HAI912I
Dev. mobile avancé, IoT et embarqué

Projet Arduino

Auteur :

Canta Thomas (21607288)
Reiter Maxime (21604458)

Master 2 - Génie Logiciel
Faculté des sciences de Montpellier
Année universitaire 2021/2022

Table des matières

Chapitre 1

Introduction

Le but de ce projet est la mise en place d'un système embarqué Arduino (ESP32/ESP86) permettant de récupérer les parkings de Montpellier disponibles - ouvert et non complet - les plus proches de notre position. Ce projet est réalisé dans le cadre de l'UE HAI912I, Développement mobile avancé, encadré par Mr. Mancheron.

Nous remercions Mr. Mancheron de nous avoir prêté un ESP32 afin de pouvoir mettre en œuvre notre projet.

Chapitre 2

Configuration

Pour faciliter la compréhension et la structuration de notre code nous avons divisé notre programme en différentes classes plutôt que l'utilisation de structure comme nous le faisions auparavant. Celle-ci s'apparente au modèle de conception MVC (Model-View-Controller) Ainsi, notre système se compose en 3 parties :

- main
- controllers/
  - ParkingController
- services/
  - ParkingService
  - DistanceService
  - DistanceLigneDroiteService
  - OSRMService
- models/
  - Parking

Un service se consacre à faire des requêtes HTTP(S) vers une où des API, une fois les données reçues désérialisées elles seront renvoyées. Un contrôleur est comme un pont faisant la liaison entre notre modèle et nos services. Il se charge de récupérer les données désérialisées, récupère les informations nécessaires et les renvoie.

Cette structuration simplifie l'ajout de nouvelles fonctionnalités, mais aussi la refactorisation dans le cadre d'une refonte logiciel ou d'une nouvelle mise à jour par exemple.

Chapitre 3

Réalisation

3.1 Serveur Web

Pour faciliter l'interaction utilisateur, nous avons ajouté un service web. Pour initialiser notre serveur, il est nécessaire de fournir un accès réseau afin de pouvoir communiquer. Pour cela il faudra modifier les valeurs du code ci-dessous.

Nous pouvons maintenant nous connecter au réseau et vérifier son état :

```
#define WIFI_SSID "ssid"
#define WIFI_PWD "password"
...
WiFi.begin (WIFI_SSID, WIFI_PWD);
while (WiFi.status() != WL_CONNECTED) {
  delay(250);
  Serial.print(".");
}
Serial.print("\n[i] Adresse IP: ");
Serial.println(WiFi.localIP());
Serial.println("/!\\ Veuillez patientez que le serveur soit ouvert /!\\");
```

Listing 3.1 – Connection internet

Une fois l'adresse IP affichée dans la console et la confirmation de l'ouverture du serveur, nous pouvons nous y connecter. Pour ce faire, il suffit de copier-coller l'adresse IP dans l'URL de n'importe quel navigateur. Nous traitons plus tard, en détail, l'affichage graphique que nous avons mis en place.

3.2 Accéder aux informations des parkings

L'agglomération de Montpellier fournit un service permettant d'obtenir les informations d'un parking via l'adresse `https://data.montpellier3m.fr/`.

De par ce lien, nous pouvons dorénavant obtenir les informations spécifiques à un unique parking via son identifiant. Pour ce faire, il faudra requêter à l'adresse ci-dessous en remplaçant <ID> par l'identifiant : `https://data.montpellier3m.fr/sites/default/files/ressources/<ID>.xml`

Le service nous répond à l'aide d'une structure XML de la forme suivante :

```
<park>
  <DateTime>...</DateTime>
  <Name>...</Name>
  <Status>...</Status>
  <Free>...</Free>
  <Total>...</Total>
</park>
```

Il faut dorénavant la parser afin d'extraire toute les informations sauf la date et heure que nous n'utiliserons pas. Pour cela, nous avons mis en place le parseur suivant :

```
String getXMLValue(String payload, String xmlKey) {
 if(payload.indexOf("<"+xmlKey+">")>0){
   int CountChar=xmlKey.length();
   int indexStart=payload.indexOf("<"+xmlKey+">");
   int indexStop= payload.indexOf("</"+xmlKey+">");
   return payload.substring(indexStart+CountChar+2, indexStop);
 }
 return "";
}
```

3.3 Affichage graphique

Notre affichage graphique est de la génération de code html directement écrite dans le code à l'aide de la fonction *handleRoot()*, présente dans *main.ino*. Pour offrir un minimum de design nous avons utilisé un framework très connu, bootstrap[2]. Ainsi, lors de l'ouverture de la page web, à l'aide de l'adresse IP, vous obtiendrez la page suivante :

FIG. 3.1 – Page d'accueil

Lorsque l'on ouvre la page, celle-ci nous propose de renseigner la latitude et la longitude et une méthode de calcul de distance (à vol d'oiseau ou trajet réel). De plus, un bouton permet de lancer la recherche. Une amélioration de notre système serait d'obtenir les coordonnées GPS en utilisant la géolocalisation de l'utilisateur afin de lui en éviter la saisie.

Une fois que vous aurez appuyé sur le bouton de soumission, celle-ci affichera un tableau répertoriant dans l'ordre du parking le plus proche au plus éloigné et les informations qui le constituent (nom, nombres de places disponibles et nombres de places totaux). Voici, une partie, d'un affichage possible :

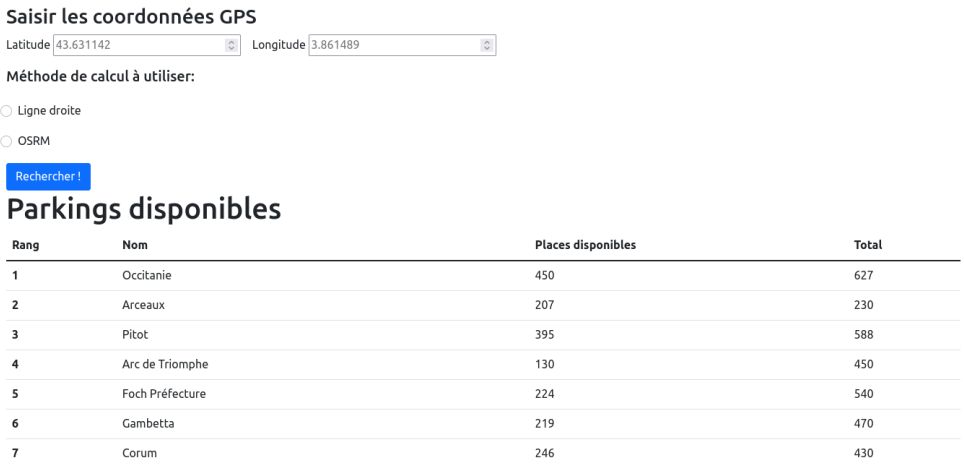

FIG. 3.2 – Exemple d'affichage

3.4 Calcul des distances

Pour calculer les distances entre deux coordonnées il existe une méthode assez simpliste, permettant d'obtenir une approximation sous la forme d'une ligne droite, ou à vol d'oiseau, sans se soucier des obstacles. Pour effectuer le calcul, nous utilisons la formule mathématique suivante :

En supposant un point $A(x_1, y_1)$ et un point $B(x_2, y_2)$

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Une méthode plus complexe serait de pouvoir obtenir la distance en fonction du trajet à parcourir, en se souciant donc des obstacles et des routes existantes. OSRM[3] est une API open-source qui permet de faire cela, en revanche, elle ne regarde pas le temps qu'il faut mettre selon le trafic. De ce fait, il faudras tenir compte qu'un parking peu être plus proche, mais plus lent à atteindre, suite à des bouchons ou autres ralentissements.

Pour cela, nous utiliserons la requête suivante, ou {coordinates} sera remplacé par la paire de coordonnées, source et destination. Source représente le point de départ, et destination celle à atteindre.

```
0 http://router.project-osrm.org/route/v1/driving/{coordinates}
```

Le résultat de cette requête est au format Json. Il est donc nécessaire de mettre en place un parseur tel que nous l'avions fait auparavant avec notre format XML. Afin de pallier à ça, nous utiliserons la bibliothèque ArduinoJson[1].Pour ce faire, nous avons ajouté une bibliothèque appelée ArduinoJson. Celle-ci nous permet de filtrer les valeurs à conserver de toute la requêtes, dans notre cas la distance, et enfin de la désérialiser. Grâce à ce filtrage, nous réduisons considérablement la taille de la donnée reçu.

Pour chaque parking nous calculons la distance par rapport à notre point source, que nous trierons par la suite à l'aide d'une méthode sort prédéfinis en C. Cette méthode nécessite que l'on mette en place une méthode de comparaison que nous utiliserons ci-dessous via une lambda expression afin de passer des paramètres.

```
0 // Compare la distance entre deux parking
1 bool compareParkingsDistances(..) {
2   return parking1->getDistance(..) < parking2->getDistance(..);
3 }

5 // Ordonne les parkings les renvoie
6 const vector<Parking>& ParkingsController::getNearestParkings(..){
7   sort(parkings.begin(), parkings.end(),
8     [longi, lati, &distanceService](Parking& p1, Parking& p2) {
9       return compareParkingsDistances(..);
10  });

12  return parkings;
13 }
```

Chapitre 4

Fiabilité

4.1 Sécurité

L'implémentation sous forme de classes nous permet d'obtenir une structure plus solide et plus consistante, à l'inverse de l'utilisation de structure, notamment grâce aux modificateurs d'accès (private, public, ...). Ceci permet donc de mieux sécuriser notre programme.

Pour l'instant, notre service web n'est pas encore sécurisé, en effet, le site tourne sous un protocole HTTP, il serait plus responsable plus tard de le faire basculer en HTTPS.

4.2 Erreurs

Pour la gestion d'erreurs, le service web renvoie une erreur 404 pour prévenir l'utilisateur et lors des appels de requêtes, on affiche l'erreur dans la console de l'ESP. En cas d'erreur lors d'une requête HTTP, un message est affiché dans la console.

Chapitre 5

Amélioration

5.1 Autonomie

Pour améliorer l'autonomie, il est bon de mettre en œuvre un système permettant de mettre en veille certains composants, lorsqu'aucun calcul n'a pas été fait depuis une certaine unité de temps. Les autres éléments inutilisés doivent être arrêtés, comme les leds, l'écran, etc.

De plus, il faudra choisir un bon compromis pour la mise à jour des informations. En effet, notre projet n'a pas la nécessité d'être en temps réel, mais il faut tout de même recharger nos données assez souvent pour informer l'utilisateur surtout si soudainement un parking manque de place. De ce fait, une mise à jour pourrait être effectuée toutes les 2 minutes, voire toutes les minutes si le parking le plus proche est quasiment vide.

5.2 Optimisation possibles

D'autres optimisations pourraient être ajoutées, notamment la mise en cache des dernières évaluations de distance. Si l'utilisateur a peu, voire pas bougé, il suffira de recharger le nombre de places disponibles uniquement.

De même, nous avons découvert, un peu tard, car impossible à refactoriser dans le temps imparti, une méthode plus simpliste et plus optimale afin de calculer la distance entre une certaine position et tous les coordonnées fournies.

OSRM[3] fournit un système qui permettrait de mapper une certaines coordonnées (celle de l'utilisateur) sur plusieurs coordonnée (celles des parkings) en une requête unique. Voici un exemple de ce que nous venons d'énoncer :

```
0 https://router.project-osrm.org/table/v1/driving/
1   3.861251,43.631004;
2   3.892530,43.607849;
3   3.882257,43.613888?sources=0&annotations=distance
```

De ce fait, ceci aurait pu largement optimiser notre temps d'exécution impliquant une meilleure autonomie de notre système. Ceci aurait pu aussi nous éviter de mettre en place un système de cache afin d'optimiser avant la fin du temps imparti.

5.3 Nouvelles fonctionnalités

Notre projet pourrait être amélioré par l'ajout de nouvelles fonctionnalités. Nous pensons au tri de nos parkings en fonction de la durée de temps de trajet et non de la distance à parcourir. Ceci permettrait à un utilisateur dans le besoin de se garer au plus vite et non au moins loin.

Bibliographie

[1] Benjamin Blanchon. *ArduinoJson*. URL : https://arduinojson.org/.

[2] *Bootstrap*. URL : https://getbootstrap.com/.

[3] *Project OSRM*. URL : http://project-osrm.org/.