Projet Parking-Finder API

Cahier des charges :

L'objectif du projet est de développer une application serveur qui expose un API REST avec Java Spring-boot.

Cette API doit permettre de retourner une liste de parkings à proximité d'un point sur une carte.

La source de données à utiliser est proposée par la ville de Bordeaux. Cette source de données étant accessible via :

http://data.lacub.fr/wfs?key=9Y2RU3FTE8&SERVICE=WFS&VERSION=1.1.0&REQUEST=GetFeature&TYPENAME=ST_PARK_P&SRSNAME=EPSG:4326

De plus, l'API doit pouvoir fonctionner avec d'autres villes. Pour chaque ville, les sources de données pourront être différentes (URL endpoint et format) mais l'API proposée ne doit pas évoluer.

Description :

L'API développée propose les 2 endpoints http suivants :

- **GET /<city>/parking** : Permet de récupérer tous les parkings de la vile « city ».

- **GET /<city>/parking/near?latitude=<latitude>&longitude=<longitude>&distance=<distance> :** Permet de récupérer tous les parkings de la ville « city » et dans une zone définie par un point central (paramètres de requête « latitude » et « longitude ») et avec une distance en mètres autour de ce point (paramètre de requêtes « distance »)

Choix techniques :

L'application Spring-boot propose un unique RestController spring appelé ParkingController. Ce controller expose 2 routes http GET (présentées ci-dessus).

Ce ParkingController va toujours renvoyer la même structure json (définie par ParkingDto), et peu importe la ville.

Le ParkingController appelle un seul service appelé ParkingService qui nous servira de facade pour vérifier que la ville demandée est gérée par l'application et si c'est le cas, dispatchera la demande vers le service de recherche de parkings dédié à la ville demandée.

ParkingService dispose d'une Map java permettant d'aiguiller vers le bon service de recherche de parking pour une ville donnée.

Pour le moment, la seule ville disposant d'un service de recherche de parking est Bordeaux (voir BordeauxParkingService).

Pour ajouter une ville, il faudra simplement créer un nouveau service de recherche de parking implémentant l'interface CityParkingService. Il faudra ensuite déclarer ce nouveau service dans le

service de facade ParkingService (en injectant le bean et en modifiant la map permettant l'association ville –> service).

Pour développer le BordeauxParkingService, j'ai choisi de faire mes requête HTTP en utilisant RestTemplate. L'url d'appel est construite avec les variables « baseUrl » et « key » qui sont définies dans le fichier application.yml, puis les paramètres fixes suivants :

SERVICE=WFS

VERSION=1.1.0

REQUEST=GetFeature

TYPENAME=ST_PARK_P

SRSNAME=EPSG:4326

Enfin, dans le cas où la route /<city>/parking/near est appelée, on ajoute en plus un paramètre filter, dont la valeur est une string XML représentant les critères de filtre :

Par exemple :

```
<Filter>
  <DWithin>
    <PropertyName>the_geom</PropertyName>
      <gml:Point srsName="EPSG:4326">
        <gml:pos>44.827794 -0.689417</gml:pos>
      </gml:Point>
    <Distance units="meter">1500</Distance>
  </DWithin>
</Filter>
```

Cette chaine XML de filtre, est construite en marshalant avec JAXB, un objet Filtre dont la classe a été correctement annoté (voir classes annotées par @XmlRootElement, @XmlAttribute…etc dans le package com/gox/parking/finder/api/jaxb)

JAXB est aussi utilisé afin de mapper la réponse XML du webservice WFS de la ville de Bordeaux, vers un objet Java FeatureCollection.

Le BordeauxParkingService transformera la réponse FeatureCollection, en liste de ParkingDto, entités qui seront à la fin, retournées au client sous forme de json. Pour effectuer cette transformation de bean vers bean, le service s'appuie sur le composant ParkingAssembler.

Axes d'améliorations :

1) Génération des classes OGS/WFS à partir des schémas XSD

Il serait intéressant de générer et mettre à disposition un jar contenant les classes des spécifications OGC et WFS.

On pourrait générer ces classes à partir des XSD disponibles à l'url : http://schemas.opengis.net

J'ai tenté d'utiliser XJC pour générer toutes les classes du protocole WFS mais j'ai eu des soucis et ai perdu beaucoup de temps. J'ai donc créé moi-même les classes avec les annotations JAXB nécessaires aux besoins de ce projet (voir package com/gox/parking/finder/api/jaxb).

Avec ces classes à disposition et JAXB, on pourra alors plus facilement interagir avec les WS OGC de type WFS.

2) WFSWebServiceConsumer

Il serait intéressant de créer un composant spécialisé dans l'appel de webservice WFS. Actuellement les appels http sont faits directement par le seul service de ville disponible à savoir BordeauxParkingService. Mais lorsque l'on souhaitera ajouter de nouvelles villes fournissant leurs données via webservice WFS, il serait intéressant d'avoir un WFSWebServiceConsumer que l'on pourrait facilement réutiliser.

3) Utilisation de MapStruct

Pour ce projet, j'ai développé un assembler permettant de faire des transformations de bean vers bean. Peut-être qu'il pourrait être intéressant d'utiliser une librairie comme MapStruct permettant de faire du mapping de beans, et qui nous permettra de supprimer ou simplifier du code fastidieux et long à écrire.

4) Configuration en base de données

Pour le moment, le projet étant simple et avec une seule ville, la facade ParkingService rempli bien son rôle. En revanche, on pourrait imaginer une solution ou la configuration des services de recherche de parkings des villes, soit stockée en base de données.

Ville	Type d'API	Base URL endpoint	API key
Bordeaux	WFS	http://data.lacub.fr/wfs	1111
Nantes	GeoJSON	http://nantes.geo.api.fr	2222
Lyon	Custom REST API	http://lyon.parking-api.fr	3333

5) Sécurité :

J'ai choisi de laisser la partie sécurité de côté. En revanche, il aurait été facile d'implémenter une Basic Auth par exemple. La marche à suivre aurait été la suivante :

- Ajouter la dépendance maven spring-boot-starter-security
- Créer une classe de config Spring-boot qui hérite de WebSecurityConfigurerAdapter et annotée de Configuraiton, EnableWebSecurity
- Override des méthodes configure du AuthentificationManagerBuilder et HttpSecurity

Pour un projet qui a vocation à monter en production, il faudrait probablement se diriger vers une solution à base d'Oauth2 et de JSON Web token (JWT).

6) Tests

J'ai développé quelques tests unitaires avec Junit et Mockito. Chaque service/component est testé en mockant ses dépendances. En revanche il manque les tests d'intégration pour tester toute la chaine :

Requete http -> ParkingController -> ParkingService -> BordeauxParkingService -> WFS response mockée -> response JSON

7) Cache de la réponse web-service WFS ?

Actuellement pour 1 appel à l'API REST développée, on fait 1 appel au webservice WFS de la ville de Bordeaux. En cas de pic d'utilisation de l'API, on pourrait imaginer que les quotas d'appel du service WFS soient épuisés. Les informations de localisation lat/long, nom, adresse des parkings n'évoluant jamais, on pourrait imaginer mettre ces réponses WFS en cache (Redis pax ex).

Par contre, se pose la question pour les champs « live », TOTAL (nombre de places total) et LIBRES (nombre de places libres), qui eux sont rafraîchis très régulièrement…

8) Ajout d'autres endpoints à l'AP

Pour récupérer un seul parking par id, on pourrait par exemple ajouter le endpoint http GET suivant :/<city>/parking/:id.

Et utiliser lors de l'appel au WS WFS le filtre suivant :

```
<ogc:Filter>
  <ogc:PropertyIsEqualTo matchCase="true">
    <ogc:PropertyName>ident</ogc:PropertyName>
    <ogc:Literal>{ id }</ogc:Literal>
  </ogc:PropertyIsEqualTo>
</ogc:Filter>
```

9) Pour chaque ville, une API parking indépendante ?

Imaginons le cas de figure suivant : l'API gère les parkings pour 3 villes. En intégrant une 4ème ville et malgré les tests, on introduit une régression et on se rend compte que des appels au service de parking de cette 4ème ville peuvent faire complètement crasher l'application. Il y aura alors un fort impact sur les clients de l'API pour les 3 premières villes implémentées, alors que la release concernait une autre ville. Il faudrait envisager d'isoler les API de recherche de parkings de chaque ville sur des applications Spring-boot indépendantes.