AQROPOL

Manuel pour la reprise du code

Participants au projet :

Jean-Baptiste Bordier
Manon Derocles
Mélian Ferrachat
Julien Garnier
Alan Gaubert
Adrien Leblanc
Thibault Leclair
Antoine Leval
Pierre Miola
Bashar Nemeh
Noureddine Kadri
Antoine Posnic
Fanny Prieur
AQROPOL
Hantavolaniaina Sylvia Rabe

Mr François Bodin
Enseignant référent du projet

Enseignant chercheur à l'université de Rennes 1(IRISA)

Table des matières

Ce manuel reprend tous les langages et codes utilisés pour l'élaboration du projet AQROPOL (2017-2018).

I. PARTIE CAPTEUR

_____Le développement des différents services s'est fait en JAVA et en utilisant Maven et PostgresSQL.

Vous trouverez le dépôt ci-dessous:

https://github.com/AQROPOL/aqropolNUC

Il faut d'abord initialiser la base de donnée avec le fichier "init.sql". La base comprend 3 tables :

- **nuc** : tables [*id*, *token*] => Stocke le nom du nuc, il n'y a donc qu'une seule ligne.
- **sensor** : tables [*id*, *name*, *type*, *unity*] => Stocke toutes les informations des capteurs connectés (et leur attribue un id). Chaque triplet (*name*, *type*, *unity*) est unique.
- **measure** : tables [*id*, *id_nuc*, *id_sensor*, *datetime*, *hash*, *value*] => Stocke les mesures prises des capteurs. La colonne *hash* est générée automatiquement grâce à une fonction plpgSQL lorsqu'une ligne est ajoutée ou modifiée.

La structure "multi-module" du projet est déterminée selon le standard Maven. Le projet racine "aqropolNUC" contient 4 sous modules. Néanmoins le projet racine possède lui-même comme projet parent Spring Boot. La structure est donc la suivante :

- spring-boot-starter-parent
    - aqropol-NUC
        - aqropolrest
        - measurepmsensor
        - aqropolmomProducer
        - aqropolmomConsumer
        - (+) Arduino

Le dernier répertoire Arduino n'est pas un module car ne nécessite pas d'être construit mais seulement d'être flashé sur la carte Arduino. Pour cela il est nécessaire de télécharger l'IDE Officiel Arduino : https://www.arduino.cc/en/Main/Software

Pour mettre en route les services ainsi que l'environnement de developpement ou de production, veuillez vous référer à la fiche du "Manuel Utilisateur" à la racine du projet : https://github.com/AQROPOL/aqropolNUC/blob/master/Manuel_utilisateur.pdf , le rôle de chaque module y es décrites, et l'essentiel du projet consistant en de la configuration, toutes les informations nécessaire se trouvent donc dans le manuel utilisateur.

Le projet ayant pour parent Spring Boot, il est important de noter que le filtrage des ressources par maven ne s'effectue pas avec le délimiteur par défaut ${ma.ressource} mais avec celui propre à spring : @ma.ressource@.
Tous les paramètres nécessaire sont parmis les properties du pom racine "aqropolNUC" et sont ensuite propagée par filtrage dans les dossiers "src/main/ressources/conf-dev et conf-prod". A noter que l'environnement de dev est celui par défaut.

Le webservice "aqropolrest" est développé avec plusieurs modules du framework SpringBoot ainsi qu'avec la JPA (Java Persistence API).

- Spring Data JPA : https://projects.spring.io/spring-data-jpa/#quick-start
- Spring Data Rest https://projects.spring.io/spring-data-rest/
- Hibernate2 comme implémentation de JPA.
- Spring Security : https://spring.io/guides/gs/securing-web/
  la classe "aqropolNUC/aqropolrest/src/main/java/config/**WebSecurityConfig.java**" permet de configurer la sécurité, les droits d'accès, etc..).
- Spring Actuator Service : https://spring.io/guides/gs/actuator-service/
  (aucun ajout/modification, excepté la sécurité (élémentaire)).

Les entitées de la base de données sont modélisé grâce aux annotations JPA dans le package "aqropolrest/src/main/java/hello/data" conformément au schéma de la base de donnée postgres (Le hash de measure est nullable car il n'est pas généré dans le programme mais à l'insertion dans la base par postgre).

Les répertoires qui permettent les requêtes REST sur ces ressources sont situé dans le package "aqropolrest/src/main/java/hello/data/repository". La représentation des mesures et des capteurs sont modifiés grâce à des projections dans le même package. La documentation de Spring Data Rest et Spring Data JPA pour des requêtes plus spécifique peut être très utile pour poursuivre le code à cet endroit.

Le web service propose en plus des repository, un contrôleur pour la ressource Measure pour permettre la suppression des mesures déjà reçu par le serveur principal : "aqropolrest/src/main/java/hello/**MeasureController.java**"

Le module "aqropolmomConsumer" n'utilise que deux modules de spring boot :
- Spring boot amqp : https://spring.io/guides/gs/messaging-rabbitmq/ ou sur le site de rabbitMQ : https://www.rabbitmq.com/tutorials/tutorial-four-spring-amqp.html
- Spring Data JPA pour l'accès aux données et la persistance comme pour le service web, avec hibernate2 encore une fois.

Le module "measurepmsensor" est à démarrer en ligne de commande avec des paramètres, et utilise donc la librairie commons-cli pour parser les paramètres, puis la librairie "firmata4j" (https://github.com/kurbatov/firmata4j) pour la communication USB.
Le module instancie un nouveau "aqropolmomProducer" pour envoyer les données reçue par l'interface USB.

Le module "aqropolmomProducer" est une librairie permettant d'instancier un nouveau producteur pour rabbitMQ, qui correspond aux paramètres du consommateur grâce aux propriétés du pom racine (voir Manuel Utilisateur). La librairie peut être instancié avec les paramètres par défaut grâce à Maven mais propose également une factory pour en instancier un avec d'autres paramètres.
Voir documentation rabbitMQ : https://www.rabbitmq.com/tutorials/tutorial-four-java.html

II. PARTIE ANDROID

Le développement de l'application Android du projet AQROPOL s'est faite avec le logiciel Android Studio.

Vous trouverez le dépôt ci-dessous:

https://github.com/AQROPOL/AQROPOL_App

Nous avons 4 classes dont 1 manifest en .xml et 5 en .java.

AndroidManifest.xml comportera le nom de l'application ainsi que son icône et toutes les permissions dont:

- La permission d'accès à un réseau Wifi,
- La permission de changement de réseau Wifi,
- La permission d'accès à Internet,
- La permission de lecture et d'écriture.

La classe **AllData.java** contiendra tous les attributs (les urls) ainsi que les méthodes getter et setter qui seront utilisés dans les autres classes.

La classe **AndroidToNuc.java** est l'implémentation de la communication entre l' android et le NUC. Le protocole entre le téléphone et le nuc est le suivant :

- Récupération des mesures de la qualité de l'air du capteur par la méthode HTTP GET et insertion dans un fichier stocké sur la carte SD (si le fichier n'existe pas, il est créé automatiquement à la racine de la carte SD sous le nom de "AQROPOL/data").
- Envoie du tableau de hash par la méthode HTTP POST. Le tableau de hash est récupéré pendant l'envoi des mesures au serveur puis stocké dans un fichier.

La classe **AndroidToServer.java** implémente la communication entre l' android et le serveur. Le protocole entre le téléphone et le serveur est le suivant :

- Envoi du fichier contenant les mesures au serveur par la méthode HTTP POST.
- Récupération du tableau de hash par la méthode HTTP GET. Le tableau de hash est stocké dans un fichier afin que l'utilisateur puisse le récupérer même si il ferme l'application.

La classe **WifiScanner.java** implémente la configuration Wifi.

La classe **MainActivity.java** implémente le traitement des permissions ainsi que la connexion et la déconnexion au Wifi.

III. PARTIE SERVEUR

Lien vers l'API: https://github.com/AQROPOL/aqropol-bdd

Dans cette partie de l'application, nous avons installé un serveur Apache2 sur le Raspberry Pi et implémenté une base de données de type MySQL.

Afin de se connecter au serveur, il suffit de passer par la commande SSH d'Unix via un terminal.

ssh pi@pilic27.irisa.fr -i path/to/cle_rsa

Note : la clé RSA est indispensable pour s'y connecter.

Les pages web du serveur se trouvent à l'emplacement habituel d'un environnement Unix, c'est à dire sous **/var/www/**.
Afin d'effectuer une modification de code en utilisant GitHub, il est préférable de suivre la procédure suivante.

1. Effectuer vos modifications et "push" sur l'espace du projet GitHub.
2. Se connecter au Raspberry Pi, aller dans le dossier ProjetAqropol, puis effectuer dans l'ordre ces commandes suivantes.
3. *eval "$(ssh-agent -s)"* (N'est nécessaire qu'une seule fois par session)
4. *ssh-add aqropol* (N'est nécessaire qu'une seule fois par session)
5. *cd aqropol-bdd*
6. *git pull*
7. *cd ..*
8. *sudo sh update.sh*

Pour plus de simplicité, cette commande vous assurera un bon fonctionnement :

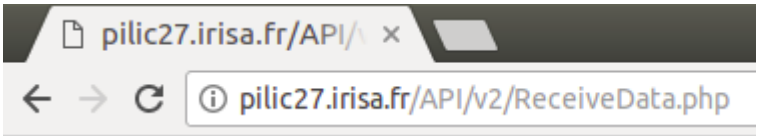
eval "$(ssh-agent -s)" && ssh-add aqropol && cd aqropol-bdd && git pull && cd .. && sudo sh update.sh

Ainsi, le code présent sur GitHub sera copié sur le serveur directement.

Le Serveur fournit une API pour:

- L'insertion des données reçues (Fig.1) :
  - Nom du hub
  - Date d'envoi
  - Nom, type du capteur
  - Unité, valeur de la mesure
  - Date d'enregistrement, coordonnées gps, hash.

- La lecture des informations stockées (Fig.2) : Interroger la base de données selon certains filtres tels que la date d'une mesure, sa valeur, son type, ... pour ensuite les afficher .



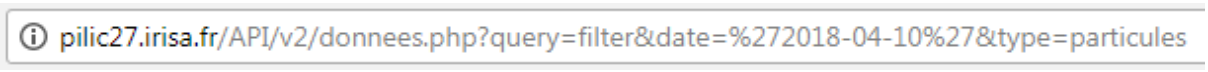


Fig.1: Url permettant une insertion de données
Fig.2: Exemple d'url permettant de questionner la base de données

Nous avons implémenté une base de données MySql grâce au script **creer_db.sql**. La base de données se compose de 4 tables :

- ***hubs***: Il s'agit de la liste des supports sur lesquels sont branchés les capteurs.
- ***capteurs***: Permet de déterminer le type des mesures effectuées et d'en connaître la provenance.
- ***mesures***: Contient l'ensemble des mesures, stockées dans un BLOB, un type de donnée non formatée.
- ***meta_mesures***: Table des informations sur les mesures, contenant l'ensemble des données communes à un jeu de mesures provenant d'un même hub et prises au même moment.

Le fichier **db_access.php** contient les commandes nécessaires afin de se connecter à la base de données et également les requêtes SQL préparées. Ces requêtes servent notamment à recueillir le contenu de différentes tables ou bien à insérer des valeurs.

La communication entre l'application Android et le serveur se fait via les commandes se trouvant dans le fichier **ReceiveData.php** :

- Réception d'une requête POST contenant les différentes informations concernant les Hubs,Capteurs et Mesures sous forme JSON (Fig.3).


```
{
    "nuc": 50,
    "dateenvoie": "2018-04-10 00:00:00",
    "mesures": [
        {
            "capteur": "capteurAir1",
            "type": "pm30",
            "unite": "Q/m3",
            "valeur": 2.3,
            "date": "2018-04-09 00:00:00",
            "lat": 48.1151495,
            "long": -1.74707547,
            "hash": "291873b4e297ed30c25583689452d90c"
        }
    ]
}
```


Fig.3: Exemple de données JSON envoyées par Android vers la Base de données

- Décodage de la variable du POST et procédure d'insertion des données reçues dans la base.
- Envoi des derniers Hash ainsi que des identifiants des Hubs en format JSON vers Android (Fig.4).

```
[
    {
        "id_hub": 1,
        "hash": "291873b4e297ed30c25583689452d90c"
    },
    {
        "id_hub": 2,
        "hash": "4d94bf3f3a436420c31fb4d65a3a6d79"
    },
    {
        "id_hub": 5,
        "hash": "52a4bafd92da0ed96a0c3986b40f22eb"
    }
]
```


Fig.4: Exemple de données JSON envoyées par la Base de données vers Android

L'objectif du fichier **donnees.php** est la gestion des requêtes et récupération des résultats en format JSON afin de les envoyer vers la Visualisation.

Le fichier **script_populace.php** permet de remplir la base de données non réelles et aléatoires. Très utile pour effectuer des tests en lien avec la Visualisation.

IV. PARTIE VISUALISATION

Une fois le serveur avec la base de données correctement installé, la visualisation peut être fonctionnelle. Le code est disponible sur l'organisation github AQROPOL:

https://github.com/AQROPOL/AQROPOLweb

Notre rôle est de récupérer les valeurs mesurées par les capteurs qui ont été stockés dans la Base de données pour afficher publiquement sur une page web les résultats. On les récupère alors sous forme GeoJSON et utilise une carte interactive pour les visualiser dans l'espace.

1. Prérequis

Un quelconque moyen d'héberger un serveur web sous le même nom de domaine que la base de donnée et son API.
Un compte chez MapBox, ainsi que le token valide fournis par ce dernier.

2. Choix Mapbox

Mapbox Maps SDK est une suite de librairies open sources de l'entreprise Mapbox (mapbox.com).
Elle permet la création de cartes interactives sur de nombreuses plateformes (Android, iOS, Linux, macOS, Node.js, Qt, React NAtive et web). Ces cartes basées sur les données OpenStreetMap nous permettent beaucoup de libertés dans la customisation de cartes.
MapBox nous permet notamment:

- L'extrusion 3D, permettant d'afficher les bâtiments.
- La stylisation de la carte pour de véritables web designers qui reprendrait le projet.
- Le support GeoJSON. Un format pour encoder diverses structures de données géographiques, qu'on utilise ici pour géolocaliser les mesures reçues par la base de données.
- Des filtres permettant diverses manières d'afficher les données. On utilise par exemple leur filtre heatmap et circle.
- La possibilité de bloquer la carte sur une zone. Dans notre cas sur la ville de Rennes et ses alentours proches.

De plus, Mapbox contient une documentation fournie exhaustive pour les différents langages. Ainsi que divers exemples, permettant d'apprécier les fonctionnalités disponibles.

3. Setup

Cloner le git AQROPOLweb, https://github.com/AQROPOL/AQROPOLweb et mettre son contenu à la racine du serveur web.

Indiquer votre token mapbox dans le fichier javascript: /sources/js/main.js

```
3   //token mapBox
4   mapboxgl.accessToken =
```

4. Explication Code

Notre implémentation utilise donc l'API js de MapBox[1]. Elle permet la création de map interactives ergonomiques et fortement designable.

a. Initialisation de la map

L'initialisation de la map se fait toujours dans le fichier main JS et appelle la fonction de l'API `new mapboxgl.Map` qui prend en paramètre un tableau avec toutes les options choisies. Nous avons donc centré la map sur Rennes via le paramètre center, choisi des bordures max via le paramètre maxBounds. Il y'a plein de possibilités de personnalisation disponible dans la Doc[2] dans la partie Parameters. Une fois la map initialisée, le fichier main appelle les fichiers MapLayers et MaptoggleLayers via une fonction précise si c'est pour afficher l'ensemble des données récoltées ou si c'est pour afficher les données récoltées pour une date précise ou un mois.

b. Gestion des Requêtes

Le fichier MapLayer est celui qui initialise les calques en fonction de la requête utilisateur. Si il n'y a pas de requête, la fonction appelée est mapLayer(map, requête). Le paramètre requête dans notre cas sera toujours d'afficher toutes les mesures sur présente dans la base de donnée. Cela se fait via un GET sur l'adresse suivante : http://pilic27.irisa.fr/API/v2/donnees.php?query=all

Si une date précise est demandée, la fonction appelée dans MapLayer sera mapLayerDate, qui prend que la map en paramètre et qui va récupérer la date demandée dans l'objet input HTML de cette façon : `var ladate = document.getElementById('dateinput').value;`

il ne reste plus qu'à intégrer notre objet la date dans l'url tel que ceci :
`var url = "http://pilic27.irisa.fr/API/v2/donnees.php?query=filter&date='"+ladate+"'";`

c. Initialisation des Layers

Ensuite les deux fonctions appellent forcément loadMapLayer qui prend le geojson récupéré par la requête et la map en paramètre . Cette fonction appelle juste map.on('load',function()) qui permet de faire une action au chargement de la map, dans notre cas l'initialisation des Layers.

1 https://www.mapbox.com/
2 https://www.mapbox.com/mapbox-gl-js/api/

map.addSource('ID',{type : geojson, data:VOTRENOMDEVARIABLE}) permet de add une source de nom ID qui a comme contenu la variable mise en paramètre data.
map.addLayer(tableau de paramètre) qui permet de créer un layer avec les paramètres mis dans le tableau. Il faut préciser la source du layers donc ID du addSource ici. Ensuite le type du layers, nous avons utilisé heatmap et circle qui sont adaptés à notre situation. Ensuite chaque type de layers à des spécificité à set, la documentation fournie par mapbox est assez complète la dessus.

d. Création des boutons et gestion des legendes

MapToggleLayers est le fichier qui traite les cliques sur les boutons pour afficher les différents layers en fonctions du choix de l'utilisateur. Il y a deux fonctions, une pour l'initialisation, qui crée les boutons (MapToggleLayer(map)) et une qui ne fait que charger les boutons et gérer leurs events (MapToggleLayerDate(map)). Ces deux fonctions ne comportent que du js.