GROUPE
Gaétan MORLET
Maxence GRIAS
Maxence PELIGRY
Paul RIVIERE

2021

4PROJ – IoT and the new consumption modes of the future

Rendu technique sur la partie applicative mise en place dans le cadre d'un POC.

RENDU AVANT LE : 04/07/2021

SUPINFO
TOURS

Site retenu : brilliant-market.com

Sommaire

1. Frontend

A. ReactJS

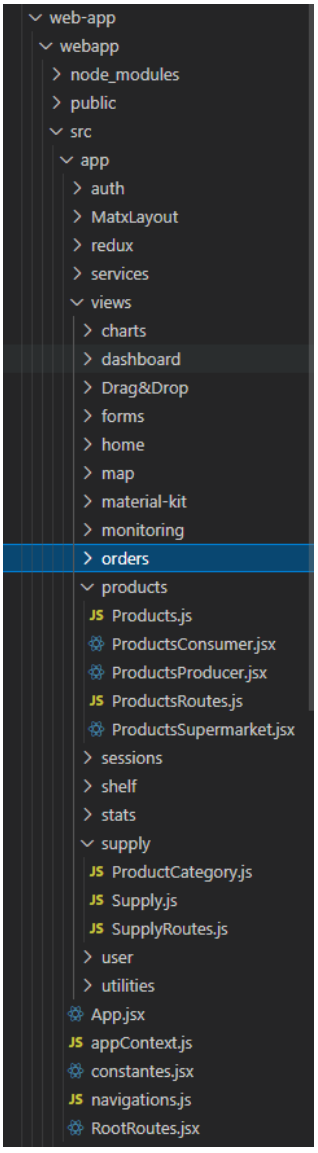
Pour l'application on a choisi de la faire avec la bibliothèque ReactJS. Cette bibliothèque est très récente et a beaucoup de documentation, limitant les problèmes.

Après avoir créé le projet, on a juste besoin de le lancer avec la commande « npm start »

```
PS K:\Cours\4PROJ\web-app\webapp> npm start
```

Cette commande lance un server local sur le port 3000.

1. Organisation du projet



Pour organiser notre projet on a choisi cette arborescence. Ça nous a permis de bien repérer toutes les pages sans soucis.

2. Exemple de code

Pour cet exemple on va regarder les pages « Supply » et « Products ».

a) **Supply**

Dès qu'on arrive sur la page on fait un appel api :

```
componentDidMount() {
    StockService.getLots().then(
        res => {
            res.json().then(
                response => {
                    if (res.ok) {
                        console.log('getLots response : ', response);
                        this.setState({lots: response.lots})
                    } else {
                        console.log("getLots failed : ", response.error);
                        // this.setState({displayError: true});
                        // this.setState({errorMessage: response.error});
                    }
                },
                error => {
                    console.log('getLots parse error : ', error);
                }
            );
        },
        err => {
            console.log('getLots error : ', err);
        }
    )
```

Ici on va récupérer tous les lots et les stocker dans une variable du state. Cette variable est accessible sur toute la page et, dès qu'on la modifiera mettre à jour la page. « componentDidMount() » est une fonction qui se lance dès qu'on arrive sur la page.

Sur la page on affiche un tableau :

```
{this.state.products.length !== 0 &&
<AgGridReact
    onGridReady={this.onGridReady}
    onColumnResized={onColumnResized}
    onRowClicked={this.onClicked}
    rowData={this.state.lots}
    cellStyle={rowStyle}
    columnDefs={colDef}
    gridOptions={gridOptions}
>
</AgGridReact>}
```

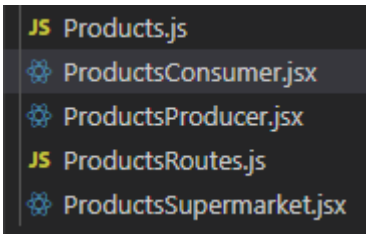
On l'affiche uniquement si la variable du state « products » n'est pas vide. Ce tableau est fait avec le composant « AgGridReact » de la librairie « ag-grid-react ». Les données sont dans l'attribut « rowData ».

« columnDefs » permet de configurer les colonnes (choisir le nom, l'attribut à afficher, un formatter ou autres.

```
const colDef = [
    {
        "headerName": "Produit",
        "field": "idProduit",
        "flex": "1",
        "minWidth": "100",
        "resizable": true,
        "suppressMovable": true,
        "autoHeight": true,
        valueFormatter: (params) => {
            let produit = this.getProduitById(params);
          return produit.name;
        }
    },
```

b) **Products**

Le gros changement par rapport à la page précédente est le fait qu'on va faire appel à d'autres pages jsx.



On va avoir une page « parent » (Products.js) qui va faire appel à « ProductsConsumer.jsx », « ProductsProducer.jsx » ou « ProductsSupermarket.jsx » en fonction du rôle de l'utilisateur.

```
{this.state.user.role === 'consumer' && this.state.products.length !== 0 &&
    <ProductsConsumer products={this.state.products} style={{overflowY: "auto"}} user={this.state.user}/>
}
{this.state.user.role === 'producer' && this.state.products.length !== 0 &&
    <ProductsProducer products={this.state.products} user={this.state.user}/>
}
{this.state.user.role === 'supermarket' && this.state.products.length !== 0 &&
    <ProductsSupermarket products={this.state.products} style={{overflowY: "auto"}} user={this.state.user}/>
}
```

On va ensuite afficher des « Card » pour chaque produit.

```
{this.state.products.map((item, index) =>
  <Grid item xs key={index}>
    <Card elevation={6} className="px-24 py-20 h-100"  style={{minWidth: 250}}>
      <div className="card-title">{item.name}</div>
      <br/>
      <div className="card-subtitle mb-24">{item.producer}</div>
      <div className="card-subtitle mb-24">{item.prix}€</div>
      <Button onClick={this.onClicked.bind(this,item)}>Details</Button>
    </Card>
  </Grid>
)}
```

B. Matx react

Nous avons fait le choix de partir d'un template pour réaliser notre application web. Ce template intègre des composants de material-ui, et permet facilement d'utiliser un thème partout dans l'application. Nous avons cependant refondu le système d'authentification, créé nos services, et créé nos pages. Nous avons pu ainsi faciliter la gestion de rôles sur l'application, ainsi que le routage.

2. Backend

Comme les autres parties du projet applicatif, le backend est dockerisé, et déployé sur kubernetes sous le même namespace que les autres déploiements.

Pour communiquer avec la base de données, nous avons fait le choix d'utiliser le driver fourni par MongoDB.

A. NodeJS

1. NodeJS avec Express

Nous avons fait le choix d'utiliser pour notre backend NodeJS avec le framework Express, pour créer une api.

Cette technologie est simple d'utilisation, et ne requiert pas beaucoup de configuration. On peut avoir facilement la main sur les configurations et redirections, ce qui a conforté notre choix.

Nous avons ainsi pu constituer nos routes, avec fichiers métier correspondants, tout en intégrant une partie de sécurité.

2. Sécurité

A l'enregistrement d'un compte un hash en sha512 est généré pour le mot de passe, avec un salt.

Ce hash/salt est utilisé au moment de la connexion pour générer un jeton d'authentification, comportant les informations principales de l'utilisateur, qui est renvoyé au frontend. Le jeton contenant l'id de l'utilisateur en base, il est ensuite simple de vérifier si le jeton est valide, et s'il a par exemple été modifié ou s'il n'est pas valide, grâce au hash récupéré en base.

Ce token est demandé sur toutes les routes de l'api, protégeant ainsi de toute utilisation par quelqu'un n'ayant pas le rôle demandé, ou pas de compte utilisateur.

B. MongoDB

1. Choix technologique

La base de données (BDD) a pour but de stocker les données de notre solution. L'interaction avec cette base se fera via une API en NodeJS. Nous souhaitions avoir plusieurs instances de cette base de données pour avoir une haute disponibilité et que la solution soit scalable horizontalement. Nous souhaitons également avoir une BDD permettant une modification facile du schéma de données.

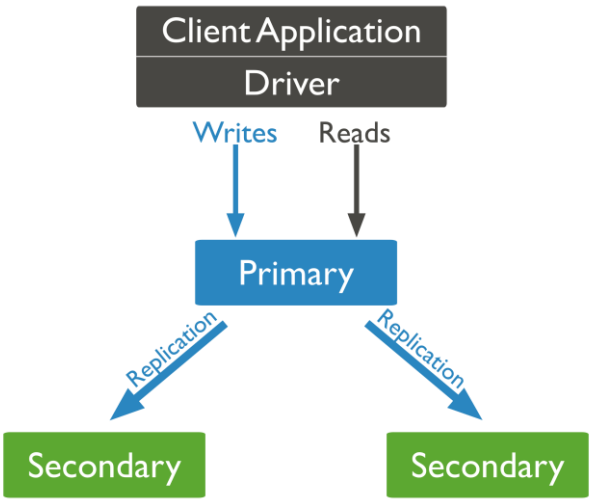
Pour répondre à ce besoin nous avons choisi MongoDB, base NoSQL orientée documents que les développeurs de l'équipe connaissent déjà. La base restitue les données sous format JSON ce qui facilite le développement de l'API qui elle-même restitue les données sous format JSON au front-end. Il y a donc moins de formatage à effectuer dans l'API. Le schéma de données est très souple et permet de le faire évoluer facilement (nous pouvons même faire cohabiter plusieurs schémas de données), là où avec une BDD relationnelle cela aurait été impossible. Pour finir MongoDB met à dispositions des patterns d'architectures permettant la réplication, haute disponibilité et scalabilité horizontale.

2. Architecture

Replica Set

Un Replica Set, correspond à un ensemble de processus mongod (d pour daemon) qui hébergent les mêmes données. Le premier mongod, le nœud primaire, reçoit toutes les opérations de lecture et écriture. Les autres instances, les nœuds secondaires, répliquent les instructions reçues par le membre primaire afin d'avoir exactement le même ensemble de données. Un Replica Set ne peut avoir qu'un seul nœud primaire uniquement qui accepte les opérations d'écriture. Afin de supporter la réplication, le membre primaire enregistre tous les changements réalisés sur le nœud dans un

fichier appelé "oplog". Les nœuds secondaires s'appuient sur ce fichier pour répliquer les opérations reçus sur le nœud primaire et ainsi appliquer les changements.



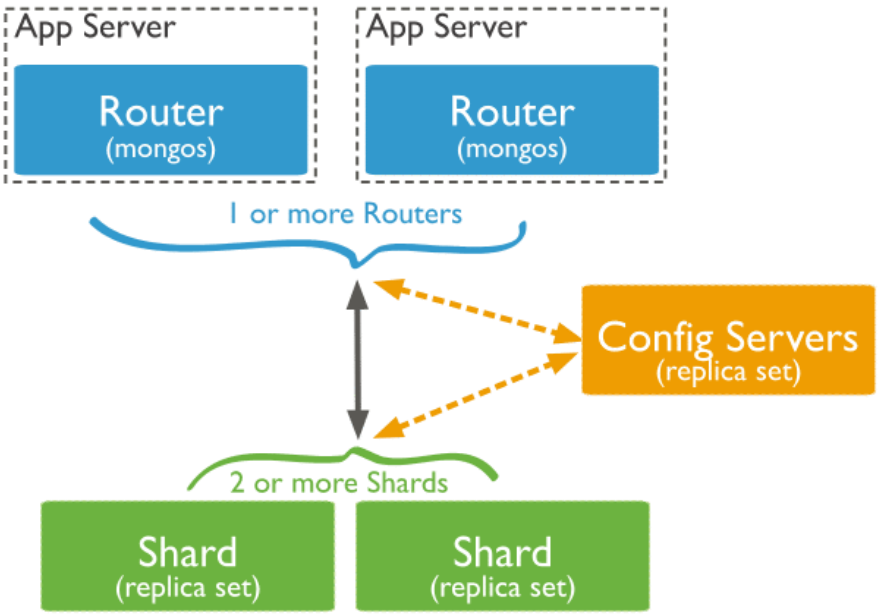
Il faut au minimum trois nœuds pour fonctionner : un primaire et deux secondaires. En cas de perte du nœud primaire, une élection se produit entre les deux nœuds secondaires pour élire le nouveau nœud primaire.

Pour explication plus détaillée, le lecteur se référera à la documentation officielle sur le sujet.

Sharding

Un Replica Set n'offre que de la scalabilité verticale. La Sharding est l'étape supérieure qui permet la scalabilité horizontale pour tenir la charge. C'est une méthode pour partitionner les données sur différents Replica Set. Il faut définir une clé de partitionnement (qui peut être composée) pour chaque collection ce qui permet de répartir la charge sur les différents Replica Set, qui eux-mêmes se chargeront de la haute disponibilité et de la réplication.

Un cluster MongoDB utilisant le Sharding est composé d'au moins deux Replica Set, un processus mongos (s pour Sharding) et un Replica Set de configuration dédié à contenir des informations internes pour le bon fonctionnement du cluster.

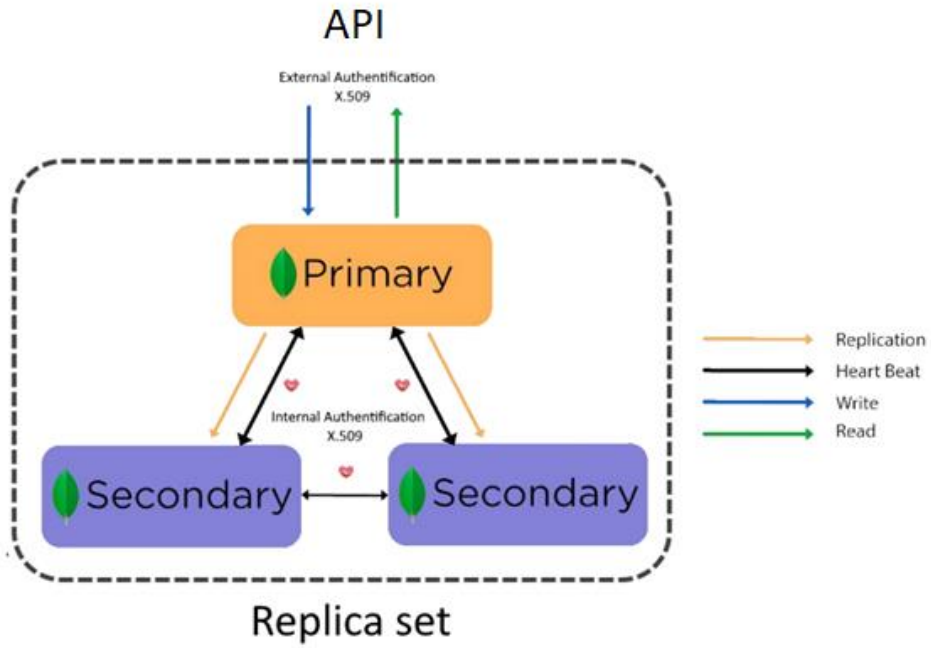


Pour explication plus détaillée, le lecteur se référera à la documentation officielle sur le sujet.

Solution technique

L'architecture retenue est celle d'un replica set composé de 3 membres (1 primaire et 2 secondaires). Celui-ci nous offre une haute disponibilité (et réplication) avec un failover automatique géré nativement par MongoDB (élection d'un nouveau membre primaire si plus des N/2 membres sont accessibles). L'authentification interne (entre les serveurs du cluster) et externe (client qui se connecte au cluster) suivent la norme X.509.

Voici le schéma qui résume cette solution.

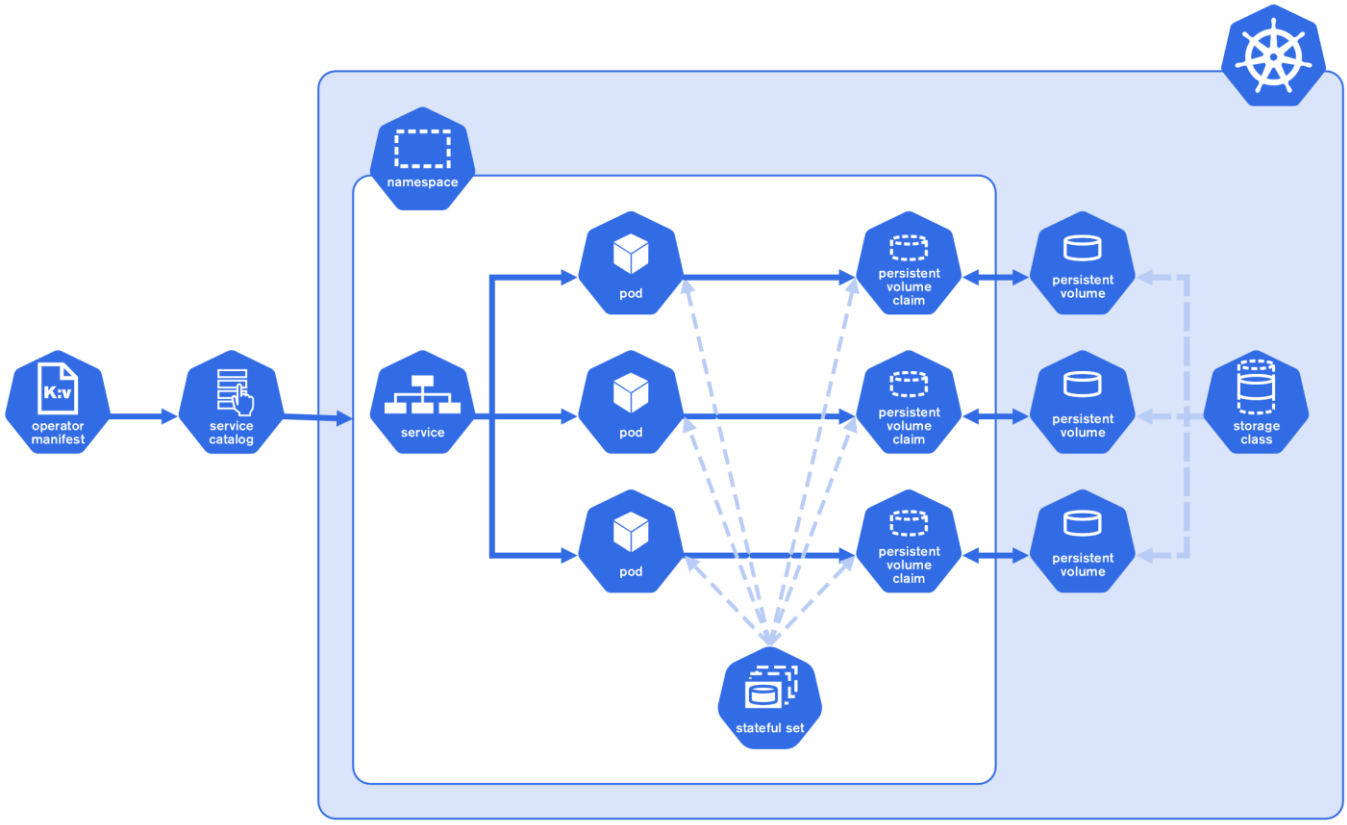


Cette solution nous donne également la possibilité de pourvoir passer d'un simple Replica Set à un cluster avec du Sharding si la solution commence à avoir beaucoup d'écritures. Il faudra alors réfléchir

à quelle clé de partitionnement ("sharding key") utiliser, le champ "_id " parait d'ores et déjà être un bon choix.

3. MongoDB Community Kubernetes Operator

Notre infrastructure étant basée sur un cluster Kubernetes (K8s) nous avons dû trouver un moyen pour déployer MongoDB dessus. Ce point critique fu étudier dès le début et nous nous sommes rapidement rendu compte que MongoDB met à disposition un opérateur K8s facilitant l'opération. Il existe deux versions de l'opérateur, une Community et une Entreprise, nous avons choisi la première car elle était gratuite et open source, nous permettant d'effectuer des modifications sur le code source par nous-même si nécessaire.



Pour l'implémentation technique, le lecteur se référa à la documentation technique de l'infrastructure, à la section MongoDB.

3. Github

Pour structurer nos projets on a choisi de les mettre sur Github. Cela nous a permis d'être toujours à jour quant à ce que faisait tout le monde. De plus, cette solution est gratuite, sécurisé et connue.