

UNIVERSITÉ JEAN MONNET SAINT-ÉTIENNE

Rapport du projet web sémantique

Elias Romdan

Nicolas Trotta

Master 2 Données et Systèmes Connectés

04/01/2021

Table des matières

1	Introduction	3
2	Technologies utilisées	3
2.1	Gestionnaire de version Git	3
2.2	Moteur de templates Thymeleaf	3
2.3	Framework Spring Boot	4
2.4	Triplestore Apache Jean Fuseki	4
3	Organisation des tâches	4
4	Ontologie	5
4.1	Schéma	5
4.2	Ressource hôpital	5
4.3	Ressource station	5
5	Côté application (localhost :8080)	6
5.1	Contrôleur	6
5.2	Parseur	6
5.2.1	Librairie jackson-jsonld	7
5.2.2	Le souci du détail	7
5.2.3	Un champ, plusieurs types	8
5.3	Vue	9
5.3.1	Google Bootstrap	9
5.3.2	Carte interactive	9
5.3.3	Leaflet	9
5.3.4	La navigation dans la page	10
5.3.5	De la carte au contrôleur	12
5.3.6	Marqueurs des hôpitaux et des stations	13
5.3.7	Mécanique de l'interaction de l'utilisateur avec les marqueurs	14
6	Côté triplestore (localhost :3030)	16
6.1	Ajout des données	16
6.2	Extraction des données	16
6.2.1	Récupération des villes	17
6.2.2	Récupération des hôpitaux	17
6.2.3	Récupération des stations	17
6.3	Stockage des données	18
7	Fonctionnalités non implémentées	18
7.1	Insertion de données dans la base de Jena Fuseki	18
7.2	Récupération des informations depuis une autre source de données	19
7.3	Véritable fonctionnement asynchrone de la page web	19
8	Perspectives	20
9	Conclusion	20

Table des figures

1	Organisation des fichiers de Thymeleaf dans un projet Spring Boot	3
2	Attributs et balises	3
3	Ontologie du projet	5
4	Aperçu d'une partie du contrôleur de l'application	6
5	Aperçu de la classe de parsing SparqlHospitalRequestLDModel.java	7
6	Exemple d'un résultat en JSON-LD d'une requête SPARQL	7
7	Cas d'un résultat d'une ville avec plusieurs hôpitaux	8
8	Cas d'un résultat d'une ville avec un seul hôpital	8
9	Les classes de parsing	8
10	Exemple des champs à risque de plusieurs valeurs	8
11	Extrait du tableau des informations sur les hôpitaux	9
12	Extrait du fichier bus.html relatif à l'intégration de la carte Leaflet	10
13	La page web bus.html	10
14	Cas où l'utilisateur choisit la ville de Saint-Étienne	11
15	Cas où l'utilisateur clique sur le marqueur du centre universitaire hospitalier de Saint-Étienne	11
16	Cas où l'utilisateur clique sur le marqueur d'un arrêt à proximité	11
17	Extrait du code source du contrôleur relatif au traitement et à l'envoi des données des villes à la vue	12
18	Code HTML/Thymeleaf du formulaire pour afficher les hôpitaux d'une ville donnée	12
19	Envoi des données des hôpitaux du contrôleur à la vue	12
20	Code HTML/Thymeleaf du tableau des hôpitaux sous la carte	13
21	Vérification de l'existence des coordonnées GPS des hôpitaux	13
22	Méthode de traitement des requêtes relatifs aux arrêts dans le contrôleur	14
23	Envoi des coordonnées du JavaScript au contrôleur lors du clique sur le marqueur d'un hôpital . . .	15
24	Récéption des coordonnées dans le back	15
25	Construction des modèles, définition des paramètres de la carte et envoi des données au front	15
26	Aperçu de l'ensemble de données hospitals	18
27	Fichier CSV des hôpitaux	19

2.3 Framework Spring Boot

Nous avons fait le choix de construire l'application à partir du Framework Java [Spring Boot](#) de manière à pouvoir développer celle-ci autour d'une architecture de type MVC. L'utilisation conjointe de l'IDE [Visual Studio Code](#) nous a permis de rapidement de disposer des librairies via le gestionnaire de dépendances [Apache Maven](#) et de mettre en place une application web minimale à partir de laquelle nous avons travaillé.

2.4 Triplestore Apache Jean Fuseki

Nous avons décidé de choisir [Apache Jean Fuseki](#) de parmi les triplestores sur le marché, vu sa disponibilité en open source, sa rapidité et facilité à s'installer et s'exécuter et vu l'expérience que nous avons acquis par son utilisation dans les séances de TP.

3 Organisation des tâches

À titre indicatif, le temps de travail a été partagé équitablement, dans le développement quotidien comme dans les périodes de rush. Aucun de nous deux n'a participé moins que l'autre au projet. Bien que mon binôme et moi-même ayons fait le choix du nous assigner à chacun la réalisation de parties spécifiques du travail, la répartition effective des tâches dans le projet de web sémantique n'a pas été binaire. La planification initiale a bien entendu évoluer au fil de l'avancée du projet et du temps que nous nous pouvions lui consacrer, au regard de nos autres obligations scolaires, non moins nombreuses et massives.

Au fur et à mesure que le projet avançait des choix ont dû être fait : Des fonctionnalités ont dû être abandonné et d'autres incluses, des sources de données ont dû être préférées à d'autres, une meilleure compréhension des objectifs attendus du projet ont parfois réorienté la direction de celui-ci. Ainsi à chaque changement amorcé dans le projet, le planning et la répartition des tâches a été revue en conséquence. L'enjeu principal étant, on le rappelle, d'avancer le plus vite possible sans se marcher sur les pieds.

Elias s'est principalement chargé de :

- Création du projet initial et du dépôt GitHub.
- Mise en place de la liaison avec Apache Jena Fuseki.
- Extraction des sources d'informations sur Wikidata.
- Écriture des requêtes SPARQL de type :
 - SELECT pour récupérer les informations depuis Wikidata et les stocker sur Apache Jena Fuseki.
 - CONSTRUCT pour créer un graphe RDF contenant le résultat de la requête et le passer en JSON-LD.
- Écritures des classes et des méthodes en Java afin de :
 - Envoyer les requêtes sur le serveur Apache Jena Fuseki.
 - Traiter le résultat des requêtes.
 - S'assurer que les méthodes renvoient la bonne réponse et le bon "input" au contrôleur.

Nicolas a majoritairement eu pour tâches de :

- Mettre en place le contrôleur de l'application
- Assurer les échanges entre Apache Jena Fuseki et le contrôleur ainsi que le contrôleur et la vue.
- Créer la vue de l'application avec HTML et la librairie Thymeleaf ainsi que la mise en place des formulaires et de la carte des hôpitaux.
- Créer les différentes classes du modèle de l'architecture MVC avec Spring Boot :
 - Les classes nécessaires au dialogue entre le contrôleur et la vue (le formulaire, la carte ou les autres données).
 - Les classes utiles au fonctionnement du contrôleur (gestion à proprement parler des informations de la vue).
 - Les classes nécessaires au parsing des objets en JSON-LD transmises par la partie de l'application chargée du dialogue avec Wikidata et Apache Jena fuseki.

Bien que nous ayons eu chacun la charge de mener à bien les tâches citées plus haut, nous avons pour le bien commun parfois allègrement braconné sur les terres de l'autre. En fonction des besoins du moment et de l'avancée de chacun, nous avons régulièrement assisté l'autre et/ou travaillé de concert. Le choix des données à extraire (CSV des hôpitaux ou ressource Wikidata), de la manière de les parser (choix des librairies), des requêtes à effectuer (SPARQL) ont par exemple pu faire l'objet d'un travail collectif.

Ce qui reste du projet final ne témoigne en rien de la direction qu'il a pu prendre ou de ses repentirs. Les nombreux tests effectués, disparus depuis, sur le parsing des données, les échanges avec le contrôleur de l'application, ou bien sur les informations à extraire des sources, ont été réalisés par nous deux, afin de faire avancer au mieux le projet, mais également dans le souci de ne pas être ignare sur l'une des parties du développement.

4 Ontologie

4.1 Schéma

Ci-dessous le schéma de l'ontologie défini pour ce projet. Le mot externe signifie que la ressource est connectée à d'autres ontologies qui ne seront pas traitées par ce projet.

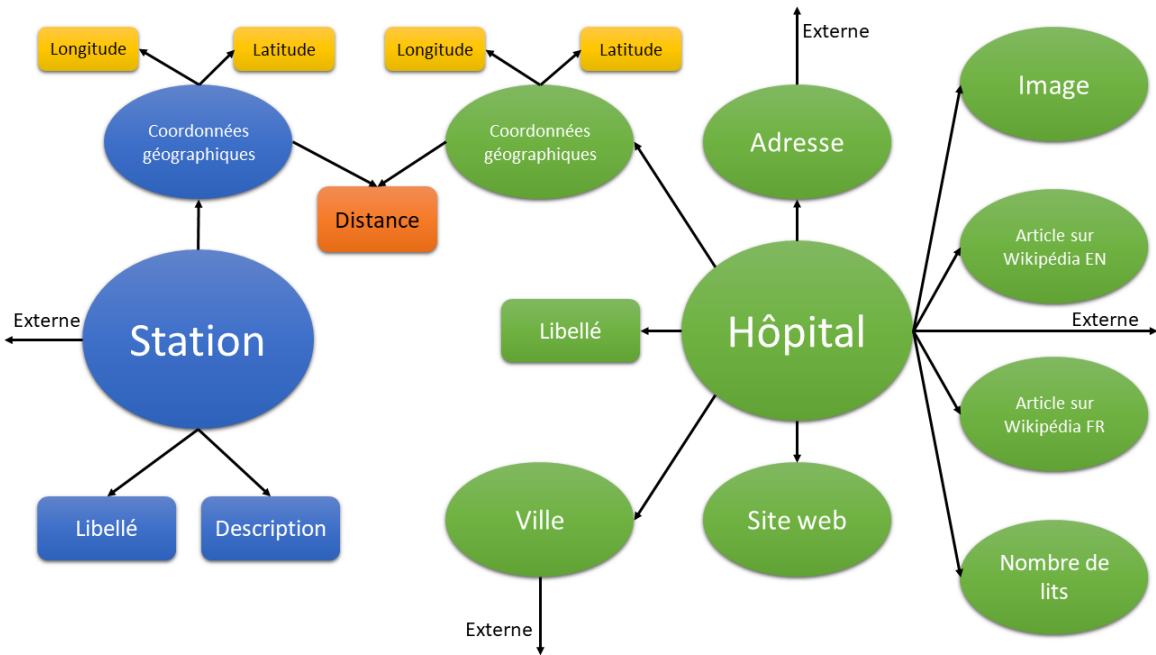


FIGURE 3 – Ontologie du projet

4.2 Ressource hôpital

Cette ressource est stockée dans l'ensemble de données hospitals sur Apache Jena Fuseki selon l'ontologie décrite ci-dessous :

Préfixe	Lien	Terminaison	Valeur
rdf	w3.org/1999/02/22-rdf-syntax-ns#	type	Hôpital (Q16917)
db	dbpedia.org/ontology/	name	Libellé (FR)
		city	Localisation administrative (FR)
		picture	Lien vers une image de l'hôpital
		bedCount	Nombre de lits d'hôpital
		address	Adresse
		Website	Site officiel
ns	w3.org/2006/vcard/ns#	longitude	Coordonnées géographiques
		latitude	Coordonnées géographiques
mo	purl.org/ontology/mo/	wikipedia	Lien Wikipedia EN de l'hôpital
gn	geonames.org/ontology/documentation.html#	wikipediaArticle	Lien Wikipedia FR de l'hôpital

4.3 Ressource station

Cette ressource est récupérée en temps réel à partir de la base Wikidata puis transmise à travers un graphe selon l'ontologie suivante :

Préfixe	Lien	Terminaison	Valeur
wikibase	wikiba.se/ontology#	label	Libellé (FR) sinon (EN)
		description	Description
		geoLongitude	Coordonnées géographiques
		geoLatitude	Coordonnées géographiques

5 Côté application (localhost :8080)

5.1 Contrôleur

Il s'agit du point névralgique du projet, le lieu de transition entre une demande de requête, son traitement et la redirection du résultat vers l'utilisateur final. Nous transmettons les données à la vue via une série de ressources situées dans le contrôleur. La transmission des données utiles du contrôleur vers la vue se fera grâce à l'objet modèle, que nous remplirons afin qu'il soit exploité dans la page HTML.

```

/*****
/* Rest Resource */
*****/

/* Index Rest Resource */
@RequestMapping("/")
public String index(Model model, Reponse reponse) {
    model.addAttribute("reponse", reponse);
    Record.load();
    return "redirect:/bus";
}

/* Data addition on Jena Fuseki Rest Resource */
@RequestMapping("/index")
public String add(Model model, Reponse reponse) {

    return "index";
}

/* Hospital Map Rest Resource */
@RequestMapping("/map")
public String map(Model model, ReponseVille reponseVille)
    throws JsonParseException, JsonMappingException, IOException {

    model = buildCitiesWithHospitalModel(model, reponseVille);

    ArrayList<HospitalLD> hospitals = new ArrayList<>();
    model = buildHospitalsByCityModel(model, reponseVille, hospitals);

    model = buildHospitalsCoordinatesModel(model, hospitals);

    return "map";
}

```

FIGURE 4 – Aperçu d'une partie du contrôleur de l'application

5.2 Parseur

Le tout n'est pas d'obtenir les bonnes données puis de les traiter, encore faut-il les transmettre dans un format lisible et exploitable. Si à la base nous faisons une requête SPARQL dont nous insérons le résultat sur le triplestore Apache Jena Fuseki, le résultat d'une requête d'un utilisateur depuis la vue, une fois traité sera transmis au contrôleur au format JSON-LD. Toute la récupération des données sur Wikidata et la création de cet objet JSON-LD sera traitée dans le chapitre suivant, je me focaliserai principalement dans ce chapitre sur le parsing de l'objet au format JSON-LD traité au niveau du serveur puis renvoyé au contrôleur afin d'être transmis à la vue.

5.2.1 Librairie jackson-jsonld

Afin de parser l'objet JSON-LD envoyé au contrôleur, nous avons fait le choix d'employer l'une des bibliothèques du parseur [Jackson](#) à savoir [jackson-jsonld](#).

5.2.2 Le souci du détail

Afin de pouvoir désérialiser l'objet JSON-LD, il sera nécessaire de créer une classe Java qui en reprendra la structure. Celle-ci contiendra toute la structure de chacun des champs de l'objet que l'on pourra paramétrer à l'aide d'annotations de type sur la classe en elle-même ou bien sur chaque variable selon le besoin. La particularité de cette classe dédiée, qui frappera tous ceux ayant le privilège de devoir l'écrire, sera la longueur et la relative complexité de sa structure. Même pour un objet JSON-LD extrêmement simple en apparence, il faudra lui associer une classe particulièrement verbeuse et compliquée, nous prendrons l'exemple de la classe `SparqlHospitalRequestLDModel.java` située dans le dossier `model` du code source de l'application.

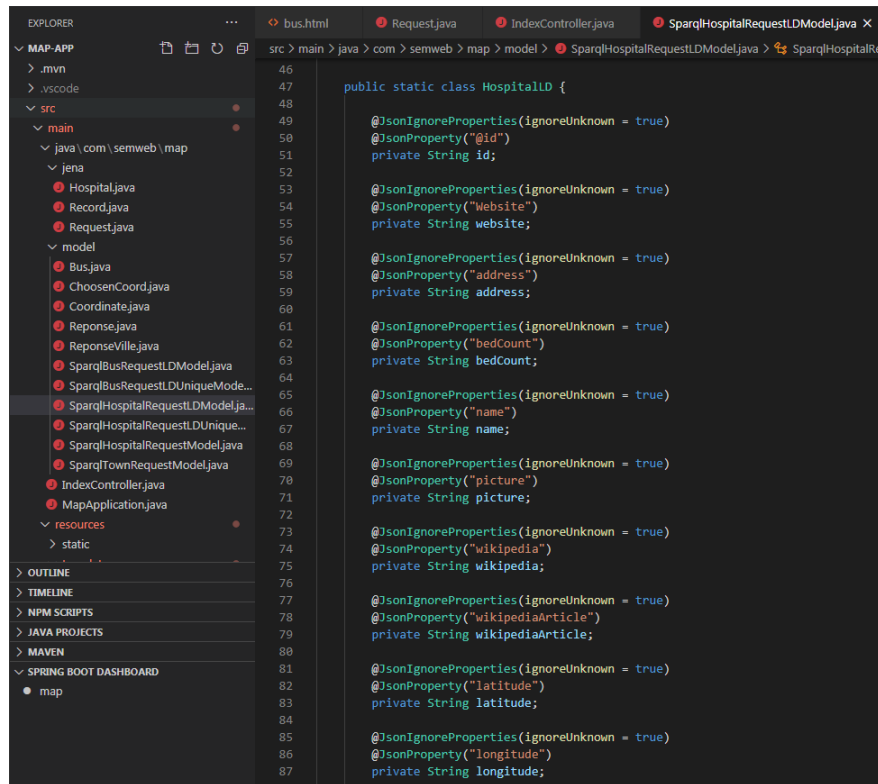


FIGURE 5 – Aperçu de la classe de parsing `SparqlHospitalRequestLDModel.java`

Afin de pouvoir exploiter les renseignements de 10 champs à peine, il nous faudra créer variables, constructeurs, getteurs et setteurs de chacune des classes, parents et enfants. On se retrouvera au final avec un fichier de 700 lignes, propice aux erreurs lors de sa rédaction en raison des sous-classes imbriquées.



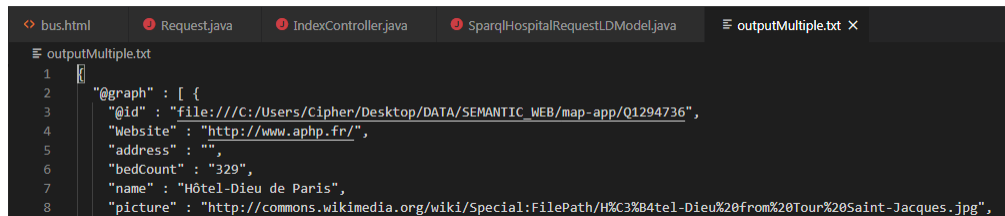
FIGURE 6 – Exemple d'un résultat en JSON-LD d'une requête SPARQL

5.2.3 Un champ, plusieurs types

L'une des difficultés rencontrées lors du parsing des résultats d'une requête tiendra au fait que le résultat d'une requête SPARQL pourra engendrer une réponse ou plusieurs pour chacun de ses champs. Cela se manifestera dans le projet dans 2 cas précis :

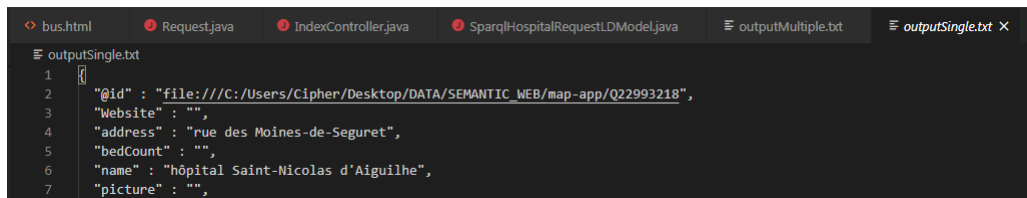
- Lorsque l'utilisateur choisira une ville. Une ville pourra disposer d'un hôpital ou de plusieurs.
- Lorsque l'utilisateur cliquera sur un hôpital pour obtenir les bus aux alentours des 5 km. Un hôpital peut avoir à proximité un arrêt, plusieurs ou alors aucun.

Dans le cas de la requête pour l'affichage des hôpitaux d'une ville donnée, nous pourrions avoir 2 types de fichiers de sortie :



```
1 [{"@graph": [{"@id": "file:///C:/Users/Cipher/Desktop/DATA/SEMANTIC_WEB/map-app/Q1294736",
2   "Website": "http://www.aphp.fr/",
3   "address": "",
4   "bedCount": "329",
5   "name": "Hôtel-Dieu de Paris",
6   "picture": "http://commons.wikimedia.org/wiki/Special:FilePath/H%C3%B4tel-Dieu%20from%20Tour%20Saint-Jacques.jpg",
7 }]}]
```

FIGURE 7 – Cas d'un résultat d'une ville avec plusieurs hôpitaux



```
1 [{"@id": "file:///C:/Users/Cipher/Desktop/DATA/SEMANTIC_WEB/map-app/Q22993218",
2   "Website": "",
3   "address": "rue des Moines-de-Seguret",
4   "bedCount": "",
5   "name": "hôpital Saint-Nicolas d'Aiguilhe",
6   "picture": ""}]
```

FIGURE 8 – Cas d'un résultat d'une ville avec un seul hôpital

Pour chacun des cas possibles, il faudra créer une classe dédiée à la désérialisation du type de requête, toute aussi copieuse, puis employer la bonne classe en fonction du nombre de résultats obtenus. Dans le projet, les requêtes des hôpitaux et des bus nous ont contraint à créer 2 types de classes : unique et multiple.

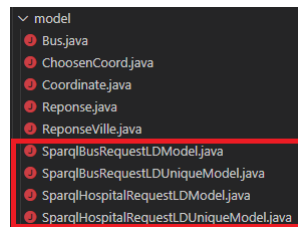
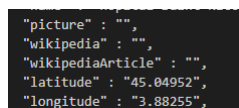


FIGURE 9 – Les classes de parsing

Il va sans dire que chaque champ de la classe principale ou de l'une de ses sous-classes pourra être unique ou bien être un tableau. Ce scénario s'est par exemple produit pour les champs picture, latitude ou longitude car un hôpital pouvait disposer dans Wikidata de plusieurs images mais aussi selon les cas de plusieurs coordonnées GPS, aussi nous avons dû modifier les requêtes afin de limiter les résultats à un seul pour les champs problématiques.



```
"picture": "",
"wikipedia": "",
"wikipediaArticle": "",
"latitude": "45.04952",
"longitude": "3.88255",
```

FIGURE 10 – Exemple des champs à risque de plusieurs valeurs

5.3 Vue

5.3.1 Google Bootstrap

Le but de ce projet est de développer une application bâtie sur la mise en oeuvre des principes du web sémantique, pas de présenter un parangon de beauté et de design, mais un minimum s'impose. Étant donné le peu de temps dont nous disposons, il n'est pas possible de réinventer la roue et de ciseler chaque détail d'une page HTML à grands renforts de CSS et de de JavaScript.

Heureusement, [Google Bootstrap](#) nous a permis de garnir, à relativement peu de frais, notre application d'une plastique agréable. L'autre avantage de ce framework HTML, CSS et JavaScript est qu'il s'intègre à merveille avec Thymeleaf.

5.3.2 Carte interactive

Tout comme on dit qu'une image vaut 1000 mots, intégrer une carte pour afficher des lieux sera préférable à n'importe quelle liste de coordonnées ou tableau de références. La meilleure preuve de ce constat est que même pour ceux qui ont développé l'application, le tableau des hôpitaux qui figure sous la carte de la page devient quasi transparente, peu importe les renseignements qu'il contient.

Pour l'utilisateur final, l'application offre 2 fonctionnalités principale :

- Afficher l'ensemble des hôpitaux installés sur la ville sélectionnée, indiqués par un marqueur bleu.
- Afficher les arrêts de transport de l'hôpital sur lequel l'utilisateur a cliqué, indiqués par un marqueur vert.

Les coordonnées GPS sont récupérées sur wikidata. Dans le tableau présent sous la carte sont affichées diverses informations complémentaires (si présentes sur Wikidata) :

- Nom de l'établissement
- Nombre de lits
- Lien hypertexte vers :
 - La page française et/ou anglaise de l'établissement sur [Wikipédia](#).
 - Une image de l'établissement.
 - Le site institutionnel de l'établissement.

Etablissement	NbLits	Wiki Fr	Wiki Eng	Photo	Site
centre hospitalier universitaire de Saint-Étienne		Wiki-Fr		Image	Site
hôpital de la Charité		Wiki-Fr		Image	

FIGURE 11 – Extrait du tableau des informations sur les hôpitaux

5.3.3 Leaflet

Plusieurs solutions existent pour intégrer une carte sur la page, 3 ont retenu notre attention. La première d'entre elles est [Google Maps](#), mais ayant changé sa politique commerciale depuis 2018, même à des fins de tests, il faudra, afin d'obtenir une clé d'API, créer un compte Google et un compte de facturation.

La seconde et la troisième, étaient gratuites et ne nécessitaient aucune inscription. Entre [Open Street Map](#) et [Leaflet](#), le besoin d'intégrer des marqueurs et de les personnaliser nous a convaincu d'adopter Leaflet.

L'avantage non négligeable de la librairie Leaflet réside aussi dans la façon dont elle se présente à ses utilisateurs, notamment aux nouveaux venus. Les tutoriaux présents sur le site sont didactiques et très bien réalisés, expliquant chaque possibilité de la librairie pas à pas.

Afin d'intégrer une carte à sa page web, il suffira concrètement d'importer la librairie CSS et JavaScript, de créer tout l'environnement objet sous JavaScript, puis de l'invoquer dans le contenu du HTML.

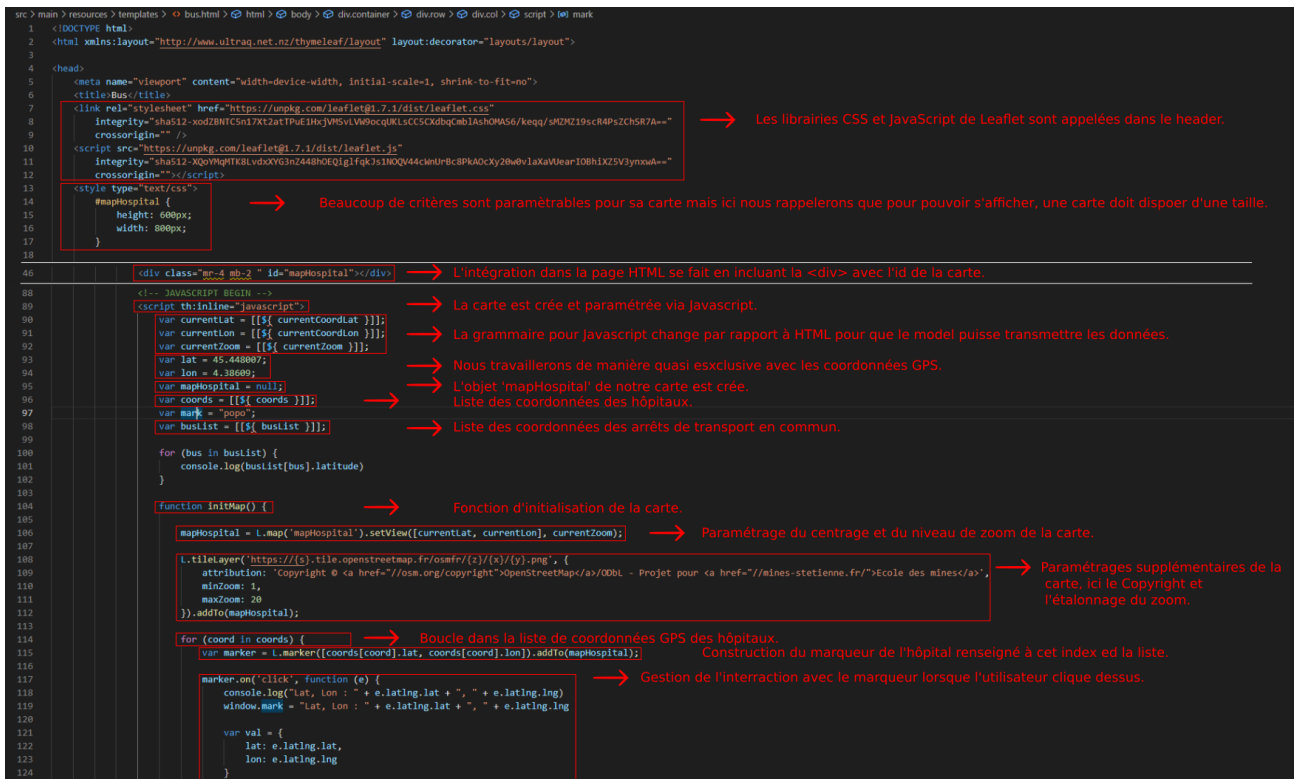


FIGURE 12 – Extrait du fichier bus.html relatif à l'intégration de la carte Leaflet

5.3.4 La navigation dans la page

La page HTML bus.html se trouve être la page principale de l'application, depuis laquelle l'utilisateur pourra sélectionner une ville afin que s'affiche les hôpitaux présents sur son territoire. Une fois les hopitaux affichés, l'utilisateur aura la possibilité d'afficher sur la carte les arrêts de transport en commun accessibles dans un rayon de 5 km autour de l'hôpital choisi. Il n'y aura au final qu'un formulaire interactif, le reste de la navigation se fera par l'intermédiaire des marqueurs affichés sur la carte. Par défaut, et comme il sera détaillé plus tard, la carte est centrée avec un niveau de zoom qui englobera l'ensemble du pays.

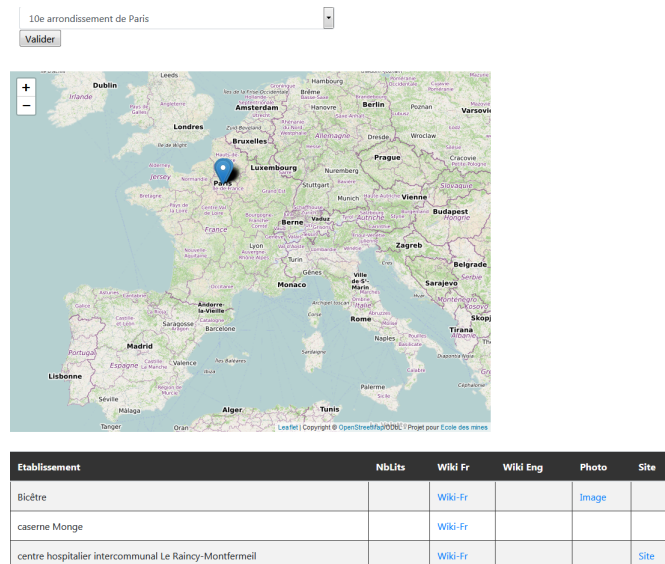


FIGURE 13 – La page web bus.html

Choisir une ville affichera les marqueurs bleux des hôpitaux sans changer le niveau de zoom.

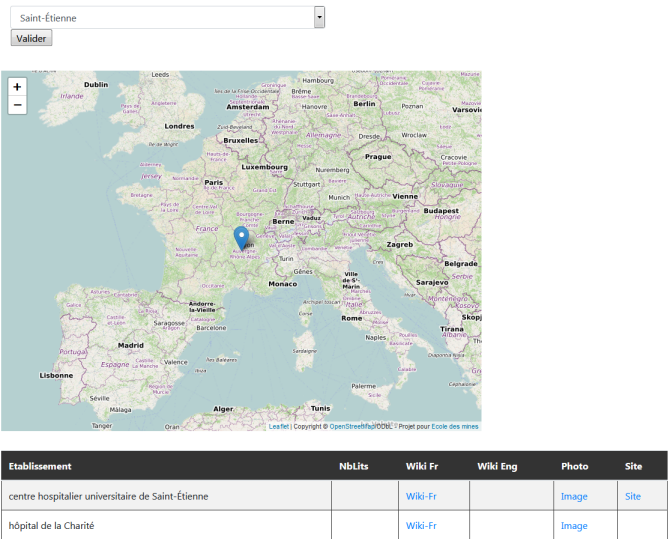


FIGURE 14 – Cas où l'utilisateur choisit la ville de Saint-Étienne

En cliquant sur le marqueur d'un hôpital, la carte zoomera jusqu'à l'échelle d'une ville et si autour de l'hôpital se trouvent des arrêts (bus, trains, ...) alors les marqueurs verts pointant ceux-ci apparaîtront.

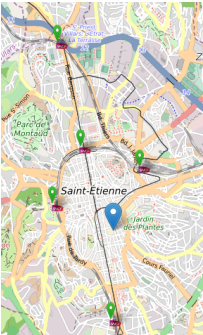


FIGURE 15 – Cas où l'utilisateur clique sur le marqueur du centre universitaire hospitalier de Saint-Étienne

En cliquant sur le marqueur d'un arrêt de bus vert, une bulle d'info apparaîtra pour donner le nom de l'arrêt si celui-ci est stocké dans la base de Wikidata.

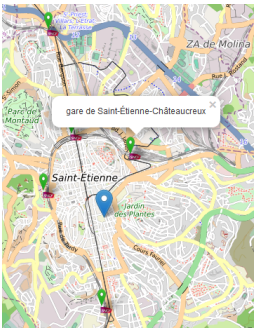


FIGURE 16 – Cas où l'utilisateur clique sur le marqueur d'un arrêt à proximité

Si l'utilisateur change de ville ou rafraîchit la page, la carte retrouvera le niveau à l'échelle du pays.

5.3.5 De la carte au contrôleur

Une fois l'application initialisée, avec les données de base récupérées sur Wikidata et insérées dans Jena Fuseki, celle-ci dispose des informations relatives aux villes et aux hôpitaux.

— Page par défaut :

Du côté contrôleur, dès que la page bus.html est chargée (dans le cas où l'utilisateur aurait cliqué sur le lien de la page ou dans le cas d'un refresh de la page), la carte est centrée sur la ville de Saint-Étienne et affiche par défaut les hôpitaux de la ville de Paris.

```
164 private Model buildHospitalsByCityModel(Model model, ReponseVille reponseVille, ArrayList<HospitalID> hospitals)
165     throws JSONException, JsonProcessingException {
166
167     /* Get hospitals by city */
168     String city = "Paris";
169
170     if (reponseVille.getName() != null) {
171         city = reponseVille.getName();
172     }
173
174     Map<String, String> hospitalsList = Request.getHospitalsByCity(city);
175
176     /* Get list with 1 hospital */
177     if ((long.valueOf(hospitalsList.get("size")) == 1) {
178         SparqlHospitalRequestUniqueModel requestHospitalUnique = objectMapper
179             .readValue(hospitalsList.get("content"), SparqlHospitalRequestUniqueModel.class);
180         HospitalID uniqueHospital = fillHospitalUnique(requestHospitalUnique);
181         hospitals.add(uniqueHospital);
182     }
183     /* Get list with more than 1 hospital */
184     else {
185         SparqlHospitalRequestMultipleModel requestHospitalMultiple = objectMapper.readValue(hospitalsList.get("content"),
186             SparqlHospitalRequestMultipleModel.class);
187         for (HospitalID hos : requestHospitalMultiple.getGraph()) {
188             hospitals.add(hos);
189         }
190     }
191
192     hospitals.sort(comparator.comparing(HospitalID::getName, String.CASE_INSENSITIVE_ORDER));
193     model.addAttribute("hospitals", hospitals);
194
195     return model;
196 }
197
198
```

→ La ville de Paris est sélectionnée par défaut.

→ Si l'utilisateur a choisi une autre ville, alors elle remplace la ville par défaut.

→ On récupère grâce à une requête sur Jena Fuseki la taille de la réponse et les hôpitaux d'une ville dans un objet de format jsonLD.

→ 2 scénarios sont alors possibles :
- Il peut n'y avoir qu'un seul hôpital.
- Il peut y en avoir au moins 2.
Il faudra alors utiliser la classe et la méthode de parsing appropriée.

→ Dans les 2 cas, une ArrayList<HospitalID> est remplie d'objets de structure HospitalID, qui contient l'ensemble des renseignements pour chaque hôpital récupéré.

→ L'ArrayList est triée par rapport au nom de l'hôpital.
(Ce tri sera utile pour l'affichage dans le tableau sous la carte)
Cette liste sera enfin intégrée au **model**, lien entre le Controller et la Vue.

→ Le model, une fois rempli, sera transmis à la Vue.

FIGURE 17 – Extrait du code source du contrôleur relatif au traitement et à l'envoi des données des villes à la vue

Du côté vue, un simple formulaire de type Select affiche les villes françaises qui disposent d'au moins un hôpital répertorié sur Wikidata. On peut noter que le nécessaire a été fait pour que la dernière ville sélectionnée par l'utilisateur soit désormais celle par défaut.

```
35 <form class="form-group col-md-6" action="#" th:action="@{/bus}" th:object="${reponseVille}"
36     method="post">
37     <select name="name" class="form-control">
38         <option th:each="city : ${cities}" th:value="${city}" th:selected="${reponseVille.name eq city}"
39             th:text="${city}">
40     </select>
41
42     <input type="submit" value="Valider">
43 </form>
```

→ Le résultat est renvoyé au Contrôleur via l'objet reponseVille sera intégré au model. ReponseVille sert de lien entre le Contrôleur et la Vue via le model.

→ Grâce à Thymeleaf (th:each), il sera possible d'itérer à travers de la liste.

→ L'utilisateur choisit la ville via le formulaire et valide pour que le résultat soit traité par le Contrôleur.

FIGURE 18 – Code HTML/Thymeleaf du formulaire pour afficher les hôpitaux d'une ville donnée

— Récupération des hopitaux d'une ville :

Une fois que l'utilisateur a fait le choix d'une ville, alors le nom de celle-ci est envoyée au contrôleur afin que sa demande soit traitée.

```
140 private Model buildCitiesWithHospitalModel(Model model, ReponseVille reponseVille)
141     throws JSONException, JsonProcessingException {
142
143     /* Get cities with hospitals */
144     SparqlTownRequestModel sparqlTownRequestModel = objectMapper.readValue(request.getCities(),
145         SparqlTownRequestModel.class);
146     ArrayList<String> townList = new ArrayList<>();
147
148     for (String city : sparqlTownRequestModel.getCity()) {
149         townList.add(city);
150     }
151
152     townList.sort(String.CASE_INSENSITIVE_ORDER);
153     model.addAttribute("cities", townList);
154     model.addAttribute("reponseVille", reponseVille);
155
156     return model;
157 }
158
159
```

→ Parsing du résultat de la requête sur Jena Fuseki.

→ Création de l'ArrayList qui contiendra le nom de chaque ville

→ Remplissage de l'ArrayList avec le résultat de la requête.

→ Tri de la liste des villes pour le formulaire de sélection.

→ Attribution de la liste des villes dans un model transmis à la vue.

→ Attribution dans un autre model de la ville choisie par l'utilisateur.

→ Envoi du model à la vue.

FIGURE 19 – Envoi des données des hôpitaux du contrôleur à la vue

Le tableau HTML sous la carte affiche les informations de chaque hôpital pointé par un marqueur.

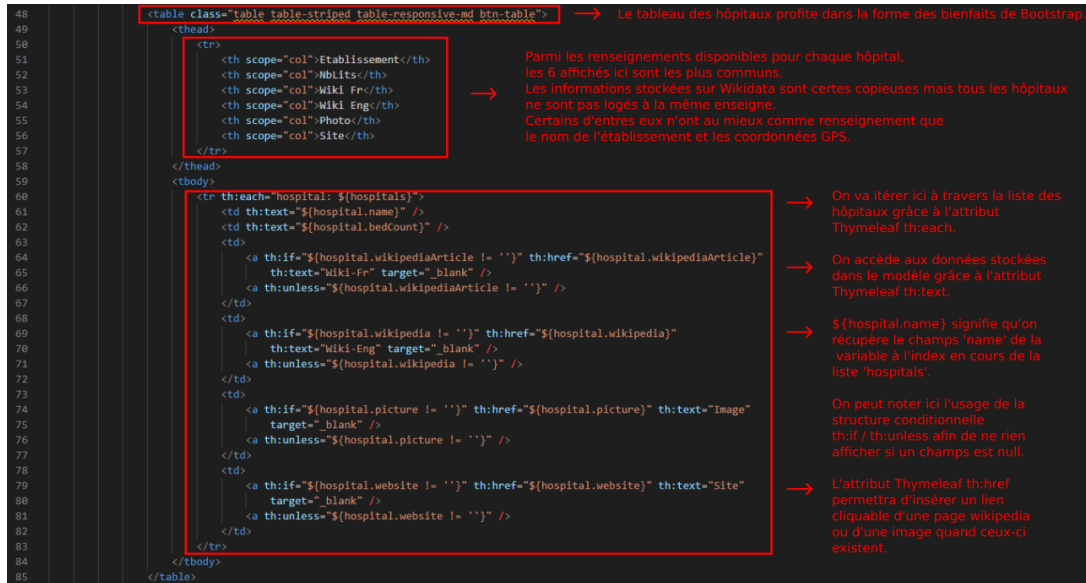


FIGURE 20 – Code HTML/Thymeleaf du tableau des hôpitaux sous la carte

5.3.6 Marqueurs des hôpitaux et des stations

Comme il a été dit plus haut, l'intégration des marqueurs à la carte affichée sur la page est l'une des fonctionnalités majeures de l'application. Les marqueurs ont le bon goût de pouvoir rendre accessoire l'ajout d'un tableau récapitulatif si ce n'est pas absolument nécessaire. Le tableau des hôpitaux pour chaque ville permet de disposer leurs informations majeures (ou mineures) mais dans le cas des arrêts de bus (de transports en commun pour être exact), un tableau de plus ne servirait à rien, les marqueurs se suffisant à eux-mêmes. De plus il suffit de cliquer sur le marqueur d'un arrêt de bus pour que son nom s'affiche à côté.

— Récupération des hôpitaux d'une ville :

Le dialogue avec le contrôleur relatif à l'obtention des informations sur les hôpitaux d'une ville donnée, a été détaillé dans le chapitre précédent. Celui-ci s'assure d'alimenter le tableau de la page bus.html situé sous la carte Open Street Map, seulement toutes les données ne sont pas réservées à l'usage du tableau. La liste d'hôpitaux, véhiculée depuis le contrôleur grâce au modèle, contient aussi les coordonnées GPS de chaque hôpital. S'il a peu été abordé jusqu'à présent, le JavaScript tient une place prépondérante dans la partie cue du projet. Parmi ses atouts, Thymeleaf peut se targuer de transmettre relativement facilement les données au JavaScript avec une grammaire qui change peut par rapport à l'intégration des données envoyées depuis le contrôleur dans le HTML. Nous n'allons pas répéter ce qui a été dit au chapitre consacré à Leaflet, par rapport à l'intégration de la carte et des marqueurs, ni ce qui a été détaillé au chapitre qui précède, par rapport au traitement des informations du contrôleur afin de récupérer les informations des hôpitaux car les coordonnées GPS, utilisés pour afficher les marqueurs, sont compris dans le reste des données. Nous signalerons juste qu'une méthode du contrôleur s'assurera que seuls les hôpitaux disposant de coordonnées GPS soit envoyés à la vue par le modèle.

```

242 private Model buildHospitalsCoordinatesModel(Model model, ArrayList<HospitalID> hospitals) {
243     /* Add hospitals coordinate */
244     ArrayList<Coordinate> coordList = new ArrayList<>();
245     for (HospitalID hos : hospitals) {
246         if (hos.getLatitude().isEmpty() || hos.getLongitude().isEmpty()) {
247             coordList.add(
248                 new Coordinate(Double.parseDouble(hos.getLatitude()), Double.parseDouble(hos.getLongitude()));
249             }
250         }
251     }
252     model.addAttribute("coords", coordList);
253     return model;
254 }
255

```

FIGURE 21 – Vérification de l'existence des coordonnées GPS des hôpitaux

— Récupération des arrêts des transports en commun à proximité d'un hôpital :

Nous verrons plus loin comment fonctionne le système de clique interactif sur un marqueur de la carte. Nous allons brièvement rappeler que le traitement depuis le contrôleur de la récupération des coordonnées GPS des arrêts est très proche de celui des hôpitaux.

```
203 private Model buildNearbyStopsModel(Model model, ReponseVille reponseVille)
204     throws JsonMappingException, JsonProcessingException {
205
206     Map<String, String> nearbyList = Request.getNearbyStations(Double.valueOf(choosenLon),
207         Double.valueOf(choosenLat));
208
209     ArrayList<Bus> busList = new ArrayList<>();
210
211     if (Long.valueOf(nearbyList.get("size")) != 0) {
212         /* Get list with 1 stop */
213         if (Long.valueOf(nearbyList.get("size")) == 1) {
214             SparqlBusRequestLDUniqueModel requestBusUnique = objectMapper.readValue(nearbyList.get("content"),
215                 SparqlBusRequestLDUniqueModel.class);
216             Bus bustmp = fillBusUnique(requestBusUnique);
217             busList.add(bustmp);
218         }
219         /* Get list with more than 1 stop */
220         else {
221             SparqlBusRequestLDModel requestBusMultiple = objectMapper.readValue(nearbyList.get("content"),
222                 SparqlBusRequestLDModel.class);
223             for (BusLD bus : requestBusMultiple.getGraph()) {
224                 Bus bustmp = fillBusMulti(bus);
225                 busList.add(bustmp);
226             }
227         }
228     }
229
230     busList.forEach(re -> System.out.println());
231
232     model.addAttribute("busList", busList);
233
234     return model;
235
236 }
```

FIGURE 22 – Méthode de traitement des requêtes relatifs aux arrêts dans le contrôleur

5.3.7 Mécanique de l'interaction de l'utilisateur avec les marqueurs

Si envoyer des informations du contrôleur à la vue est relativement aisé dans ce projet grâce à Thymeleaf, il en va tout autrement dans le sens contraire. Envoyer des données du JavaScript au contrôleur s'avère largement plus problématique car cette fonctionnalité n'est pas directement prise en charge par une des bibliothèques de Spring Boot ou Thymeleaf.

Afin de transmettre un objet depuis le JavaScript jusqu'au contrôleur, il sera nécessaire de procéder à une requête asynchrone de type GET ou POST selon le cas et la technologie employée. Dans notre cas, il s'agira de envoyer les coordonnées GPS de l'hôpital, sur lequel a cliqué l'utilisateur, de la vue au contrôleur afin qu'une requête SPARQL interroge Wikidata, que la réponse au format JSON-LD soit parsée puis traitée par le contrôleur et enfin que la liste des arrêts de bus, trams, etc. à proximité d'un hôpital soit renvoyée à la vue via le modèle afin que ces arrêts soit affichés par des marqueurs sur la carte.

Les coordonnées du marqueur iront du front vers le back où la requête sera traitée puis les nouvelles données seront renvoyées au front. Il faut noter que même s'il s'agit d'une requête Ajax asynchrone, le contrôleur traitera la demande et la page HTML sera rechargée afin que les changements soient pris en compte sur la carte.

Du côté HTML, voici ce qu'il se passe lorsqu'un utilisateur clique sur le marqueur d'un hôpital :

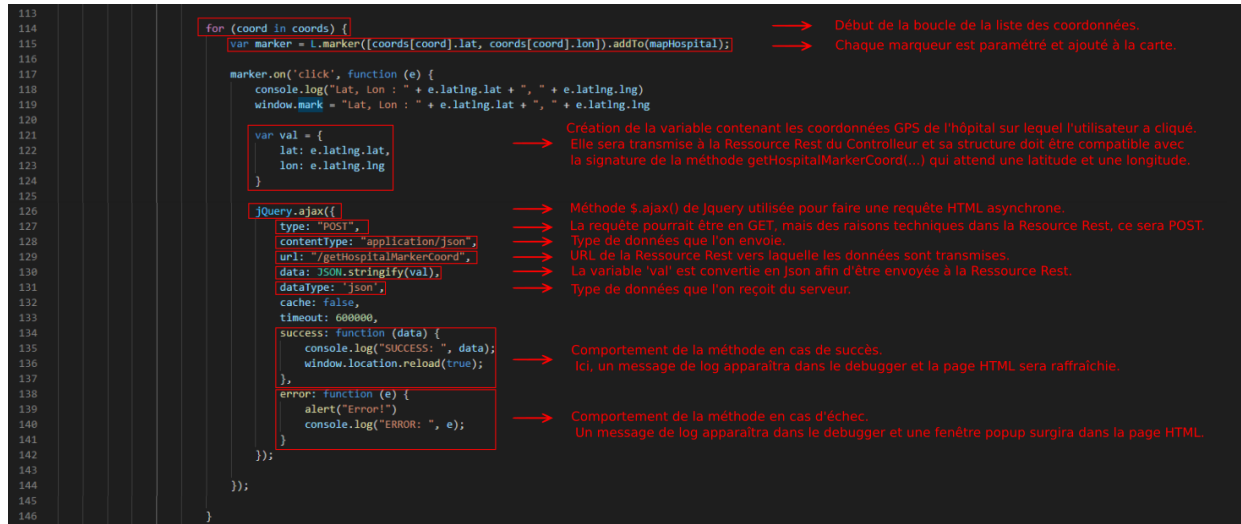


FIGURE 23 – Envoi des coordonnées du JavaScript au contrôleur lors du clique sur le marqueur d'un hôpital

Du côté back, voici la ressource qui réceptionne les coordonnées transmises depuis le front :

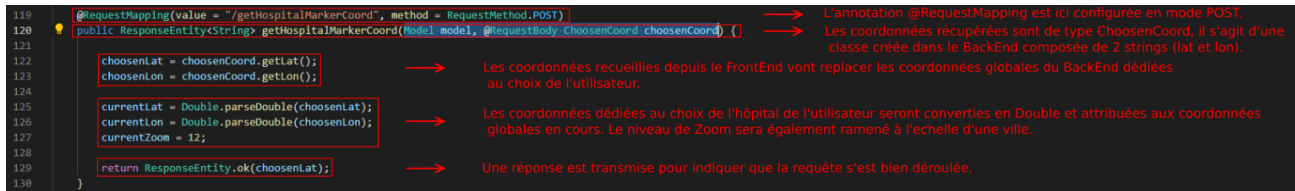


FIGURE 24 – Réception des coordonnées dans le back

Toujours du côté back, voici la ressource qui fait office de méthode principale dans la construction des modèles et transmission des données de la carte vers le front :

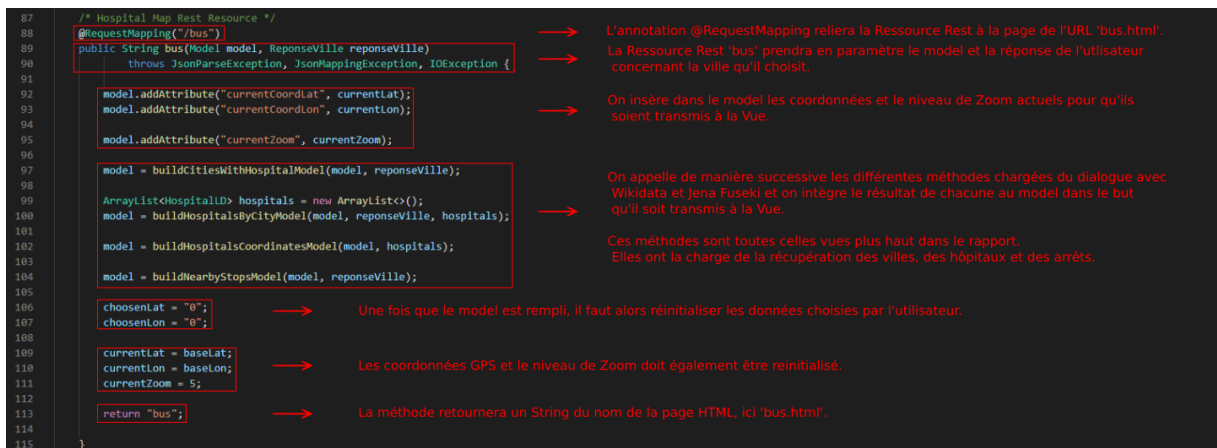


FIGURE 25 – Construction des modèles, définition des paramètres de la carte et envoi des données au front

6 Côté triplestore (localhost :3030)

6.1 Ajout des données

L'ajout des données est réalisé dans la classe `Record.java` du package `com.semweb.map.jena`. Après avoir lancé le serveur Apache Jena Fuseki à l'adresse par défaut [<http://localhost:3030>] et créer dessus un nouveau ensemble de données `hospitals`. Au lancement de l'application à l'adresse [<http://localhost:8080>], l'ajout des hôpitaux dans le triplestore s'effectuera de façon automatique. La récupération des différentes informations sur les hôpitaux définies dans la section ontologie est effectuée en interrogeant Wikidata à l'adresse [<https://query.wikidata.org/sparql>] avec une requête SPARQL de type `SELECT`.

Dans cette requête, `DISTINCT` est ajouté pour éviter de récupérer des doublons. Ensuite, `SAMPLE` est utilisé pour ne récupérer qu'une seule valeur dans le cas où, pour une raison quelconque, une propriété possède plusieurs valeurs. On a déjà confronté à des situations où un hôpital possède plusieurs coordonnées, pour éviter de déclencher des erreurs par la suite pendant le parsing ou l'affichage des données, on ne prendra qu'une seule valeur aléatoire. L'utilisation d'une telle agrégation doit être couplée avec `GROUP BY` sur l'identifiant de l'hôpital. On utilise également `OPTIONAL` devant certaines propriétés non essentielles afin d'avoir un nombre maximal de résultats. Enfin, on effectue un `FILTER` pour ne récupérer que les noms des hôpitaux et des villes qui sont disponibles en français, étant donnée que l'application est proposée principalement à des utilisateurs francophones.

Dans la clause `WHERE` on vérifie également que la ressource est de type hôpital (Q16917) et qu'elle se trouve en France (Q142). Après exécution de la requête, les informations seront récupérées comme suit :

Propriété de l'hôpital	Variable avant SAMPLE	Variable après SAMPLE	Récupération
Identifiant	?item	∅	Obligatoire
Nom	?name	?n	Obligatoire
Ville	?city	?c	Obligatoire
Image	?pic	?p	Optionnelle
Nombre de lits	?bed	?b	Optionnelle
Adresse	?str	?s	Optionnelle
Site web	?web	?w	Optionnelle
Coordonnées	?geo	?g	Obligatoire
Lien vers Wikipedia EN	?wikien	?we	Optionnelle
Lien vers Wikipedia FR	?wikifr	?wf	Optionnelle

Les informations ci-dessus seront par la suite stockées dans une classe définie dans `Hospital.java` du même package. Une fois que toutes les rangées du résultat sont traitées, une liste d'hôpitaux sera envoyée à la méthode avec la signature `void model()` qui stockera ces données sur Apache Jena Fuseki.

Au début, cette méthode crée un modèle RDF vide. Ensuite, elle attache à ce modèle les préfixes, pointant vers des liens réels, de l'ontologie définie dans le projet, puis prépare en fonction des préfixes les propriétés et les ressources adéquates. Enfin, elle boucle sur la liste des hôpitaux, forme un triplet de type `Sujet (Identifiant de l'hôpital) + Propriété (Type de l'information) + Objet (Valeur de la propriété)`, et ajoute ce triplet au modèle.

Une fois que le remplissage du modèle est terminé, une connexion à l'adresse [<http://localhost:3030/hospitals>] est établie pour stocker le modèle RDF dans l'ensemble des données `hospitals` qui se trouve sur le serveur Apache Jena Fuseki.

6.2 Extraction des données

L'interrogation du triplestore Apache Jena Fuseki ainsi que du Wikidata est réalisé dans la classe `request.java` du package `com.semweb.map.jena`. Dans cette classe, on effectue 3 requêtes SPARQL distribuées dans des méthodes avec une signature comme suit :

- *String* `getCities()`
- *Map < String, String >* `getHospitalsByCity(String city)`
- *Map < String, String >* `getNearbyStations(Double lon, Double lat)`

6.2.1 Récupération des villes

On utilise dans la méthode `getCities()` une requête de type `CONSTRUCT` dans le but de construire un graphe avec toutes les villes qui se trouvent dans l'ensemble de données `hospitals`. Le modèle résultant est transformé en `JSON-LD` puis envoyé au contrôleur à travers un objet `String`.

6.2.2 Récupération des hôpitaux

Les hôpitaux sur la page web seront affichés en fonction de la ville. La méthode `getHospitalsByCity(String city)` aura besoin de récupérer en paramètres le nom de la ville sélectionnée par l'utilisateur qui sera ensuite passé à la clause `WHERE` de la requête de type `CONSTRUCT`. Comme la précédente requête, un graphe contenant les informations sur les hôpitaux de la ville choisie est créé, converti en `JSON-LD` puis stocké dans un objet `String`. Cet objet est ensuite ajouté à une structure `Map`, car on aura besoin de passer une information supplémentaire au contrôleur : la taille du graphe. En effet, nous avons remarqué que la structure d'un `JSON-LD` contenant un seul hôpital, est différente de celle contenant plusieurs hôpitaux. Par conséquent, le contrôleur doit être au courant de la taille du graphe pour pouvoir déclencher le bon parseur. La taille d'un graphe sera calculé avec la formule :

$$\frac{model.size()}{construct.split(";").length}$$

La variable `model` désigne l'objet du package `org.apache.jena.rdf.model.Model` contenant le graphe du résultat. Appliquer la méthode `size()` sur cet objet, nous permettra d'avoir le nombre de noeuds composant ce graphe. Ensuite, il faut diviser par le nombre de triplets appartenant à la clause `CONSTRUCT` pour avoir le nombre d'hôpitaux dans le graphe. Dans notre requête, les triplets sont séparés par un `;` vu qu'ils appartiennent tous au même sujet, dans le cas où cette règle est brisée, la formule de calcul doit être adaptée. Enfin, le résultat de la formule sera ajouté, à son tour, à la structure `Map`, puis cette dernière sera passée au contrôleur.

6.2.3 Récupération des stations

Pour atteindre un des objectifs de l'application, la méthode `getNearbyStations(Double lon, Double lat)` se chargera de récupérer les stations de transport public qui sont à proximité d'un hôpital sélectionné par l'utilisateur. À la différence des 2 requêtes précédentes, c'est Wikidata qui sera interrogé au lieu de l'ensemble de données `hospitals` qui ne contient pas d'informations sur les stations. L'esprit de la requête sera le même, on utilise un `CONSTRUCT` pour générer un graphe contenant les coordonnées et les noms des stations récupérés à partir de Wikidata, la sortie du graphe sera en `JSON-LD`, stockage dans un objet `String`, puis dans une structure `Map` en appliquant le même principe qu'avec la récupération des hôpitaux, pour pouvoir distinguer les graphes avec une seule station, et ceux avec plusieurs stations.

Nous avons remarqué, comme avec les hôpitaux, que certaines stations possèdent plusieurs coordonnées. Sauf que dans ce cas on utilise un `CONSTRUCT` et on n'a pas réussi à trouver un équivalent au `SAMPLE` utilisait dans la requête `SELECT` de l'ajout des données. Pour éviter les erreurs lors du traitement du graphe, nous avons décidé d'ignorer les stations possédant plusieurs coordonnées.

La méthode récupère en paramètres les coordonnées de l'hôpital où l'utilisateur a cliqué sur la carte à travers 2 objets de type `Double` qui désignent la longitude et la latitude. Lors de la requête ces objets sont concaténés pour former un point, comme le montre cette partie du code :

```
POINT(" + lon + " " + lat + ")
```

Avec `lon` et `lat` désignant respectivement la longitude et la latitude de l'hôpital cliqué. On aura maintenant la possibilité de comparer ces coordonnées, aux coordonnées des stations avec cette partie du code :

```
FILTER(geof:distance(?geo1, \"POINT(" + lon + " " + lat + ")\") <= 5) .
```

Avec `?geo1` désignant les coordonnées d'une station (Q548662) qui se trouve en France (Q142). On filtre les résultats pour récupérer uniquement les stations qui sont à 5 km au moins de distance de l'hôpital cliqué. Nous avons choisi une telle distance car sur Wikidata on ne trouve pas assez d'informations sur les stations des petites villes, il faut élargir le cercle de recherche sur 5 km pour avoir des résultats concrets. De même, on limite la requête à 10 résultats pour éviter d'avoir un excès de résultats dans les grandes villes. Le calcul de la distance est réalisé grâce à la fonction [<http://www.opengis.net/def/geosparql/function/distance>] de la norme `GeoSPARQL` qui applique la formule d'Haversine.

6.3 Stockage des données

	subject	⌘	predicate	⌘	object	⌘
1	<Q36698330>		<http://dbpedia.org/ontology/Website>		"http://www.ch-jury.fr"	
2	<Q36698330>		<http://dbpedia.org/ontology/name>		"Centre hospitalier spécialisé de Jury"	
3	<Q36698330>		<http://dbpedia.org/ontology/bedCount>		""	
4	<Q36698330>		<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>		<http://www.wikidata.org/wiki/Q16917>	
5	<Q36698330>		<http://dbpedia.org/ontology/address>		""	
6	<Q36698330>		<http://dbpedia.org/ontology/picture>		"http://commons.wikimedia.org/wiki/Special:FilePath/Chapelle%20Jury.JPG"	
7	<Q36698330>		<http://www.w3.org/2006/vcard/ns#longitude>		"6.2542"	
8	<Q36698330>		<http://purl.org/ontology/mo/wikipedia>		""	
9	<Q36698330>		<http://www.w3.org/2006/vcard/ns#latitude>		"49.0706"	
10	<Q36698330>		<http://dbpedia.org/ontology/city>		"Jury"	
11	<Q36698330>		<http://www.geonames.org/ontology/documentation.html#wikipediaArticle>		"https://fr.wikipedia.org/wiki/Centre_hospitalier_sp%C3%A9cialis%C3%A9_de_Jury"	
12	<Q16507435>		<http://dbpedia.org/ontology/Website>		"http://www.ch-guillaumeregner.fr"	
13	<Q16507435>		<http://dbpedia.org/ontology/name>		"centre hospitalier Guillaume-Régner"	
14	<Q16507435>		<http://dbpedia.org/ontology/bedCount>		""	
15	<Q16507435>		<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>		<http://www.wikidata.org/wiki/Q16917>	

Showing 1 to 15 of 15 entries

FIGURE 26 – Aperçu de l'ensemble de données hospitals

Le tableau résultant de l'exécution d'une requête SPARQL par l'interface de Apache Jean Fuseki pour avoir 15 des triplets enregistrés sur l'ensemble de données hospitals. On remarque que les données enregistrées respectent la notion de triplet : Sujet + Propriété + Objet. Le sujet (subject) sera équivalent à l'identifiant d'un hôpital sur Wikidata. Les propriétés (predicate) définissent l'ontologie visualisé par le projet, ils utilisent un vocabulaire existant déjà. L'objet (object) sera la plupart du temps une valeur de type String, sauf pour la propriété `rdf:type` où la ressource pour désigner un hôpital sur Wikidata sera utilisé. Dans le cas où une propriété ne contienne pas de valeur sur Wikidata, on stockera une chaîne vide qui sera ignorée pendant le parsing. Bien qu'on utilise le type String pour stocker les valeurs, la conversion nécessaire sera effectuée au niveau du serveur de l'application sur certaines propriétés pour avoir le bon type.

7 Fonctionnalités non implémentées

7.1 Insertion de données dans la base de Jena Fuseki

Le premier formulaire crée pour l'application est aussi celui qui a toujours servi à ce qu'il n'était pas au départ. Initialement, la page `index.html` a été un champ d'expérimentation pour tester l'intégration des divers fonctions au fur et à mesure du développement. Après avoir permis de s'assurer que les échanges entre back et front, puis inversement, se déroule bien, ce formulaire a été pendant un long moment du développement le mécanisme qui nous permettait d'initialiser l'application au démarrage. Nous démarrions les serveurs puis nous nous rendions sur la page `index.html` afin de récupérer les données sur Wikidata et les insérer dans Apache Jena Fuseki. L'automatisation de cette étape d'initialisation a été l'une de nos tâches primordiales, de manière à améliorer l'usage de l'application pour tout utilisateur, autre que le développeur. Très tôt dans le projet, nous avons prévu de laisser à l'utilisateur la possibilité de rajouter des villes, des hôpitaux ou des arrêts par l'intermédiaire de 3 formulaires dédiés. Mais nous transmettons déjà de plusieurs manières (formulaire, Ajax) des informations du front au back dans la page `bus.html`, rajouter une requête d'insertion de données pour nourrir la base de Apache Jena Fuseki n'aurait selon nous rien apporter de plus en dehors de rajouter une fonction pas nécessairement indispensable, aussi nous avons préféré mettre l'accent sur la navigation de l'utilisateur à travers les hôpitaux de France grâce au formulaire des villes et des marqueurs de la carte. Faute de temps, il n'a pas été possible de mener cette tâche à terme.

7.2 Récupération des informations depuis une autre source de données

Quiconque (chez les développeurs) sait combien parser un document bien formaté est agréable et aisé, sans forêts de noeuds compliqué à gérer ni structure sans queue ni tête. Un document CSV est dans ce sens l'ami du développeur, dès qu'on a trouvé les caractères de séparation, récupérer les informations qu'il contient et les transformer en objet (Java), devient un jeu d'enfant. Aussi c'est d'autant plus agaçant voire rageant de constater que dans aucun des projets scolaires, il n'ait été possible d'exploiter le moindre fichier CSV, aucun. Quel est l'intérêt de pouvoir parser facilement des données d'une source si ces données se révèle être inexploitable ?

Nous avons pour but de pouvoir exploiter plusieurs sources d'informations, à savoir dans notre cas une liste d'hôpitaux ou d'arrêts de bus. Mais la totalité des documents CSV trouvés étaient inutilisables car incomplets. Si le site data.gouv.fr permet de trouver des échantillons de bases de données dans des domaines variés, force est de constater que le cas où nous pourrions utiliser ce que nous y avons trouvé ne s'est jamais présenté. Nous avons trouvé des fichiers pour les hôpitaux ou les arrêts de bus mais nettoyer des données des caractères spéciaux non reconnues ou bien ne comportant aucune entité disposant d'un jeu de données complet (nom complet, coordonnées GPS, ville, ...).

A1				FID						
	A	B	C	D	E	F	G	H	I	J
1	FID									
2	france_hospitals_point.41497406	41497406	hospital	Centre Hospitalier de Saint-Laurent-du-Pont						
3	france_hospitals_point.3253864247	3253864247	hospital	Port Louis-Riantec						
4	france_hospitals_point.4307578009	4307578009	hospital	Croix Bleue						
5	france_hospitals_point.702147381	702147381	hospital	Centre hospitalier de Coulommiers	GHEF		Hôpital Abel Leblanc		Grand Hôpital de l'Est Francilien	public
6	france_hospitals_point.-2518786	-2518786	hospital	Hôpital de St Geniez d'Olt						
7	france_hospitals_point.562894624	562894624	hospital	Centre Hospitalier Ferdinand Grall						
8	france_hospitals_point.646792094	646792094	hospital	Centre Hospitalier Georges Heuyer						
9	france_hospitals_point.190401225	190401225	hospital	Hospital de Cerdanya						
10	france_hospitals_point.686698759	686698759	hospital	Hôpital Privé des Cailles d'Armor						
11	france_hospitals_point.53435124	53435124	hospital	Bâtiment 101 - Centre de protonthérapie		Institut Curie - Centre de protonthérapie		Institut Curie		private_non
12	france_hospitals_point.76281073	76281073	hospital	Hôpital Flaubert				Groupe hospitalier du Havre		public
13	france_hospitals_point.80608013	80608013	hospital	Établissement Public de Santé Mentale						
14	france_hospitals_point.289201202	289201202	hospital	Centre Hospitalier de Sarrebourg, Site de Hoff						
15	france_hospitals_point.389190750	389190750	hospital	Centre Hospitalier de Nîmes-Les-Bains						
16	france_hospitals_point.72366917	72366917	hospital	Polyclinique de Franche-Comté						
17	france_hospitals_point.133168941	133168941	hospital	Centre Psychothérapie de l'Orne						
18	france_hospitals_point.7609044081	7609044081	hospital	Pharmacie de Guerledan						
19	france_hospitals_point.777989718	777989718	hospital	Hôpital Elisée Charra Lamastre		Hôpital Elisée Charra		Hôpital Locale Lamastre		
20	france_hospitals_point.7248066786	7248066786	hospital	Dieulefit Santé						
21	france_hospitals_point.561868812	561868812	hospital	Centre hospitalier d'Altkirch						public
22	france_hospitals_point.187423601	187423601	hospital	Centre Hospitalier Spécialisé d'Altkirch						
23	france_hospitals_point.38701149	38701149	hospital	Centre Hospitalier de Chauny						public
24	france_hospitals_point.673936038	673936038	hospital	Maison de santé de l'Epte						
25	france_hospitals_point.149900875	149900875	hospital	Clinique La Mare						
26	france_hospitals_point.25745753	25745753	hospital	Maison de santé protestante de Bagatelle						
27	france_hospitals_point.142064822	142064822	hospital	Polyclinique des 3 Frontières						private
28	france_hospitals_point.355306681	355306681	hospital	Centre Hospitalier Régional d'Orléans	CHRO		Hôpital de la Source			public

FIGURE 27 – Fichier CSV des hôpitaux

Le fichier CSV des hôpitaux que nous avons trouvé nous imposait même un format de coordonnées géographiques (une variante du Web Mercator) qu'il nous a été très difficile d'identifier, puis de pouvoir convertir en trouvant quelle formule de calcul était appliquée, afin de l'implémenter en tant que méthode dans notre application. Au final, plus que le manque de temps, ce sera l'absence du caractère complet de la source de données et de ses informations parfois exotiques qui nous ont détourné d'elles.

7.3 Véritable fonctionnement asynchrone de la page web

L'usage de la méthode `jQuery.ajax()` dans le script JavaScript de la page `bus.html`, destiné à transmettre des données de la vue au contrôleur, a ouvert la voie à l'introduction de l'asynchrone pour notre page principale, afin que celle-ci ne charge que les parties mises à jour. Dans la version rendue du projet, nous n'avons pas eu le temps de faire le nécessaire pour ne pas avoir à recharger la page à chaque changement de ville, d'hôpital ou de marqueur. Nous aurions pu développer la majorité de la page en Ajax ou bien employer un Framework JavaScript comme VueJS ou Angular, afin d'intégrer cette fonctionnalité de manière quasi native. Mais comme il a été dit plus haut dans le rapport, cette dernière solution nous aurait contraint à rajouter l'usage d'un autre serveur. Comme nous savions que nous ne pourrions pas présenter nous même notre application au cours d'une soutenance et qu'elle devrait être pouvoir lancée après avoir lu un simple Readme, nous avons pensé qu'il vaudrait mieux simplifier la structure de l'application, dans un contexte où le temps est une denrée beaucoup trop rare.

8 Perspectives

Pendant ce projet, nous nous sommes focalisés sur des axes majeures, à savoir l’affichage des coordonnées géo spatiales des structures (les hôpitaux et les stations dans notre cas), récupérées de différentes sources, sur une page web en appliquant le concept de web sémantique et d’interopérabilité. Cependant d’autres améliorations peuvent être effectuées à l’état actuel de l’application :

- Donner la possibilité à l’utilisateur de rentrer des données sur les hôpitaux et les stations à travers l’application. Cette fonction permettra de compléter les informations non trouvées dans nos sources de données. En revanche, la saisie de l’utilisateur doit être contrôlée, par un administrateur par exemple ou un système de vote, pour éviter les informations erronées.
- Pour l’instant, la distance requise pour récupérer les stations proches d’un hôpital est d’au maximum 5 km. L’idée est de laisser à l’utilisateur le choix de cette distance tant qu’elle reste raisonnable, C’est-à-dire une valeur positive et qu’elle ne dépasse pas un certain seuil (un seuil de 30 km par exemple). On pourra aussi laisser à l’utilisateur le choix du nombre de stations qui seront affichées, qui est fixé à 10, la valeur sera également contrôlée dans ce cas.
- L’affichage des adjacences des stations ainsi que les horaires de pointe des transports en public rendra l’application plus enrichissante en informations. De plus la liaison entre les données d’un hôpital dans le tableau avec ses coordonnées sur la carte permettra une lecture plus compréhensible.

9 Conclusion

Dans ce projet nous avons manipulé des données géo spatiales dans une approche différente de ce qu’on a l’habitude de faire en développement web. Les données étaient traitées comme étant des triplets en utilisant le modèle de graphe RDF en syntaxe JSON-LD, en suivant une ontologie définie de base pour assurer une homogénéité entre les différentes sources de données.

Références

- [1] Java Code Geeks :
<https://www.javacodegeeks.com/2015/05/rethinking-database-schema-with-rdf-and-ontology.html>
- [2] API de l'école des Mines de Saint-Étienne :
<https://ci.mines-stetienne.fr/sparql-generate/get-started.html>
- [3] Developpez :
<https://web-semantique.developpez.com/tutoriels/jena/arq/introduction-sparql/>
- [4] Apache Jena :
<https://jena.apache.org/>
- [5] Dskow Github :
<https://dskow.github.io/2017/07/12/crud-repo-inject-spring-boot.html>
- [6] Baeldung :
<https://www.baeldung.com/json-linked-data>
- [7] Thymeleaf :
<https://www.thymeleaf.org/>
- [8] Spring :
<https://spring.io/guides/gs/serving-web-content/>
- [9] Codeflow :
<https://www.codeflow.site/fr/article/thymeleaf-in-spring-mvc>
- [10] Quick Start :
<https://leafletjs.com/examples/quick-start/>
- [11] Markers :
<https://leafletjs.com/examples/custom-icons/>
- [12] SPARQL Tutorial :
https://www.wikidata.org/wiki/Wikidata:SPARQL_tutorial
- [13] SPARQL Query :
<https://query.wikidata.org/>
- [14] GeoSPARQL :
<https://www.ogc.org/standards/geosparql>
<https://www.navigae.fr/geosparql/doc/>