devait d'être efficace en termes de coûts réseaux, tout en gardant les informations nécessaires.

Nous avons aimé ce projet car il nous a permis, à travers chacune des étapes, de réaliser un véritable passage à l'échelle tel qu'il serait dans une entreprise. Cette autonomie nous a également montré les problématiques liées à chaque étape.

Nous avons particulièrement apprécié les phases de Scalability et d'Application pour le Cloud, car celles-ci sont en quelque sorte la finalité du projet. La gestion de serveurs virtuels a permis d'améliorer nos connaissances dans un environnement Linux, et de mettre en place des services systemd pour démarrer mongos/mongod au boot de la VM. Les tests de performance pour obtenir le nombre optimal de shards ont également été intéressants. La phase d'application pour le Cloud nous a permis de mettre en place un API ainsi qu'un client React derrière un serveur Nginx, chacun au sein d'un container, qui nous a simplifié le processus de développement. L'aspect visuel et statistique des deux dernières parties a été appréciable car il a permis d'afficher un rendu visuel sur le travail accompli et sur le jeu de données dans son intégralité.



```
FROM node:alpine

WORKDIR /usr/app

COPY . /usr/app

RUN apk --no-cache add --virtual builds-deps build-base python && npm install

EXPOSE 5000

CMD [ "npm", "start" ]
```

Figure 10: Dockerfile API

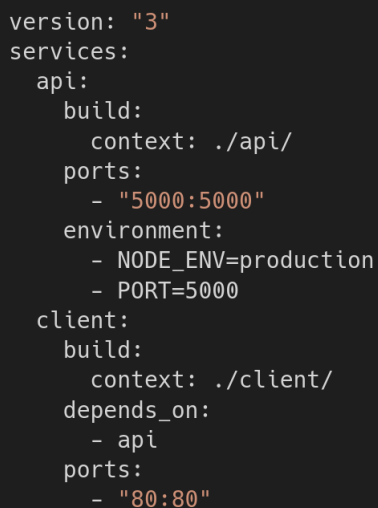


```
FROM nginx:alpine

COPY ./build /var/www
COPY nginx.conf /etc/nginx/nginx.conf

EXPOSE 80
ENTRYPOINT ["nginx", "-g", "daemon off;"]
```

Figure 11: Dockerfile client



```
version: "3"
services:
  api:
    build:
      context: ./api/
    ports:
      - "5000:5000"
    environment:
      - NODE_ENV=production
      - PORT=5000
  client:
    build:
      context: ./client/
    depends_on:
      - api
    ports:
      - "80:80"
```

Figure 12: Docker Compose file

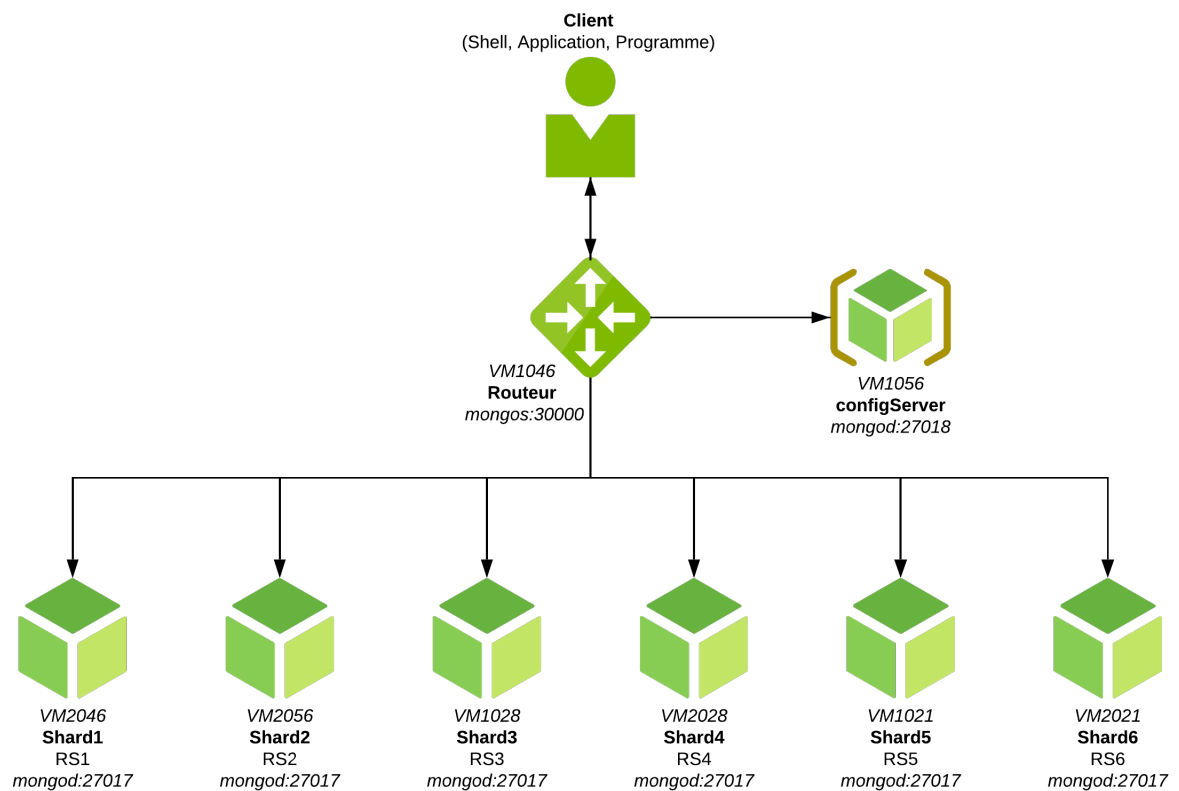


Figure 13: Diagramme de l'architecture MongoDB