BLARD Christopher

Internship report - GeoTimeWFS

i3Mainz / BKG

ESIREM

Under the supervision of Dr. Claire Prudhomme and Dr. Jean-Jacques Ponciano

July 2020

Index

List of figures

Acknowledgement

First, I would like to thank Dr. Jean-Jacques Ponciano et Dr. Claire Prudhomme who gave me the opportunity to work with them on a big project, and for their help during this training period.

Thanks to Charles Meunier, teacher at ESIREM, who recommended me for this internship.

Thanks to Jonas Veller from i3Mainz for his help on the Leaflet library.

I would also like to thank Pierre Mazurek with whom I worked during this training period.

Finally, I am grateful to Dr. Falk Würriehausen and Eike Schott, working for the Federal Agency for Cartography and Geodesy, who were always open to any question and motivating about our progress on the project.

General introduction

GeoTimeWFS (**WFS** standing for **W**eb **F**eature **S**ervice) is an application using the Semantic Web to create maps based on different geographic data. This data can be data imported by users or can be learned by the machine from “open” sites like Wikidata[1] which provides linked data. This is where the "WFS" comes in, since this protocol makes it possible to query mapping servers in order to be able to manipulate objects on maps (points, lines, areas, etc.).

The purpose of this application is to enrich geographic data with open data. The maps can then be created dynamically, or manually by the user. Development of this application started two years ago and is being carried out by researchers from the i3Mainz[2] institute from University of Mainz on behalf of the **BKG**[3] (German: **B**undesamt für **K**artographie und **G**eodäsie, English: Federal Agency for Cartography and Geodesy).

During this internship, I was able to work on the development of this application, especially on the "thematic maps" part, which I will present later.

My main tasks during this internship were to process geographic data files, to use geographic data from open sites and to create maps from this data. I also worked on improving the user experience on the application.

In this report, I will first present the application and its technology in a global view, and then I will detail the tasks that I have handled, before concluding with the problems that I have encountered and the next steps that could be achieved.

1. Application overview

The GeoTimeWFS application provided access to many features which will be presented below. Most of the functionalities are interrelated. To use any of these features, it is necessary to master them all. The features that will be presented are part of the final prototype of the application but may not be implemented at the time of writing.

I. Technology

The overall principle of the application is to store geospatial data in an ontology and then retrieve this data to create maps and display them.

The application is developed using the Spring[4] Java framework, which operates according to the Model-View-Controller architecture. For example, for each functionality of the application, there will be two Java files : the controller which will contain methods and values, the model which will contain definition of classes, and the view in HTML/Javascript which will display the content for the user.

As expected in the MVC architecture, the view will therefore manipulate the controller, the controller will update the model, and the view will then have to read the model.

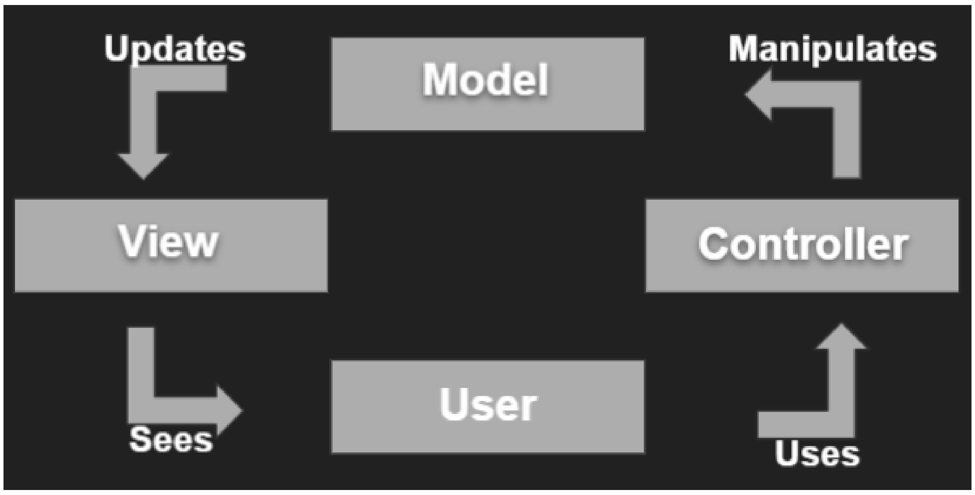


Figure 1: Model-View-Controller architecture

II. Menus

Figure 2 shows the menu bar as it should appear at the end of application development.

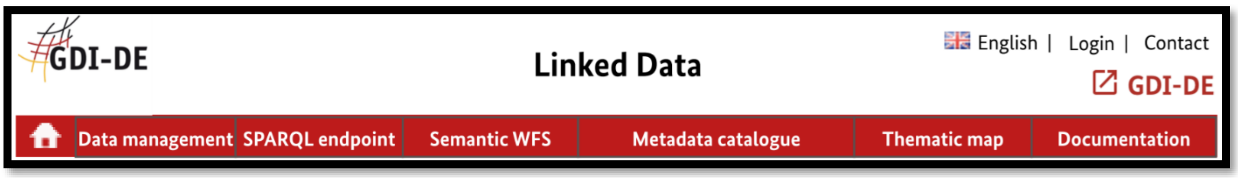


Figure 2: Final prototype menu bar

The **Data Management** menu is meant to upload data files and check their content. The different types of data files can be GeoJSON files (JSON files for geospatial data), shapefiles or even XSD files (definition of XML schemas). The information can also be collected via open data such as data available on Wikidata.

In the **SPARQL Endpoint** menu, the user will be able to perform SPARQL queries to find different types of information in RDF data. RDF is the language of the Semantic Web and is based on Subject, Object, Predicate triples. SPARQL queries can be used in the application ontology or to retrieve data opened via Wikidata.

The **Semantic WFS** menu will offer the possibility to browse in the ontology.

The **Metadata Catalog** menu will give the opportunity to import metadata files and link them to data.

The **Thematic maps** menu will allow the user to create specific maps with different layers of information according to their needs. For example, it will be possible to search for schools in a region, universities in a city, hospitals in the country, etc. This is the part I mostly worked on during the internship.

The **Documentation** menu will contain documentation on all the features of the application.

Among all these features, we will now focus on processing geographic data, creating some examples of SPARQL queries and displaying thematic maps.

2. Task 1: Handle shapefiles and GeoJSON files

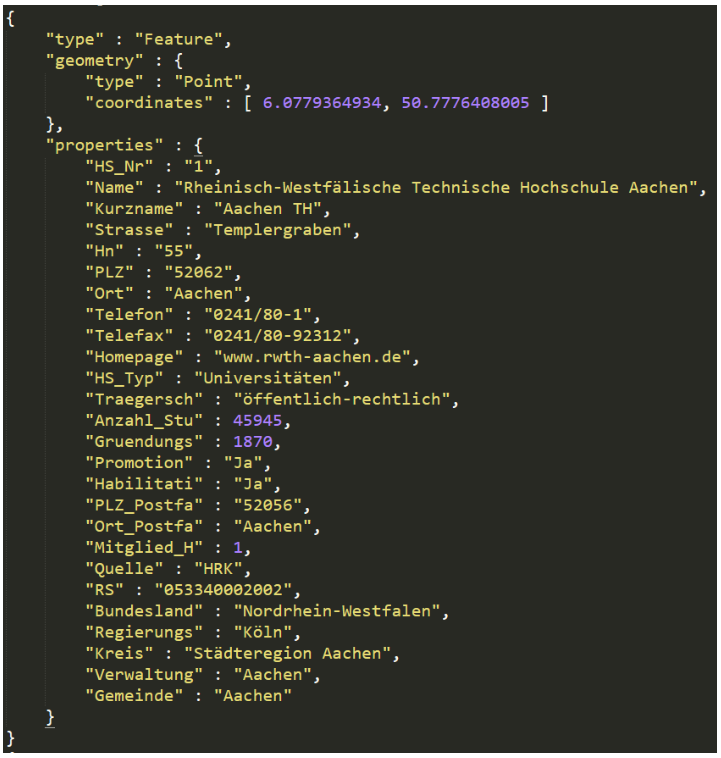
The two main file formats on which the application is based are going to be GeoJSON and shapefile, two formats that allow you to encode geographic data.

I. GeoJSON files

The GeoJSON is a JSON format suitable for geographical data. You can create different types of objects such as points, lines or polygons. You can also create a "Feature" that is to say an object with other properties than basic set.

Figure 3 shows a "Feature" of a German university with its basic attributes (type: a point on the map, its coordinates, its name) but also many added attributes (telephone number, website, city, taught subject) that we can use later.

For example, with all this information, it would be possible to create a map of all the universities that teach public law, which would show the university from Figure 3.



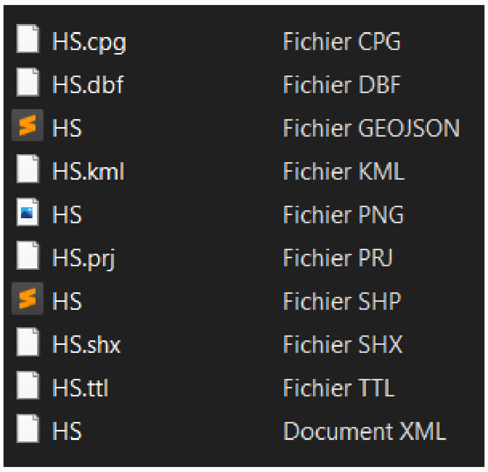
```
{
    "type" : "Feature",
    "geometry" : {
        "type" : "Point",
        "coordinates" : [ 6.0779364934, 50.7776408005 ]
    },
    "properties" : {
        "HS_Nr" : "1",
        "Name" : "Rheinisch-Westfälische Technische Hochschule Aachen",
        "Kurzname" : "Aachen TH",
        "Strasse" : "Templergraben",
        "Hn" : "55",
        "PLZ" : "52062",
        "Ort" : "Aachen",
        "Telefon" : "0241/80-1",
        "Telefax" : "0241/80-92312",
        "Homepage" : "www.rwth-aachen.de",
        "HS_Typ" : "Universitäten",
        "Traegersch" : "öffentlich-rechtlich",
        "Anzahl_Stu" : 45945,
        "Gruendungs" : 1870,
        "Promotion" : "Ja",
        "Habilitati" : "Ja",
        "PLZ_Postfa" : "52056",
        "Ort_Postfa" : "Aachen",
        "Mitglied_H" : 1,
        "Quelle" : "HRK",
        "RS" : "053340002002",
        "Bundesland" : "Nordrhein-Westfalen",
        "Regierungs" : "Köln",
        "Kreis" : "Städteregion Aachen",
        "Verwaltung" : "Aachen",
        "Gemeinde" : "Aachen"
    }
}
```

Figure 3: GeoJSON file sample

However, to create relevant maps, a lot more information and points are necessary. This is why the GeoJSON format allows you to create "Feature Collections" that list a category of points for example. The university in Figure 3 is part of a feature collection of over 500 universities, provided by BKG.

II. Shapefile

The shapefile is another possible format for processing geographic data. This is a folder with several files with the same name but different extensions, as shown in Figure 4. This file contains information on German universities (HS = Hochschule, “University” in German).



HS.cpg	Fichier CPG
HS.dbf	Fichier DBF
HS	Fichier GEOJSON
HS.kml	Fichier KML
HS	Fichier PNG
HS.prj	Fichier PRJ
HS	Fichier SHP
HS.shx	Fichier SHX
HS.ttl	Fichier TTL
HS	Document XML

Figure 4: Shapefile sample

The three types of data required in a shapefile are .shp, .shx, and .dbf. The .shp is the main file and the .shx is the shapefile index.

The other files can contain the icon that each point must take (instead of the icon provided on the map), the different shapes that can be displayed on the map, metadata files, etc.

These two file formats are used to process the data included and to store those data in the ontology in order to be able to display them on certain maps.

3. Task 2: SPARQL request examples

Once the GeoJSON files or shapefiles are imported into the application, the application will keep them in memory in the ontology. The application will then "learn" from these files and will therefore add options for creating the maps. It is also possible to enrich this data with linked data from Wikidata.

I. Introduction to SPARQL requests

This information is then available in two ways: via ontology, or via open data. In both cases, the information is stored as an RDF triplet. An RDF triplet is a Subject-Predicate-Object association. Simply put, the example given by Wikipedia[5] is as follows:

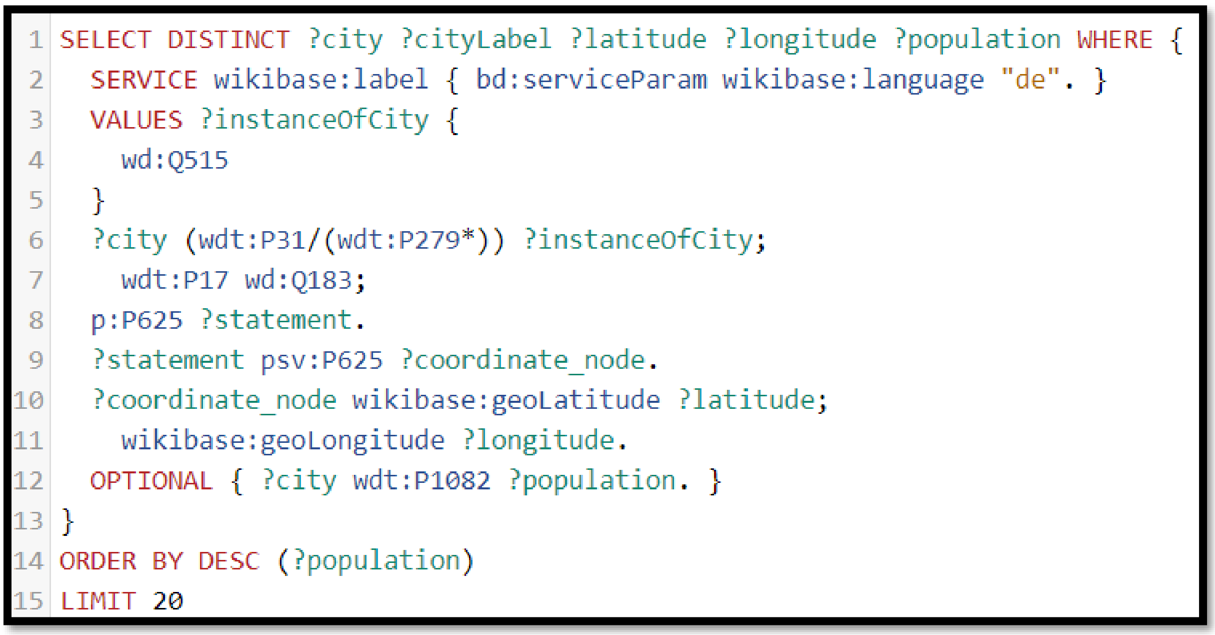
"The sky has the color blue" ➔ "The sky" is the subject, "has the color" is the predicate, "blue" is the object. Objects are stored this way.

SPARQL is a query language for databases used to manipulate RDF data. This is the language we will use to retrieve data from Wikidata, or from the ontology.

At the time of writing this report, on the Linked Data Enrichment page, there are about twenty queries available, including the list of schools, hospitals, police stations, libraries, cinemas, etc. The following examples will also be part of the sample queries and it will possible to display the results on a map on Linked Data Enrichment page.

II. Example 1 : 20 biggest cities in Germany

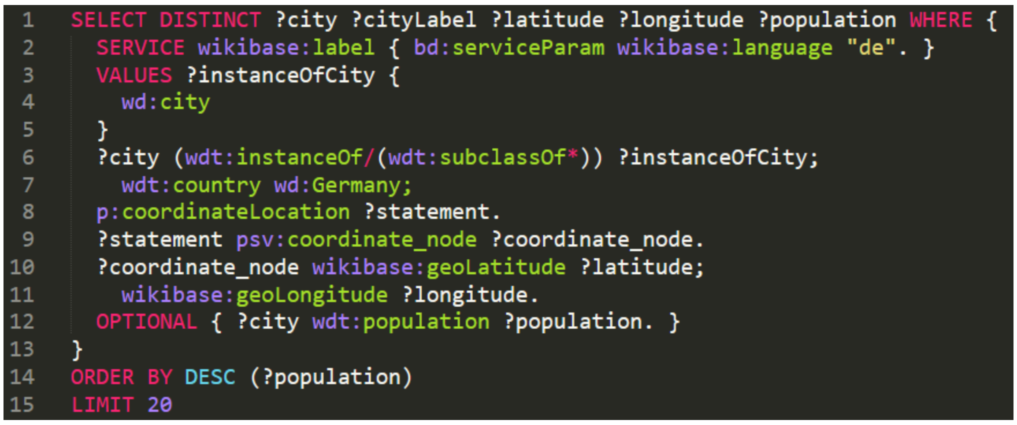
In the following example, we will find the 20 biggest cities in Germany sorted by population on Wikidata. The query presented in figure 5 is executed on the Wikidata Query Service[6] site.



```
SELECT DISTINCT ?city ?cityLabel ?latitude ?longitude ?population WHERE {
  SERVICE wikibase:label { bd:serviceParam wikibase:language "de". }
  VALUES ?instanceOfCity {
    wd:Q515
  }
  ?city (wdt:P31/(wdt:P279*)) ?instanceOfCity;
    wdt:P17 wd:Q183;
  p:P625 ?statement.
  ?statement psv:P625 ?coordinate_node.
  ?coordinate_node wikibase:geoLatitude ?latitude;
    wikibase:geoLongitude ?longitude.
  OPTIONAL { ?city wdt:P1082 ?population. }
}
ORDER BY DESC (?population)
LIMIT 20
```

Figure 5: SPARQL request to get the 20 most populous cities in Germany

The different indexes such as P31 or Q183 correspond to specific data: “city”, “Germany”, … These data have been replaced in Figure 6.



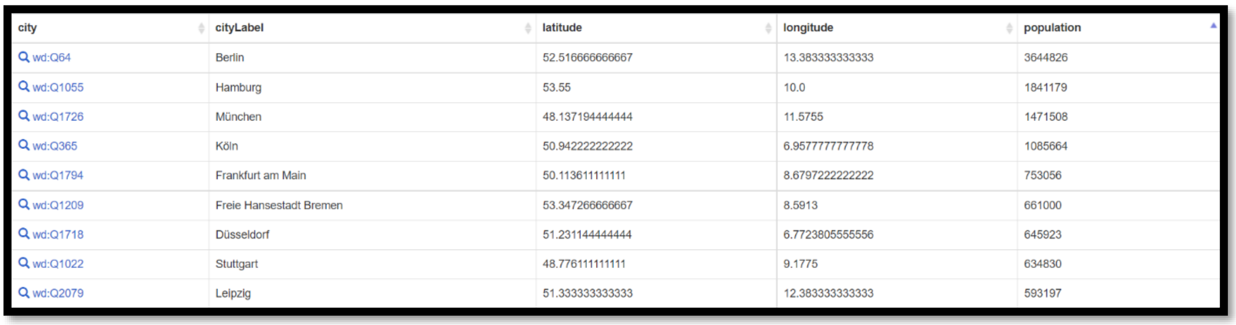
```
SELECT DISTINCT ?city ?cityLabel ?latitude ?longitude ?population WHERE {
  SERVICE wikibase:label { bd:serviceParam wikibase:language "de". }
  VALUES ?instanceOfCity {
    wd:city
  }
  ?city (wdt:instanceOf/(wdt:subclassOf*)) ?instanceOfCity;
    wdt:country wd:Germany;
  p:coordinateLocation ?statement.
  ?statement psv:coordinate_node ?coordinate_node.
  ?coordinate_node wikibase:geoLatitude ?latitude;
    wikibase:geoLongitude ?longitude.
  OPTIONAL { ?city wdt:population ?population. }
}
ORDER BY DESC (?population)
LIMIT 20
```

Figure 6: Same request as figure 3, with indexes replaced by « real » values

The query therefore corresponds to the following question:

Among "cities", "which of them are" "in Germany"?

Once this request has been made, we will take care to collect the coordinates as well as the respective populations of each city. The end of the query is just to sort the results in decreasing order of population (***ORDER BY DESC (?population)***), and then keep only the first twenty results (***LIMIT 20***). The first lines of the query results are shown in Figure 7.



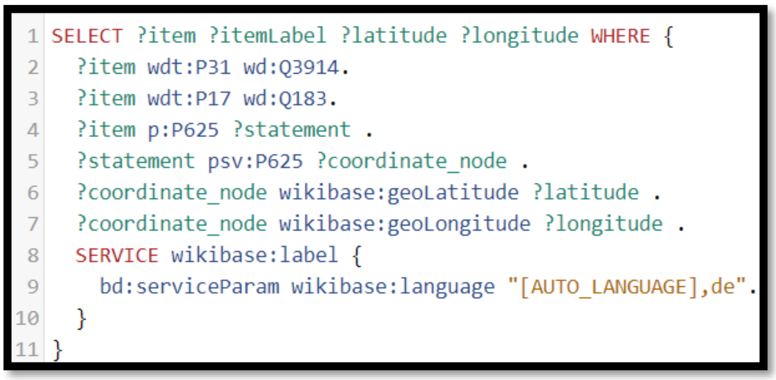
city	cityLabel	latitude	longitude	population
wd:Q64	Berlin	52.516666666667	13.383333333333	3644826
wd:Q1055	Hamburg	53.55	10.0	1841179
wd:Q1726	München	48.137194444444	11.5755	1471508
wd:Q365	Köln	50.942222222222	6.9577777777778	1085664
wd:Q1794	Frankfurt am Main	50.113611111111	8.6797222222222	753056
wd:Q1209	Freie Hansestadt Bremen	53.347266666667	8.5913	661000
wd:Q1718	Düsseldorf	51.231144444444	6.7723805555556	645923
wd:Q1022	Stuttgart	48.776111111111	9.1775	634830
wd:Q2079	Leipzig	51.333333333333	12.383333333333	593197

Figure 7: First results for the previous request

The results of this request can be stored in a JSON file and then downloaded directly on the website, or retrievable in Java via the controllers. Once these requests have been made in Java, it is possible to transmit these values to the HTML / Javascript file to retrieve the coordinates and display them on a map.

III. Example 2 : All schools in Germany

The example in figure 8 retrieves the list of all schools in Germany that are listed on Wikidata.

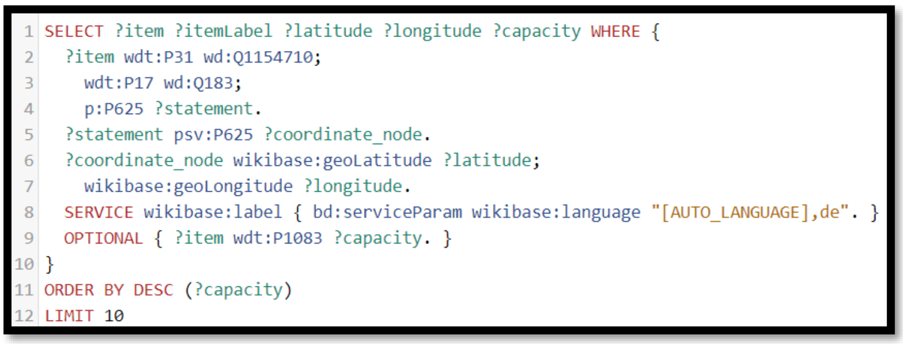


```
SELECT ?item ?itemLabel ?latitude ?longitude WHERE {
  ?item wdt:P31 wd:Q3914.
  ?item wdt:P17 wd:Q183.
  ?item p:P625 ?statement .
  ?statement psv:P625 ?coordinate_node .
  ?coordinate_node wikibase:geoLatitude ?latitude .
  ?coordinate_node wikibase:geoLongitude ?longitude .
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "[AUTO_LANGUAGE],de".
  }
}
```

Figure 8: SPARQL request to get all schools in Germany

IV. Example 3 : 10 football stadiums with the highest capacity

The example in figure 9 gives the 10 biggest football stadiums in Germany in terms of capacity.



```
SELECT ?item ?itemLabel ?latitude ?longitude ?capacity WHERE {
  ?item wdt:P31 wd:Q1154710;
    wdt:P17 wd:Q183;
    p:P625 ?statement.
  ?statement psv:P625 ?coordinate_node.
  ?coordinate_node wikibase:geoLatitude ?latitude;
    wikibase:geoLongitude ?longitude.
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],de". }
  OPTIONAL { ?item wdt:P1083 ?capacity. }
}
ORDER BY DESC (?capacity)
LIMIT 10
```

Figure 9: SPARQL request to retrieve the 10 biggest football stadiums in Germany

Now that the two methods to retrieve geographic data have been discussed, the JavaScript plugin that is used to display a map will be presented.

4. Task 3: Generate thematic maps with Leaflet

Leaflet[7] is an open-source JavaScript library that allows you to display maps on web pages.

I. Layers

There are different possible views (streets, grayscale, landforms, satellite view, etc.) as shown in figures 10 and 11.

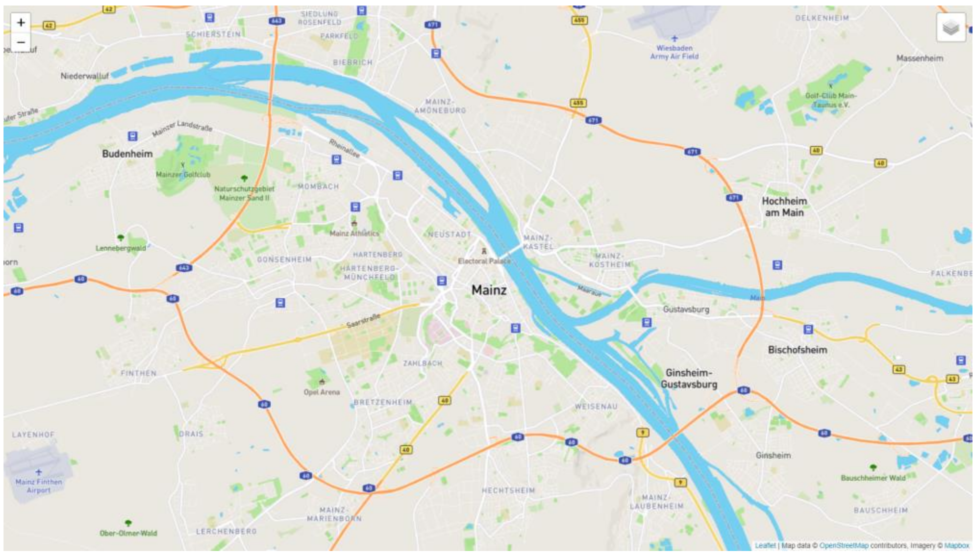


Figure 10: Streets map

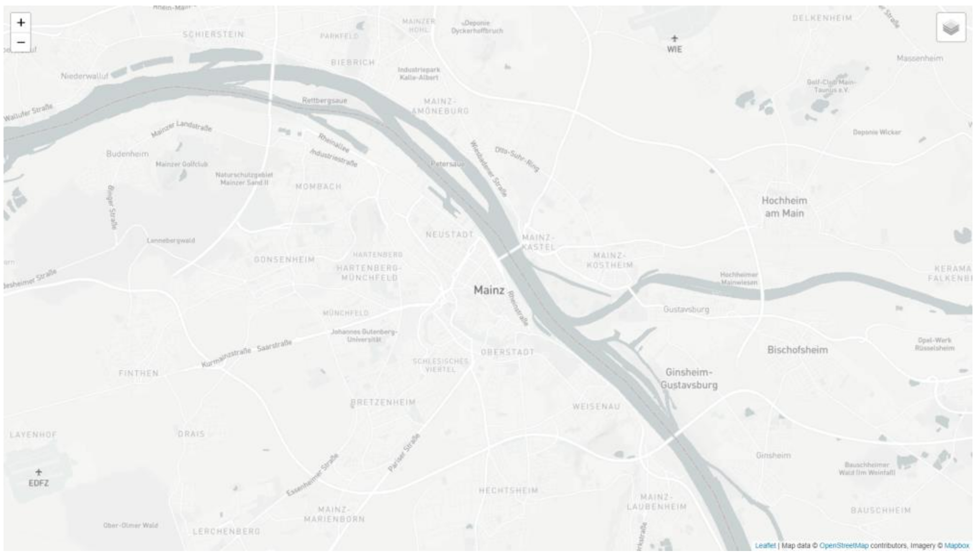


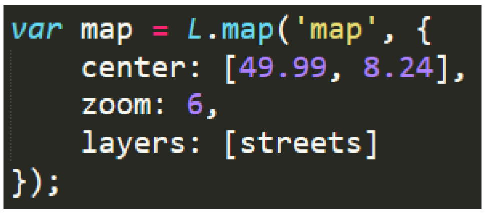
Figure 11: Grayscale map

The figure 12 shows a sample of layers definition used in Leaflet. The variable L represents the variable used by Leaflet to configure the map.

```
var grayscale   = L.tileLayer(mbUrl, {id: 'mapbox/light-v9', tileSize: 512, zoomOffset: -1, attribution: mbAttr}),
streets   = L.tileLayer(mbUrl, {id: 'mapbox/streets-v11', tileSize: 512, zoomOffset: -1, attribution: mbAttr});
```

Figure 12: Definition of layers in Javascript

It is also possible to configure basic map settings, such as the coordinates of the center of the map, the zoom level, the layers that will be displayed when the page loads, etc.



```
var map = L.map('map', {
    center: [49.99, 8.24],
    zoom: 6,
    layers: [streets]
});
```

Figure 13: Map basic settings

II. Additional information on the map

It is possible to add points, lines and shapes and add display options. For example, it is possible to create groups of points and to display them or not on the map. The Leaflet documentation[8] gives many possibilities to customize the map. It is possible to display custom icons on the map[9], to show videos on the map – for example, for meteorological purposes–, to optimize user experience for mobile users, etc.

III. Displaying maps from Wikidata requests or from GeoJSON files

In addition to what was done previously, we will create a map of points to illustrate the requests shown in 3.[10] : the 20 largest cities in Germany, all schools in Germany and the 10 biggest football stadiums in Germany. We will also create a map of universities from the GeoJSON file provided by BKG.

Once these requests are executed, the Linked Data Enrichment page will show the map with all the results, the number of results and a table with all the results. These maps and results are shown in figures 14, 15, 16, 17.

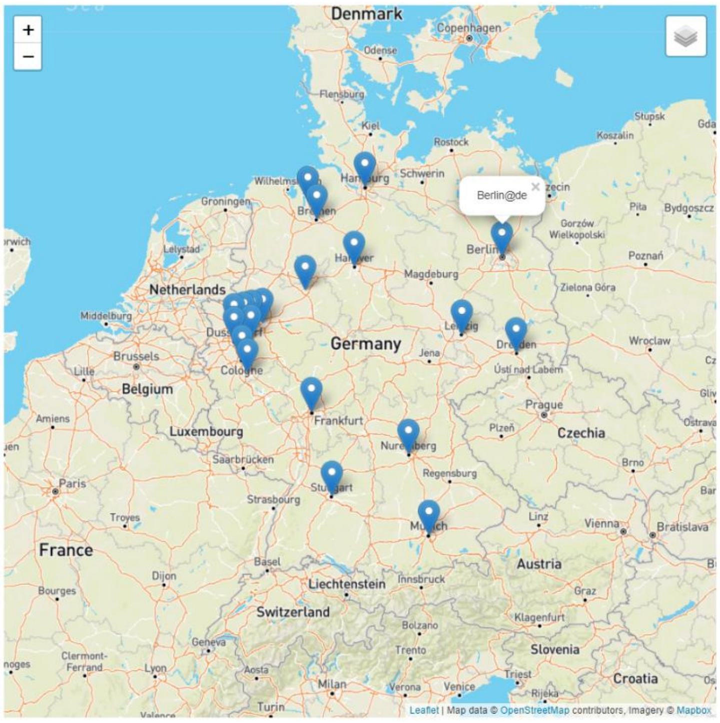
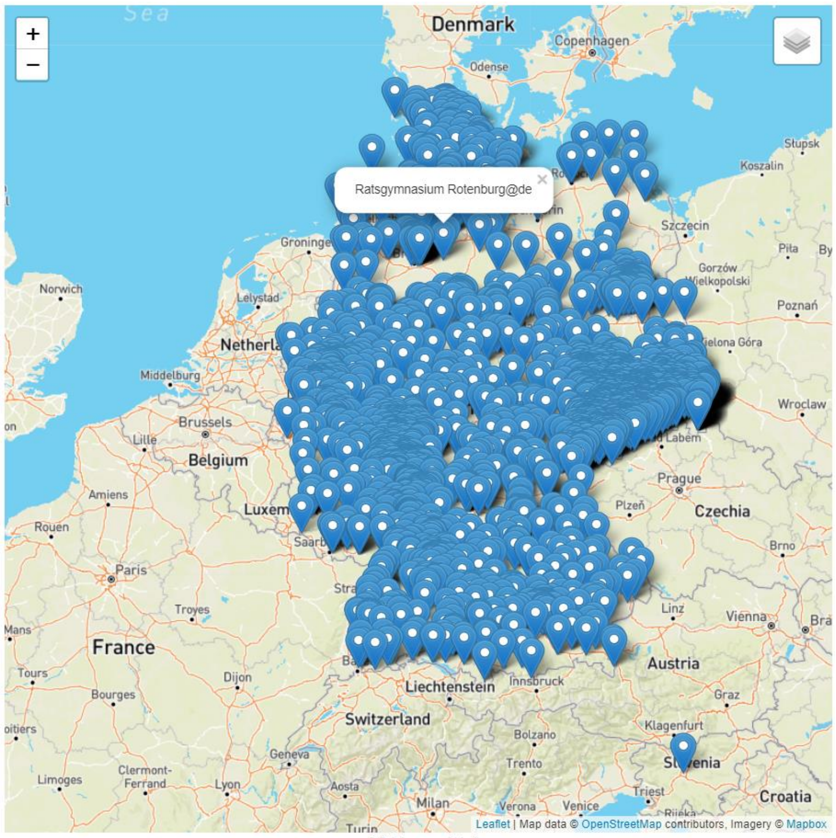
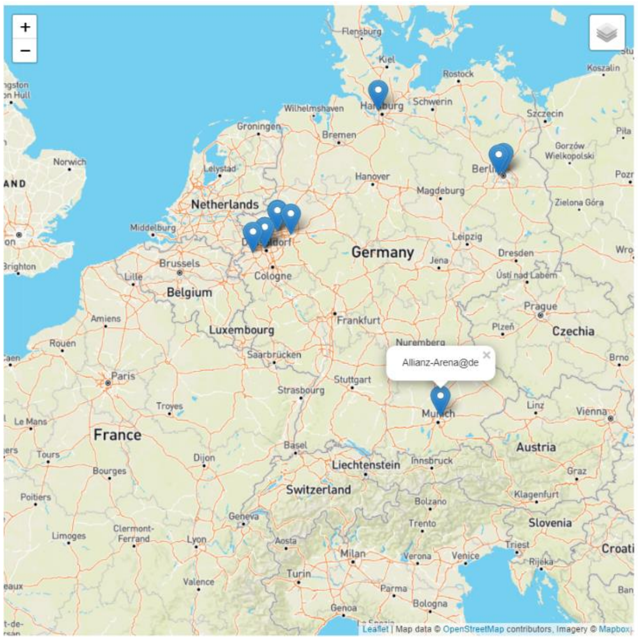


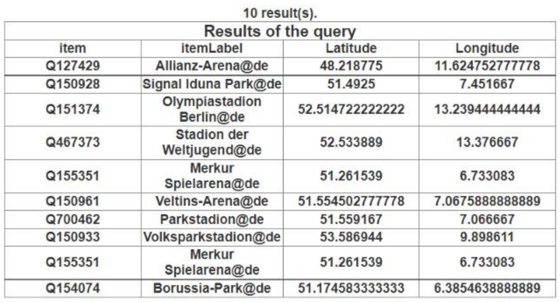
Figure 14: 20 biggest cities in Germany


Ratsgymnasium Rotenburg@de
Leaflet | Map data © OpenStreetMap contributors, Imagery © Mapbox

3044 result(s).

Figure 15: All schools in Germany and number of results


Allianz-Arena@de
Leaflet | Map data © OpenStreetMap contributors, Imagery © Mapbox


10 result(s).

Results of the query

item	itemLabel	Latitude	Longitude
Q127429	Allianz-Arena@de	48.218775	11.624752777778
Q150928	Signal Iduna Park@de	51.4925	7.451667
Q151374	Olympiastadion Berlin@de	52.514722222222	13.239444444444
Q467373	Stadion der Weltjugend@de	52.533889	13.376667
Q155351	Merkur Spielarena@de	51.261539	6.733083
Q150961	Veltins-Arena@de	51.554502777778	7.0675888888889
Q700462	Parkstadion@de	51.559167	7.066667
Q150933	Volksparkstadion@de	53.586944	9.898611
Q155351	Merkur Spielarena@de	51.261539	6.733083
Q154074	Borussia-Park@de	51.174583333333	6.3854638888889

Figure 16: 10 biggest football stadiums in Germany, number of results and list of these results

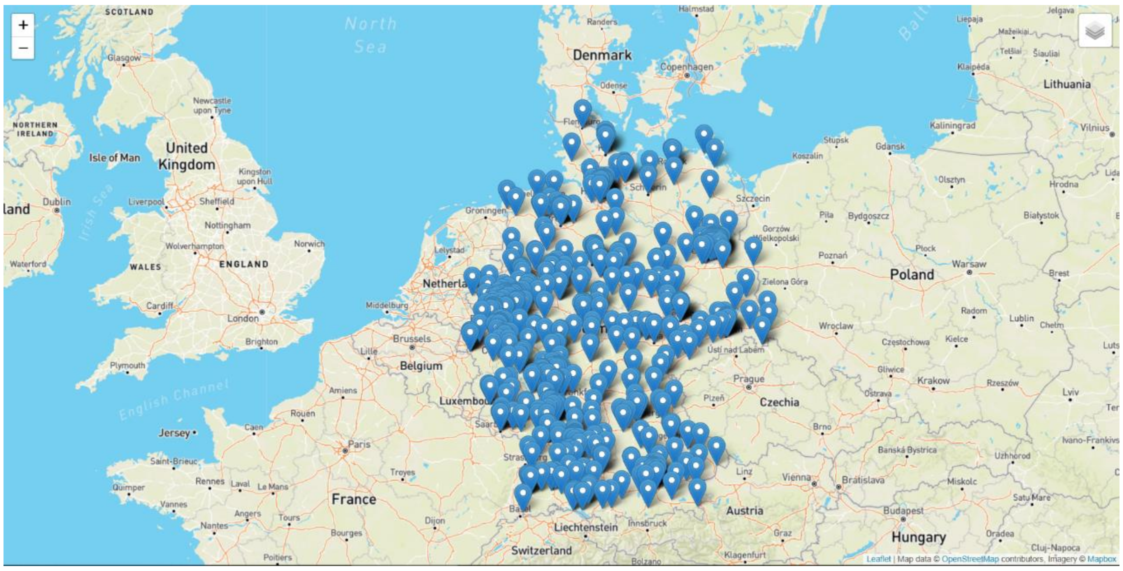


Figure 17: Universities in Germany from the GeoJSON file provided by BKG

IV. Javascript code

To add a point on the map, you have to know its latitude, longitude and its name. Then you can add it to a layer or not. If the point is not added to a layer, it will not be possible to hide it with a checkbox.

```
// To add a point :
L.marker([latitude, longitude]).bindPopup(nameOfThePoint).addTo(nameOfTheLayer);
```

Figure 18: Adding a point to the map

To retrieve data from a SPARQL request, we will need to loop this functionality to add the number of points corresponding to the number of results.

To display data from a GeoJSON file, there are two possibilities : you can use a basic Leaflet functionality that allows to quickly add a GeoJSON variable to the map (Figure 19), or you can create your own method. However, if you use the basic functionality implemented in Leaflet, the map will display the point but you will not be able to click on the point and show its name.

```
L.geoJSON(hs).addTo(universities);
```

Figure 19: Adding a GeoJSON variable to a layer

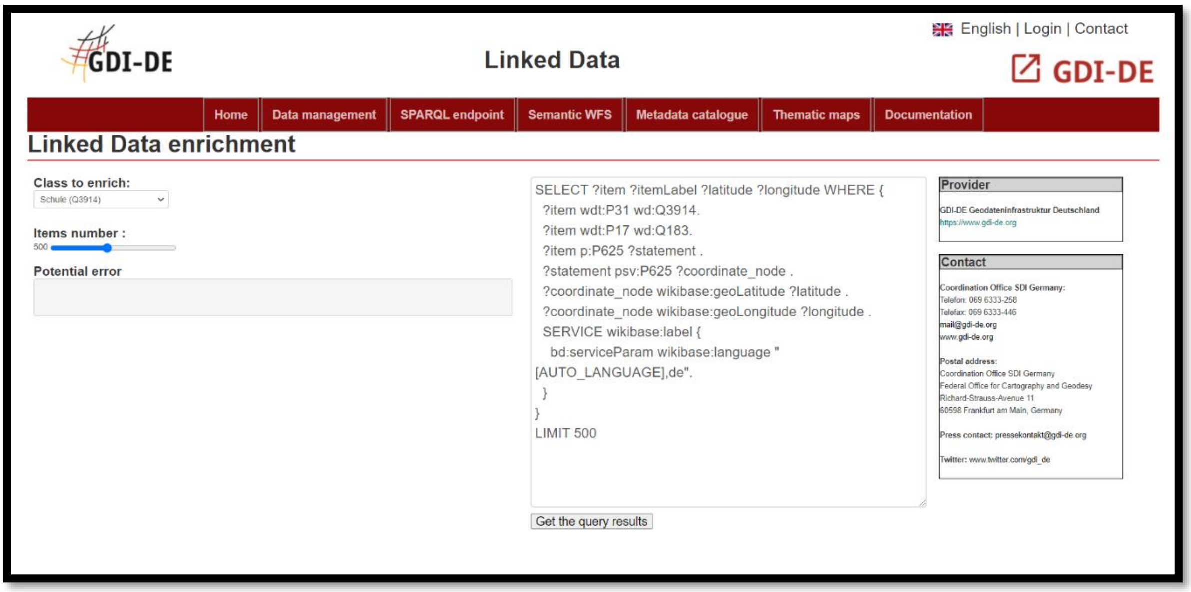
We now have an application that can learn and incorporate data from a user or from linked data, and we are able of displaying these data on thematic maps.

5. Task 4: Creation of the Linked Data enrichment page

For the creation of this page, I did a pair work with Pierre Mazurek, another student who also worked on the project. He took care of the queries, the storage of data and the style of the page while I worked on processing the data and displaying the map.

I. Overview

The figure 20 shows the page at the time of writing this report.



Linked Data enrichment

Class to enrich: Schule (Q3914)

Items number : 500

Potential error

```
SELECT ?item ?itemLabel ?latitude ?longitude WHERE {
  ?item wdt:P31 wd:Q3914.
  ?item wdt:P17 wd:Q183.
  ?item p:P625 ?statement .
  ?statement psv:P625 ?coordinate_node .
  ?coordinate_node wikibase:geoLatitude ?latitude .
  ?coordinate_node wikibase:geoLongitude ?longitude .
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "
[AUTO_LANGUAGE],de".
  }
}
LIMIT 500
```

Get the query results

Provider

GDI-DE Geodateninfrastruktur Deutschland
https://www.gdi-de.org

Contact

Coordination Office SDI Germany:
Telefon: 069 6333-258
Telefax: 069 6333-446
mail@gdi-de.org
www.gdi-de.org

Postal address:
Coordination Office SDI Germany
Federal Office for Cartography and Geodesy
Richard-Strauss-Avenue 11
60598 Frankfurt am Main, Germany

Press contact: pressekontakt@gdi-de.org

Twitter: www.twitter.com/gdi_de

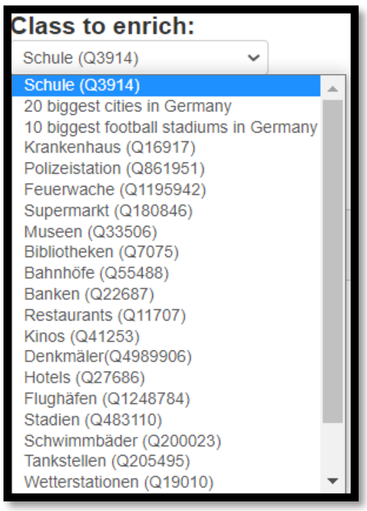
Figure 20: Linked Data enrichment page

On the left of the page, there is at first a list of around twenty query examples, as shows in figure 19. Then, we have a slider bar which will allows you to choose the number of items the request have to select. Under this slider, a text area which will displays the error code provided by Wikidata if the request is not correctly written.

On the right of the page, a text box with the query is displayed. This query changes depending on the one selected in the list on the left. It also changes depending on the value of the slider.

It is also possible to enter a request manually.

A new page will then open. This page will show a map, the number of results and the list of those results if the query works. If it does not work, then an error code will be displayed in the box provided.


Class to enrich:

Schule (Q3914)

Schule (Q3914)
20 biggest cities in Germany
10 biggest football stadiums in Germany
Krankenhaus (Q16917)
Polizeistation (Q861951)
Feuerwache (Q1195942)
Supermarkt (Q180846)
Museen (Q33506)
Bibliotheken (Q7075)
Bahnhöfe (Q55488)
Banken (Q22687)
Restaurants (Q11707)
Kinos (Q41253)
Denkmäler(Q4989906)
Hotels (Q27686)
Flughäfen (Q1248784)
Stadien (Q483110)
Schwimmbäder (Q200023)
Tankstellen (Q205495)
Wetterstationen (Q19010)

Figure 21: Query examples

II. Javascript code

The first step is to retrieve the data of the executed query which is stored in the Java controller file. The "/*<![CDATA[*/" and "/*]]>*/" tags are used to transmit the table of values to the HTML / Javascript file. It is then important to retrieve the number of results to be able to display it to the user. This is what is done in the size() function.

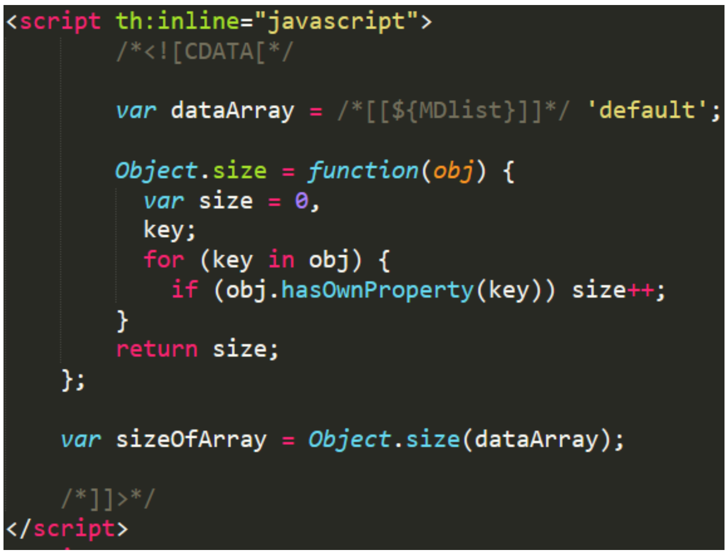
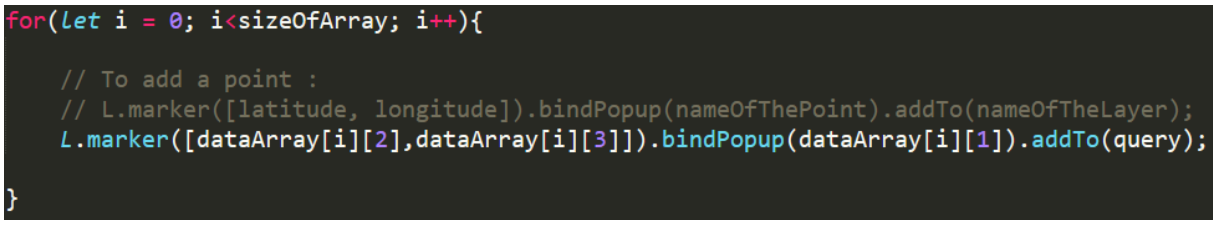

```
<script th:inline="javascript">
        /*<![CDATA[*/

        var dataArray = /*[[${MDlist}]]*/ 'default';

        Object.size = function(obj) {
          var size = 0,
          key;
          for (key in obj) {
            if (obj.hasOwnProperty(key)) size++;
          }
          return size;
    };

    var sizeOfArray = Object.size(dataArray);

    /*]]>*/
</script>
```

Figure 22: Javascript function to retrieve results and the number of results

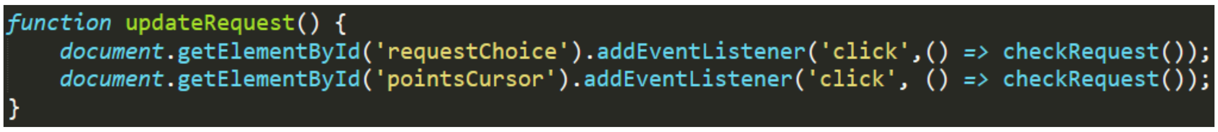
The data table is made up of four columns: object, object name, latitude and longitude. We will use 3 of these 4 columns to display the points on the map. For each element of this table, we will add a point and its coordinates to the map.



```javascript
for(let i = 0; i<sizeOfArray; i++){

    // To add a point :
    // L.marker([latitude, longitude]).bindPopup(nameOfThePoint).addTo(nameOfTheLayer);
    L.marker([dataArray[i][2],dataArray[i][3]]).bindPopup(dataArray[i][1]).addTo(query);

}
```

Figure 23: Javascript function to add every points on the map

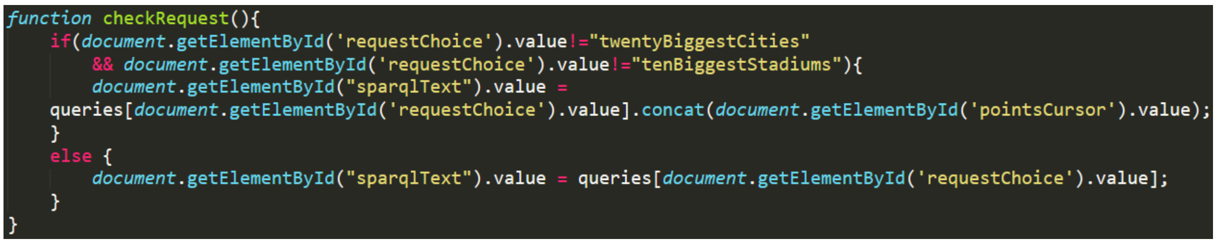
We can now focus on the functionalities linked to the display of the query. Two update functions are required: an update when an example is selected, and another when the slider is moved.



```javascript
function updateRequest() {
    document.getElementById('requestChoice').addEventListener('click',() => checkRequest());
    document.getElementById('pointsCursor').addEventListener('click', () => checkRequest());
}
```

Figure 24: updateRequest() function which updates the request with new values

The checkRequest() function is called whenever a click is detected on the list of queries, or on the slider bar and update the query with the new values. However, one exception should not be forgotten: some "special" queries, such as those showing the 20 largest cities or the 10 biggest stadiums, should not be updated during the change of the slider value. This exception is then added to the code shown in figure 25.



```javascript
function checkRequest(){
    if(document.getElementById('requestChoice').value!="twentyBiggestCities"
        && document.getElementById('requestChoice').value!="tenBiggestStadiums"){
        document.getElementById("sparqlText").value =
    queries[document.getElementById('requestChoice').value].concat(document.getElementById('pointsCursor').value);
    }
    else {
        document.getElementById("sparqlText").value = queries[document.getElementById('requestChoice').value];
    }
}
```

Figure 25: checkRequest() function which checks if the request should be updated or not

6. Problems encountered and potential next steps of development

The functionalities of the application which were presented have been completed for most of them but certain functionalities still must be developed or improved.

Problems encountered and unfinished tasks

The code for processing shapefiles is not implemented yet. As shapefiles do not always contain the same types of files, it has been difficult for me to understand their overall behavior and develop working code that can handle any shapefile.

I also didn't have time to develop an option that allowed the user to choose the icon that would be used on the map instead of the base icon. A list of suitable icons could be available depending on the fields (education, medical environment, sports, etc.).

I spent a lot of time trying to understand the display of some in-app elements (especially maps), since CSS files of several thousand lines were already in place, and some elements of the page were hiding some. others.

Finally, two submenus of the "Thematic Maps" menu have not been completed. The spatio-temporal data example page has not yet been integrated, and the page with the list of existing thematic maps has not yet been developed.

Next steps?

If my internship was not over, I would obviously have focused on the last missing pages of the final prototype and the design of the triplestore, but I also had other ideas to improve the user experience and the addition of features:

-the addition of a CSV or JSON file converter to GeoJSON, since the file formats that data on Wikidata can be downloaded are CSV or JSON. There is no direct functionality to download a GeoJSON file.

-the addition of a feature for the calculation of distance, which would make it possible to find the doctor or the closest hospital to a school, a university, ...

-optimization of the map display for mobile users.

References

[1] : Wikidata, https://www.wikidata.org/

[2] : i3Mainz, https://i3mainz.hs-mainz.de/

[3] : BKG, https://www.bkg.bund.de/DE/Home/home.html

[4] : Spring, https://spring.io/

[5] : Wikipedia triplet example,
https://en.wikipedia.org/wiki/Resource_Description_Framework#Overview

[6] : Wikidata query service, https://query.wikidata.org/

[7] : Leaflet, https://leafletjs.com/

[8] : Leaflet documentation, https://leafletjs.com/reference-1.7.1.html

[9] : Leaflet custom icons tutorial, https://leafletjs.com/examples/custom-icons/