2019-2020

Documentation technique

Robin pierrat 293042 – Axel Kimmel 295114 – Clément teyssandier 292621

Table des matières

[I. Application mobile 2](#_Toc43048840)

[A. Information 2](#_Toc43048841)

[B. Prérequis 2](#_Toc43048842)

[C. Installation 2](#_Toc43048843)

[D. Lancer l'application sur un émulateur ou smartphone Android 2](#_Toc43048844)

[E. Créer le bundle Android 4](#_Toc43048845)

[F. Lancer l'application sur un simulateur/smartphone IOS 6](#_Toc43048846)

[G. Créer le bundle IOS 6](#_Toc43048847)

[H. Technologies 6](#_Toc43048848)

[II. API 7](#_Toc43048849)

[A. go-bongo 7](#_Toc43048850)

[B. main.go 7](#_Toc43048851)

[C. Les structs 8](#_Toc43048852)

[D. Usage 10](#_Toc43048853)

[III. Architecture 14](#_Toc43048854)

[A. Environnement de développement 14](#_Toc43048855)

[B. Environnement de production 15](#_Toc43048856)

[IV. Panel administrateur par internet 16](#_Toc43048857)

[A. Introduction 16](#_Toc43048858)

[B. Connexion 16](#_Toc43048859)

[C. Dashboard 16](#_Toc43048860)

[V. Sauvegarde de la base de données 17](#_Toc43048861)

[A. Introduction 17](#_Toc43048862)

[B. Dump de la base de données 17](#_Toc43048863)

[C. Envoie du fichier vers le stockage 17](#_Toc43048864)

[D. Lancement du processus régulièrement 17](#_Toc43048865)

## Application mobile





### Information



Le développement de cette application a été faite en React-Native enfin de pouvoir toucher un plus large public (IOS/Android) et simplifier le développement cross-platform.

L'application consiste à faire ses courses dans le magasin via son smartphone. L'utilisateur pourra voir les produits de chaque rayon sur son smartphone, grâce à un scan qui détecte les rayons, puis il pourra les ajoutez à son Smartpanier. Une fois ses courses finies, il pourra payer en ligne.

Un apk pour android est disponible "supmarket.apk", il faut le transférer sur un smartphone et lancer l'installation. Une fois l'apk installé, il vous sera possible de tester l'application. Un compte de teste est disponible pour tester l'application sans être bloqué par le scan des rayons. Les scans bluetooth seront désactivés, les rayons pourront être choisis manuellement.

Compte teste - Id: apptest@apptest.fr, Mdp: apptest4

### Prérequis



Il est nécessaire de suivre les étapes d'installation de [React Native Environement Setup 'CLI Quickstart'](https://reactnative.dev/docs/environment-setup) suivant votre environnement et votre cible d'OS.

### Installation



Clonez le repertoire du git



$ git clone https://github.com/Robin-Pierrat/4PJT-MobileApp.git

Installez toutes les dependences du projet



* cd \4PJT-MobileApp $ npm install

### Lancer l'application sur un émulateur ou smartphone Android

Spécifiez dans le dossier android le chemin du SDK dans un fichier local.properties



$ cd android

$ touch local.properties



#/android/local.properties

sdk.dir=C\:\\Users\\Robin\\AppData\\Local\\Android\\Sdk

Nettoyez le compilateur android *gradle*



$ ./gradlew clean

Attention, le smartphone doit être en mode développeur avec le débogage USB actif et branché en transfert de fichier

Utilisez *adb* pour voir et récupérer les id des emulateurs/smartphone connectés



$ adb device

Lancez l'application sur le smartphone



$ react-native run-android --deviceId=XXXXXXXXXXXX

### Créer le bundle Android

Générez une clé avec l'outil **keytool** (Pour Windows dans C:\Program Files\Java\jdkx.x.x\_x\bin) Vous allez devoir entrer plusieurs informations !!! A retenir !!!



$ keytool -genkey -v -keystore VOTRE\_CHEMIN\supmarket-key.keystore -alias supmarket-key-alias -keyalg RSA -keysize 2048 -validity 10

Récupérez le fichier crée à l'endroit ou vous avez choisi et le déplacer dans l'application à ./android/app

Dans android/gradle.properties, ajoutez les lignes suivante en remplacant par vos informations:



MYAPP\_RELEASE\_STORE\_FILE=supmarket-key.keystore

MYAPP\_RELEASE\_KEY\_ALIAS=supmarket-key-alias

MYAPP\_RELEASE\_STORE\_PASSWORD=\*\*\*\*\*

MYAPP\_RELEASE\_KEY\_PASSWORD=\*\*\*\*\*

Modifiez android/app/build.gradle avec les nouvelles valeurs de la release



* android/app/build.gradle

...

android {

...

defaultConfig { ... } signingConfigs {

release {

if (project.hasProperty('MYAPP\_RELEASE\_STORE\_FILE')) {

storeFile file(MYAPP\_RELEASE\_STORE\_FILE)

storePassword MYAPP\_RELEASE\_STORE\_PASSWORD

keyAlias MYAPP\_RELEASE\_KEY\_ALIAS

keyPassword MYAPP\_RELEASE\_KEY\_PASSWORD

}

}

}

buildTypes {

release {

...

signingConfig signingConfigs.release

}

}

}

...

Puis lancez le build de l'apk, qui se trouvera dans le dossier /android/app/build/outputs/apk/



$ cd android && ./gradlew assembleRelease

### Lancer l'application sur un simulateur/smartphone IOS

Pour lancer l’application sur un simulateur, vous devez être sur mac et avoir configurer XCode. Ensuite, lancer :

react-native run-ios

Pour lancer sur un smartphone IOS, il faut un compte Apple Développer. (License 99$)

### Créer le bundle IOS

Il faut un compte Apple Développer. (License 99$) et lancer le build du bundle via Xcode.

### Technologies



SupMarket utilise plusieurs projets open source

* [Node](https://nodejs.org) - Node.js JavaScript runtime.
* [React Native](https://reactnative.dev) - A framework for building native apps with React.
* [React Native BLE PLX](https://github.com/Polidea/react-native-ble-plx) - React Native Bluetooth Low Energy library.
* [Native Base](https://nativebase.io) - Essential cross-platform UI components for React Native.
* [React Native App Intro Slider](https://github.com/Jacse/react-native-app-intro-slider) - An easy-to-use yet very configurable app introduction slider/swiper.

## API

API go pour 4PROJ. Cette API permet d'effectuer des intéractions avec la base de données MongoDB.

### go-bongo

Cette API utilise l'ODM *"Bongo"* qui permet de simplifier l'utilisation de *mgo* et les intéractions avec *MongoDB*

Description de *Bongo* : "We couldn't find a good ODM for MongoDB written in Go, so we made one. Bongo is a wrapper for mgo (<https://github.com/go-mgo/mgo>) that adds ODM, hooks, validation, and cascade support to its raw Mongo functions."

*Bongo* est disponible sur GitHub (<https://github.com/go-bongo/bongo>) et est donc libre d'utilisation.

### main.go

Pour utiliser l'api, il faut la build puis l'exécuter. Une fois exécutée elle écoutera sur l'IP locale et le port 8080 (modifiable), et sera en attente d'une requête à traiter.

Les routes suivantes existes :

*//routes "globales" (applicables à tous les types d'objets)*  
xxx.xxx.x.xx:8080/insert  
xxx.xxx.x.xx:8080/update  
xxx.xxx.x.xx:8080/delete  
xxx.xxx.x.xx:8080/find  
  
*//routes "ciblées" (spécifiques à certains types d'objets)*  
xxx.xxx.x.xx:8080/register  
xxx.xxx.x.xx:8080/login  
xxx.xxx.x.xx:8080/transactions  
xxx.xxx.x.xx:8080/transactionDetails  
xxx.xxx.x.xx:8080/beacons  
xxx.xxx.x.xx:8080/products  
xxx.xxx.x.xx:8080/changeAdmin  
xxx.xxx.x.xx:8080/isAdmin  
xxx.xxx.x.xx:8080/getClients  
xxx.xxx.x.xx:8080/recommandations

Les routes dites *globales* permettent d'effetuer les opérations de base pour MongoDB.

### Les structs

Les structs suivantes existent, elles sont modifiables et il est possible d'en rajouter si nécessaire.

*//PostObject : objet de base*  
**type** PostObject **struct** {  
 Token string  
 ObjectType string  
}  
  
*//Client : client du magasin*  
**type** Client **struct** {  
 bongo.DocumentBase `bson:",inline"`  
 ID int `json:"ID"`  
 Username string `json:"Username"`  
 PwHash string `json:"PwHash"`  
 IsAdmin bool `json:"IsAdmin"`  
 Firstname string `json:"Firstname"`  
 Lastname string `json:"Lastname"`  
 Token string `json:"Token"`  
}  
  
*//Transaction : achats d'un client*  
**type** Transaction **struct** {  
 bongo.DocumentBase `bson:",inline"`  
 ID int `json:"ID"`  
 Token string `json:"Token"`  
 Amount float32 `json:"Amount"`  
 RealDate time.Time `json:"RealDate"`  
 ItemsID []int `json:"ItemsID"`  
}  
  
*//Promotion : promotion ponctuelle d'un ou plusieurs produits*  
**type** Promotion **struct** {  
 bongo.DocumentBase `bson:",inline"`  
 ID int `json:"ID"`  
 StartDate time.Time `json:"StartDate"`  
 EndDate time.Time `json:"EndDate"`  
 Description string `json:"Description"`  
 Price float32 `json:"Price"`  
 ItemsID []int `json:"ItemsID"`  
}  
  
*//Item : objets vendus, composés de produits*  
**type** Item **struct** {  
 bongo.DocumentBase `bson:",inline"`  
 ID int `json:"ID"`  
 ProductID int `json:"Products"`  
 Quantity int `json:"Quantity"`  
 BarCode string `json:"BarCode"`  
 Price float32 `json:"Price"`  
}  
  
*//Product : produits présents en stock, compose les items*  
**type** Product **struct** {  
 bongo.DocumentBase `bson:",inline"`  
 ID int `json:"ID"`  
 Name string `json:"Name"`  
 Brand string `json:"Brand"`  
 Price float32 `json:"Price"`  
 CategoryID int `json:"CategoryID"`  
 QuantityStock int `json:"QuantityStock"`  
}  
  
*//Category : catégorie de produit*  
**type** Category **struct** {  
 bongo.DocumentBase `bson:",inline"`  
 ID int `json:"ID"`  
 Name string `json:"Name"`  
 ShelfID int `json:"ShelfID"`  
}  
  
*//Beacon : balise rattachée à un rayon*  
**type** Beacon **struct** {  
 bongo.DocumentBase `bson:",inline"`  
 ID int `json:"ID"`  
 MAC string `json:"MAC"`  
 EntryBeacon bool `json:"EntryBeacon"`  
 Name string `json:"Name"`  
 ShelfID int `json:"ShelfID"`  
 PosX float32 `json:"PosX"`  
 PosY float32 `json:"PosY"`  
 LastSignalDate time.Time `json:"LastSignalDate"`  
}  
  
*//Shelf : rayon du magasin*  
**type** Shelf **struct** {  
 bongo.DocumentBase `bson:",inline"`  
 ID int `json:"ID"`  
 Name string `json:"Name"`  
 SupermarketID int `json:"SupermarketID"`  
 PosX float32 `json:"PosX"`  
 PosY float32 `json:"PosY"`  
}  
  
*//Supermarket : magasin*  
**type** Supermarket **struct** {  
 bongo.DocumentBase `bson:",inline"`  
 ID int `json:"ID"`  
 Address string `json:"Address"`  
 Sign string `json:"Sign"`  
}

### Usage

#### Routes globales

Toutes les routes *globales* s'utilisent de la même manière.

Les données à enregistrer en bdd ou à utiliser pour récupérer des objets de la bdd sont à fournir en json.

Il est nécessaire de fournir une variable **ObjectType** qui définira le type de l'objet et un **Token** valide (token d'un client).

Ainsi que **les données de l'objet**

**L'ID est défini automatiquement lors de l'insertion d'un objet**

{  
 "ObjectType": "Client",  
 "Token": "..."  
}  
{  
 "Username": "Axel",  
 "PwHash": "...",  
 "IsAdmin": true,  
 "Firstname": "Axel",  
 "Lastname": "Kimmel"  
}

Pour les opérations de *deletion (/delete)* et *récuperation (/find)* **fournir l'ID de l'objet suffit**.

Pour l'opération de récupération et il est possible de récupérer **tous les objets d'une collection en donnant -1 pour ID**.

{  
 "ObjectType": "Client",  
 "Token": "..."  
}  
{  
 "ID": -1  
}

#### Routes ciblées

Register

Pour le *register (/register)*, il est nécessaire de fournir uniquement les infos du compte client à l'exception de l'ID et du Token qui seront générés automatiquement.

Ici, ObjectType n'est pas nécessaire

{  
 "Username": "axel",  
 "PwHash": "password",  
 "IsAdmin": true,  
 "FirstName": "Axel",  
 "LastName": "Kimmel"  
}

Login

Pour le *login (/login)*, il est nécessaire de fournir uniquement le username et le hash du password (PwHash), si la combinaison est correcte, le token du client sera retourné.

{  
 "Username": "axel",  
 "PwHash": "password"  
}

Transactions

Les *transactions (/transactions)* sont récupérables en fournissant uniquement le token du client concerné.

{  
 "Token": "..."  
}

Détails d'une transaction

Les *détails d'une transactions (la transaction et les items qui la composent) (/transactionDetails)* peuvent être obtenus en fournissant le token du client ainsi que l'ID d'une transaction. Il est possible de passer -1 en ID pour obtenir la dernière transaction effectuée

{  
 "Token": "..."  
}  
{  
 "ID": 1  
}

Balises

Les *beacons (/beacons)* d'un magasin sont récupérables en fournissant un token valide et l'id du magasin. A noter que la balise d'entrée est identifié par le champ *EntryBeacon* à *true*.

{  
 "Token": "..."  
}  
{  
 "ID": 1  
}

Rayons

Il est possible d'obtenir les *products (/products)* présents sur un rayon en fournissant un token valide ainsi que l'id de la balise rattachée au rayon.

{  
 "Token": "..."  
}  
{  
 "ID": 1  
}

Changement Admin

Il est possible de changer le statut *IsAdmin (/changeAdmin)* d'un compte en fournissant le token du compte. A noter que le changement peut se faire dans les deux sens *true->false et false->true*.

{  
 "Token": "..."  
}

Is Admin

Il est possible d'obtenir le statut *IsAdmin (/changeAdmin)* d'un compte en fournissant le token du compte.

{  
 "Token": "..."  
}

Get Clients

Il est possible d'obtenir les informations des différents clients *(/getClients)* en fournissant un token valide. Ici, c'est un tableau des Clients qui est retourné.

{  
 "Token": "..."  
}

Get Beacons

Il est possible d'obtenir les informations des différentes balises *(/getBeacons)* en fournissant un token valide. Ici, c'est un tableau des Balises qui est retourné.

{  
 "Token": "..."  
}

Recommandations

Il est possible d'obtenir des *recommandations (/recommandations)* pour un client en fournissant son token. Les recommandations sont basées sur les transactions passées d'un client ainsi que des promotions actuelles.

{  
 "Token": "..."  
}

## Architecture

### Environnement de développement

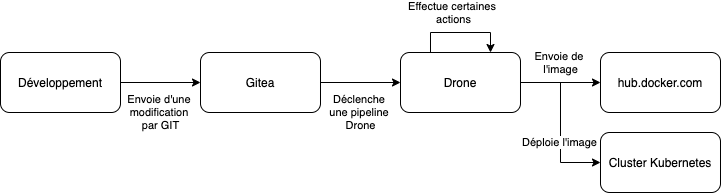
Une image contenant équipement électronique

Description générée automatiquement

Pour développer l’application, ainsi que pour apporter des modifications par la suite aux applications proposées, nous avons mis en place un serveur de développement.

Ce serveur est une instance n1-standard-1, hébergée sur Google Cloud. Il héberge un service de versioning Gitea et un service de CI/CD (continuous integration, continuous development), sous forme de conteneurs Docker.

Ces services permettent de développer avec une logique DevOps, c’est-à-dire qu’ils permettent d’automatiser un maximum le développement.



Lorsqu’on publie une modification vers Gitea, une pipeline Drone se déclenche et effectue différentes actions (cela dépend de l’application), puis envoie l’image Docker sur hub.docker.com et publie ensuite l’image vers le serveur de production (décrit ci-dessous).

### Environnement de production

Une image contenant équipement électronique

Description générée automatiquementL’environnement de production est totalement indépendant et différent de l’environnement de développement. Il est également hébergé sur Google Cloud, mais sous forme d’un cluster Kubernetes.

Ce cluster est composé d’un minimum de 3 nœuds, qui peuvent être augmentés en fonction des besoins et de la charge.

Nous avons décidé d’utiliser un minimum de load balancers, par soucis d’économies et de simplicité. Nous avons donc deux load balancers, qui se chargent de diriger le trafic.

Le premier load balancer, redirige le trafic vers un service de reverse proxy nommé Traefik, il permet de rediriger les requêtes vers les autres services, en fournissant du HTTPS (avec des certificats générés par Let’s Encryt). Traefik permet donc de rediriger les requêtes à destination d’un certain service, en fonction du sous domaine du DNS de la requête, en fonction des routes que nous définissons dans un fichier de route Ingress de Kubernetes.

Le second load balancer, sert à combler une des seules limites de Traefik, qui permet uniquement le traitement des requêtes TCP HTTP(S), en redirigeant les requêtes vers la base de données. En effet, la base de données Mongo DB utilise bien le TCP pour communiquer, mais avec son propre protocole « mongodb ».

Une image contenant capture d’écran

Description générée automatiquement

Le reste du cluster héberge les services de la base de données MongoDB, l’API, l’interface administrateur et les jobs de backup de la base de données.

## Panel administrateur par internet

### Introduction

Afin d’avoir une vue d’ensemble, simple et rapide, de l’environnement de production, nous avons conçu une interface administrateur, sous la forme d’une application web, utilisant Go et un template front end. L’application est disponible à l’adresse suivante : [https://web.295114-293042-292621-4pjt.tk](https://web.295114-293042-292621-4pjt.tk/).

### Connexion

L’application Go gère les requêtes en redirigeant les personnes connectées (et administrateur) vers le dashboard et les personnes non connectés vers l’index.

Lorsque l’on est sur l’index, l’application Go execute une template html, qui est renvoyée à l’utilisateur.

Lorsqu’il saisit ses identifiants, il post un formulaire vers /login, qui récupère ses identifiants, les envoie à l’API pour vérifier qu’ils soient valides (en hashant le mot de passe). Si tout va bien, l’API retourne le token unique à l’utilisateur.

On teste ensuite avec ce token, en demandant à l’api, si l’utilisateur est bien administrateur. Si c’est le cas, un cookie contenant le token en question est créé et l’utilisateur est redirigé vers le dashboard.

### Dashboard

A l’arrivée sur le dashboard, l’application vérifie que la personne est bien connectée et si elle est bien administrateur, sinon, elle est redirigée vers l’index.

L’application Go exécute une template html, en lui passant un objet contenant certaines informations (décrites ci-dessous). Le template utilise les données passées pour itérer sur les objets et créer un visuel en fonction de qui lui est donné.

#### Récupération des informations des serveurs

Pour construire la page, l’application récupère les informations des différents serveurs.

Pour le serveur de développement, on vérifie qu’il soit en ligne et son temps de réponse. On fournit également des données en dure, tel que le nombre de CPU, la quantité de RAM utilisée, par rapport à la RAM disponible, ces données pourront être récupérées dynamiquement dans l’avenir.

Pour l’environnement de production, on utilise un fichier kubeconfig, pour se connecter au cluster Kubernetes et récupérer dynamiquement des informations sur les nœuds, tel que le nombre de CPU, la quantité de RAM utilisée, par rapport à la RAM disponible, leurs adresses IP.

Cela nous permet ensuite de récupérer d’autres informations tels que leurs statuts (en ligne ou hors ligne) et leurs temps de réponses.

Dans le template HTML, on itère sur le tableau de serveurs, pour afficher les informations de chacun d’en eux.

#### Récupération des informations des balises

Pour construire la page, l’application récupère différentes informations sur les balises en interrogeant l’API. L’application parse ensuite le résultat pour créer un tableau de balises.

Dans le template HTML, on itère sur le tableau de balises, pour afficher les informations de chacune d’elles.

Dans le futur, leurs positions pourront être utilisées pour les placer sur une carte.

#### Récupération des informations des comptes

Pour construire la page, l’application récupère différentes informations sur les comptes en interrogeant l’API. L’application parse ensuite le résultat pour créer un tableau de comptes.

Dans le template HTML, on itère sur le tableau de comptes, pour afficher les informations de chacun d’eux.

Dans le futur, on pourra modifier le statut d’un compte pour le faire passer administrateur ou utilisateur.

## Sauvegarde de la base de données

### Introduction

Afin de sauvegarder la base de données, nous avons décidés d’effectuer les différentes actions nécessaires, sous forme d’un job Kubernetes, qui est lancé tous les jours à 4h30 sur le cluster de production, par Drone.

### Dump de la base de données

Un conteneur Docker Mongodb lance une commande « mongodump » pour récupérer une sauvegarde complète de la base de données.

### Envoie du fichier vers le stockage

Une fois le fichier récupéré, il est copié dans un dossier accessible par l’application Go. Cette application va ensuite se connecter à un Bucket de stockage Google Cloud, grâce à un fichier de configuration. Puis l’application Go lance l’envoie du fichier vers le Bucket.

### Lancement du processus régulièrement

Pour effectuer cette action régulièrement, nous avons défini un cron job dans Drone, qui lance un pipeline qui construit l’image, puis l’envoie sur hub.docker.com et la déploie sur le cluster Kubernetes.

