Semantic Web Project

Group Members

Aninda Maulik

Dharmalingam Dhayananth

Academic Supervisor

Antoine Zimmermann

Mines Saint Etienne

Université Jean Monnet

Acknowledgement

We would like to thank Prof. Antoine Zimmermann who guided throughout this project in order to complete it successfully.

Aninda Maulik

Dharmalingam Dhayananth

Abstract

In addition to classical web technologies, semantic web technology provides additional capability to implement “web of things” and “web of linked data”, that enables machines and human to utilize the web. The ultimate goal is to enable machines to do more powerful task and to develop systems that can support interactions over the network.

This vision can be achieved by integrating different technologies of linked data such as RDF, SPARQL, OWL and SKOS. (W3C Org, 2014)

In this project we have demonstrated semantic web technologies by integrating them with our program. This report explains detailed information about the architecture and the functionalities of the applications and how the semantic web technology concepts have been achieved. We followed guidelines and good practices that have been addressed in the W3C papers and guided by our professor.

The project has been initiated with finding proper dataset from different data sources (Heterogenous data sources) and we have put a lot of effort to assess the gathered data. Thereafter, additional efforts were put to cleanse and validate the data.

This resultant dataset was then pushed to the chosen triple store during the boot up time of the application.

In this report, the ontology design and the project architecture has been explained with details. Then the stack of the project has been discussed. Following this, the implementation is explored with some screenshots of the actual code. Finally, we have addressed the references.

Table of Contents

Introduction

The application can give information to the travellers who wish to travel around France. Travellers can search different cities based on interest, the application will display available train stations, bike stations, and hospitals. The train station data contains detailed information such as timetable of a particular station, station location, and direction. The timetable data is retrieved in real time using SNCF API portal.

Also, bike stations data show real time availability of bikes in available bike stations and also the location of it. So the users will now know which bike station they should reach in order to grab a bike or to drop a bike. Because to leave a bike, it is necessary to know the empty slot (rack) in the station.

The hospital data is shown based on the searched city with some useful information such as specialty division of a hospital, location on map, contact details, and corresponding head hospital.

All this information is shown in the map with associated map icons. The map will initially show the user location based on location service of client device. If no location service is available in client device, then the local service providers' information will be utilized to predict the location.

Additionally, the real time weather information will be shown to the travellers which can help to pre plan their trip.

Ontology diagram

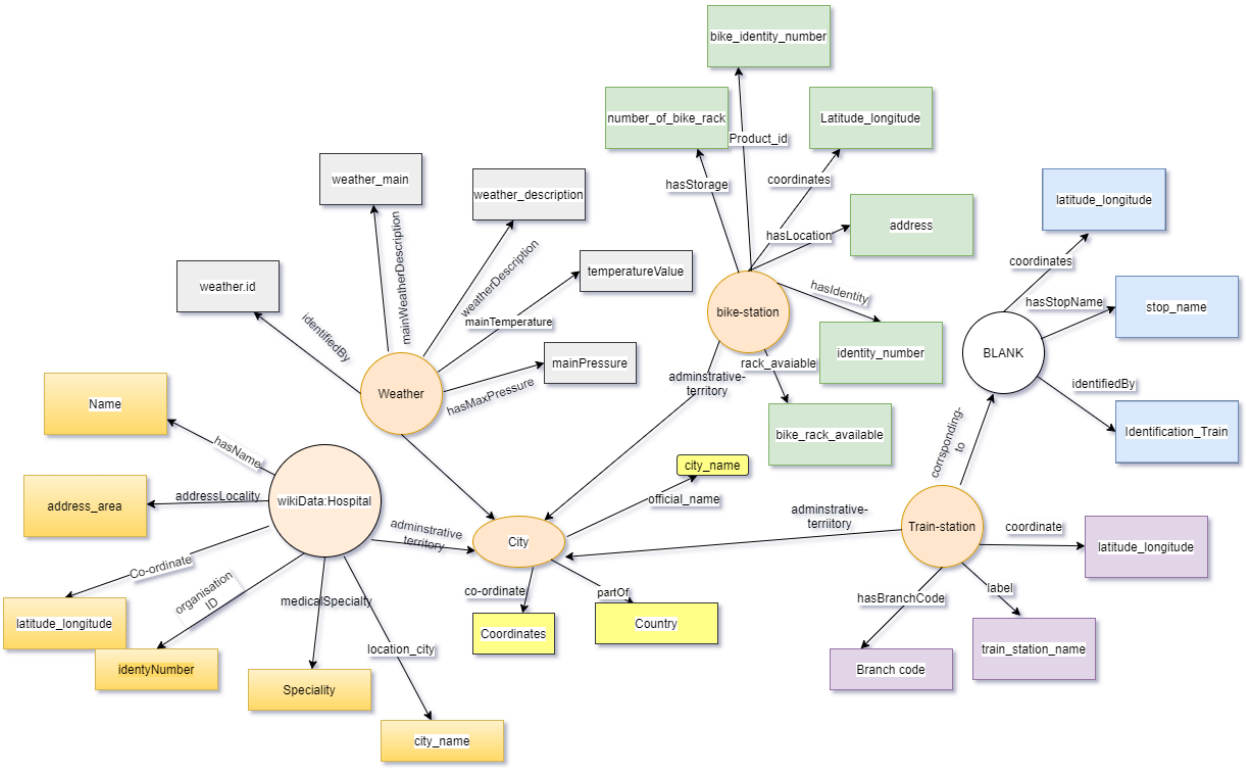


Figure 1: Ontology diagram

Above ontology diagram explains the knowledge graph of the project. There were **5 main entities** which are city, bike stations, railway stations, hospitals and weather. Also, we have timetable entity of train stations that is marked as **blank-node**, since it has been prepared as **blank-node** in **RDF graph (Blank nodes does not contain any IRI).** These entities have been shown in elliptical shape in the diagram. All entities were mainly connected to the city. This shows that the linked data is prepared mainly based on cities. The properties of each entity have been shown as edge. The rectangular representations are just **literals** in RDF graph.

Each edge shows relation between entities and attributes, and also, with other entitles. These are known as **predicates** in RDF graph.

Stack

Figure 2: Stack of the project

Front-End technologies

- Angular 10
    - We are using Angular framework to develop our client-side web interface. This framework fully supports TypeScript.
- Bootstrap
    - Also we use Bootstrap for designing our interface with good looking components.
- Webpack
    - Finally we use webpack to bundle our JavaScript application. This tool will do the job of “typings”, “uglifying” and “minifying” process.
- JSON-LD
    - JSON-LD is a lightweight library for creating Linked Data Format in HTML websites. It is completely based on most popular content format which is JSON. JSON-LD empowers people to create Linked-Data on web. This is also used by search engines for the purpose of rich content extraction and search engine optimization. (JSON-LD org, 2018)

Web server technologies

- Java –Spring Boot
    - We use Spring boot for developing server application with Rest API. This is a framework that developed using Java programming language and this provides pre-configured REST API implementations.

- SWAGGER
    - We use Swagger for API Documentation (Accessible via http://localhost:8080/swagger-ui.html)
- Apache-Jena
    - Apache Jena is a free open source Java based library that gives capabilities to developed semantic web and linked-data applications. (Apache jena org, 2020)

Triple store

- Graph-DB
    - Ontotex Graph DB is a most efficient graph database (triple store) with RDF and SPARQL support. It has integrated SPARQL endpoint that enables users to query RDF data using SPARQL query. GraphDB allows to store RDF data in different format such as turtle, N-Triples, and N-Quads. (GraphDB org, 2020)

Portege

Protege is a free open source ontology editor tool that allows to create OWL ontology. We can create simple and complete ontology base application using Protege. It provides better graphical user interface to simplifies the ontology creation and edition. This tool has been used to create our own ontology graph with Web Ontology Language (OWL). (Protege team, 2020)

Github

GitHub is a version control system that has been used to manage the application code. GitHub keeps the history of different version of code files and keeps track of modifications. Developers can collaborate and contribute to the development and can easily integrate their work in a centralised repository.
(BROWN, 2019)

For this project, there are three repositories were designed.

1. Server-Application : This repository contains the Java code for “Web-Server” application.
2. Front-app: This repository is responsible for front-end angular application code.

1. **Server-application**

Server application is a java application that is responsible for handle client requests and responses that are been send from user interface (Front-ap) and responsible to store and retrieve data from triple store. There are two main tasks of this application. First task is to prepare RDF data and store it to the triple store, and the second task is to retrieve RDF data from triple store based on the search criteria and return it to the front app. The functionalities were further discussed in implementation section of this report.

2. **Front-App**

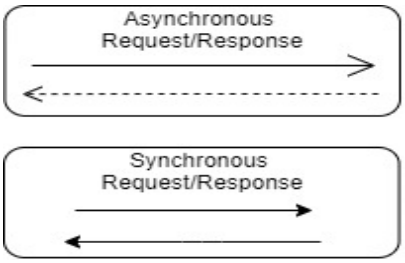
Front-app is a client-side graphical user interface integrated application that has been developed using Angular framework. Angular is a JavaScript-based library which allows developing front side (SPA) Single Page Application. This framework fully supports TypeScript. Also, we use Bootstrap for designing our interface with good looking components. Finally, we use webpack to bundle our JavaScript application. This tool will do the job of "uglifying" of "minifying" process and build our front-app.

HTTP, WS (WebSocket)

The http standard protocol is used to communicate to the server application from "front-app". In addition to http, the WebSocket API is also used for extended use cases. This WS protocol helps to enable and open two-way interactive communication session instead of stateless communication (HTTP are stateless communication). Using this communication protocol, it is possible to send response from server to client at any time asynchronously. (MDN contributors, 2020)

Design Diagrams

Note: Request response arrows



Sequence Diagram

Search city function

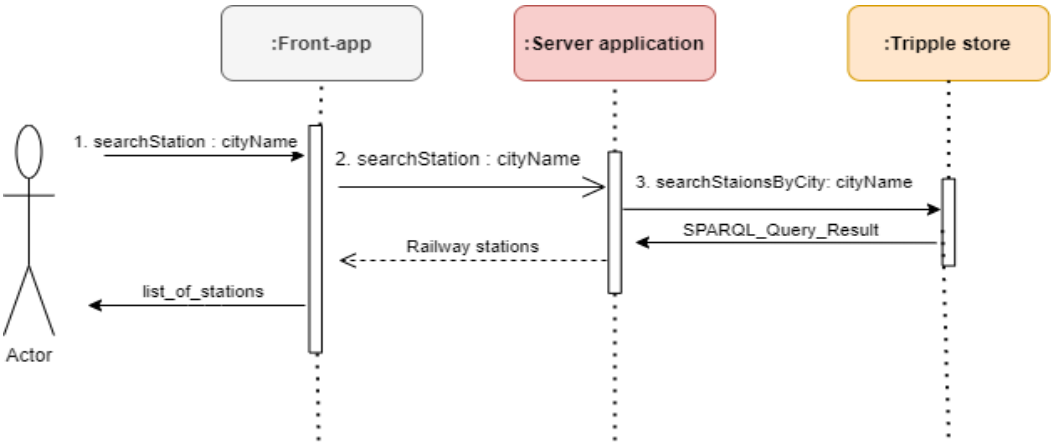


Figure 3: Railway station search sequence diagram

Above image shows the functional call and execution flow of city search process. This flow of process gets started when user enters a city name in the search bar of front-app and clicks the search button. The city name query parameter will be passed to the server application with asynchronous request (Asynchronous requests are shown in open arrowhead- [2]). The web server will get the request with city name and proceed to query GRAPHDB with SPARQL endpoint ([3]). Then the GraphDB will return the results to the server application. Server application will process further to create appropriate object list from RDF graph and return the result to the front app which is then shown in the graphical interface.

The above sequence diagram shows the search function of railway station entity of a specific city. This same sequence of functions will be applied to other entity search such as Bike stations, and Hospitals.

Railway station timetable display function

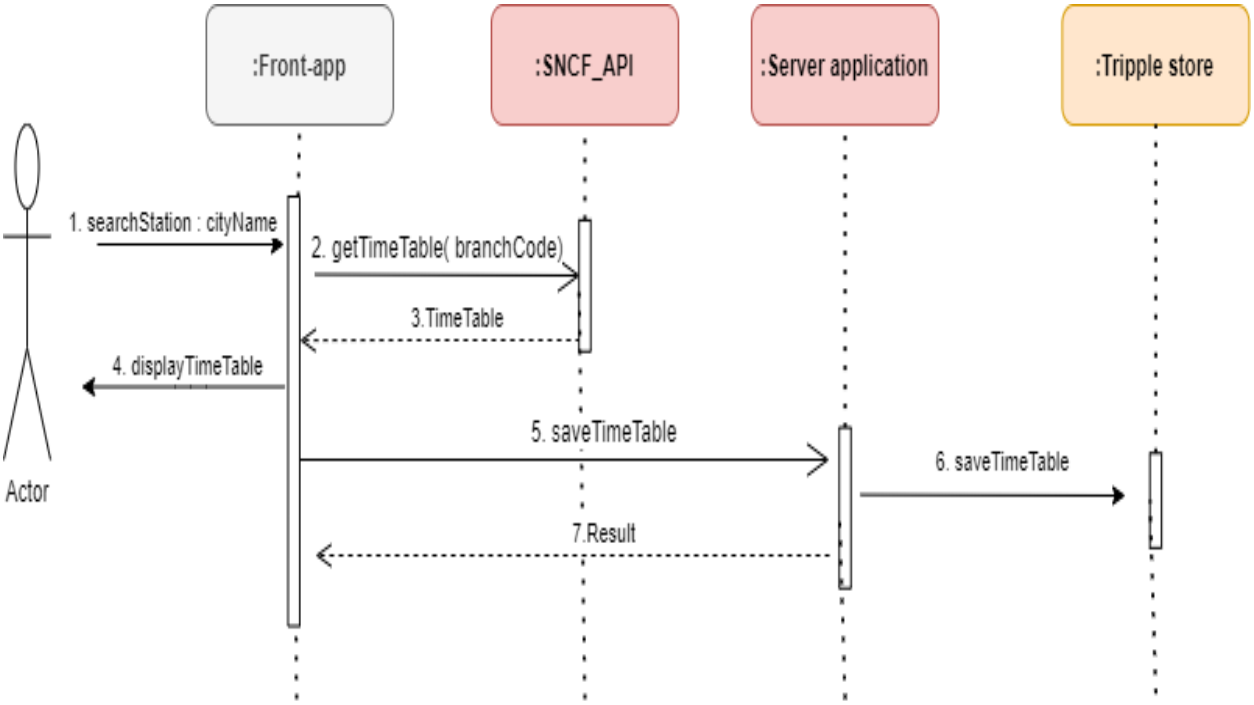


Figure 4: Sequence Diagram for real time timetable request

Above figure shows the shows the sequence diagram of Timetable API service. When user click on particular railway station on table, the "showTimeTable" function will be triggered [1]. This function will issue an async get request to SNCF API portal with the branch code of selected station [2]. The SNCF service will return the timetable of that particular stations (not for all stations) [3]. The retrieved result will be shown to the user. Afterwards, the results will be sent to the server application. This data will be stored to the GraphDB as RDF Blank-node [6]. Then the server application will respond to the front app with status code 200, if everything is successful [7].

Get Bike Station Real time data function (Rennes city)

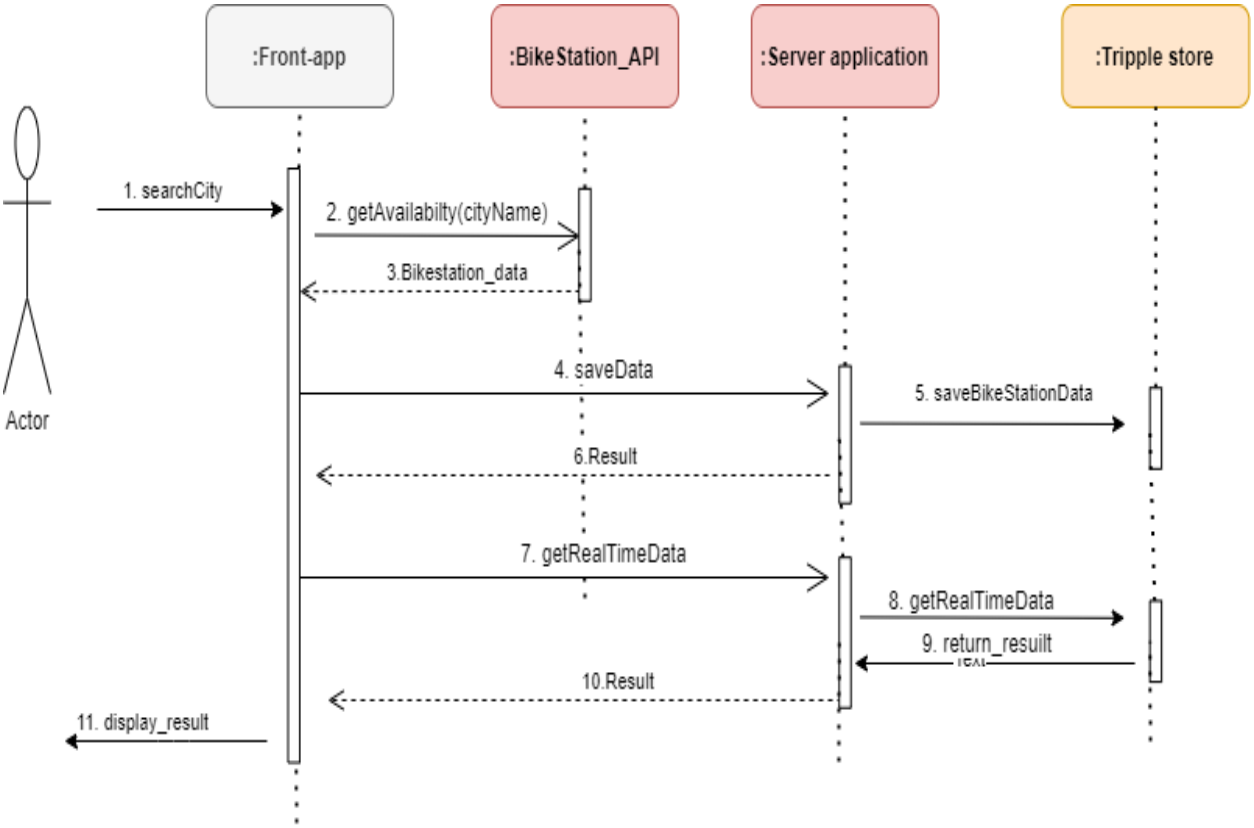


Figure 5: Sequence Diagram for realtime bike station data

Above figure shows the shows the sequence diagram of Rennes bike station API service. When user search for "Rennes" city, the "getAvailability" function will be triggered [2]. This function will issue an async GET request to Bike station API portal [2]. The bike station API will return the real time bike station data such as availability of rented bikes of different bike stations around "Rennes" city [3]. The retrieved results will be sent to the server application [4]. This data will be stored to the GraphDB as RDF Blank-node [5]. Then the server application will respond to the front app with status code 200, if everything is successful [6]. Then again front app will issue an async GET request to server application [7]. The server application will query GraphDB to get real time data of "Rennes" city and return the result. The request will be processed by server application to create appropriate list of objects. Then this list will be returned to front app [10]. Front app will display the result in table [11].

Get weather data function

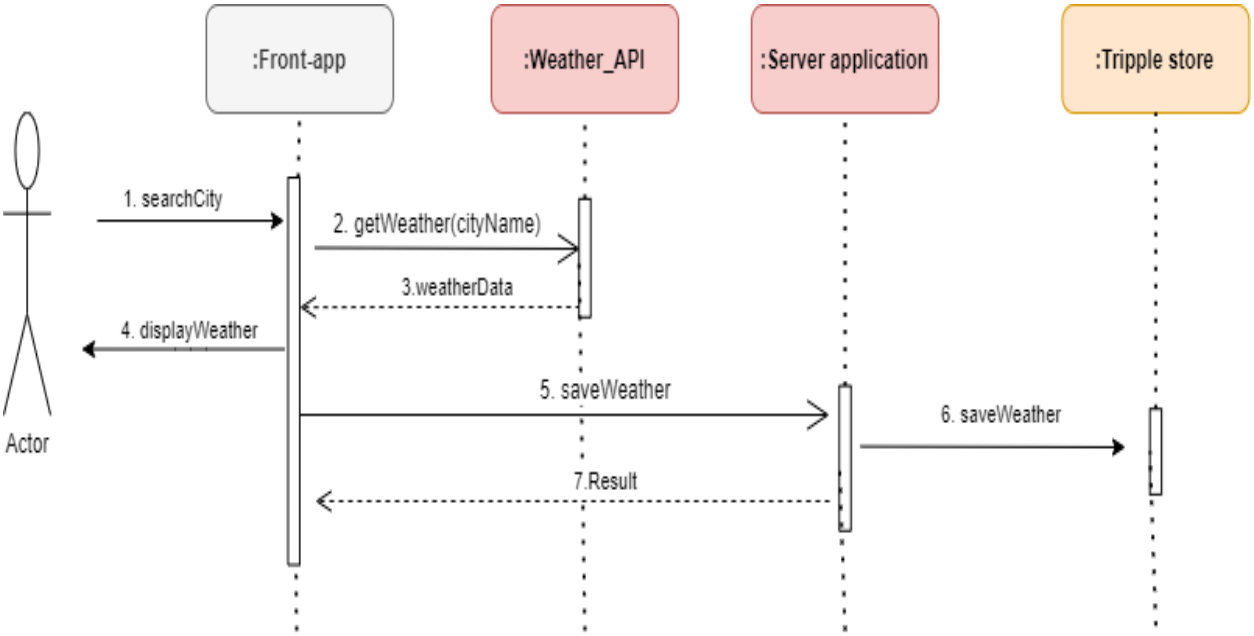


Figure 6: Sequence Diagram for realtime weather data

Above figure shows the shows the sequence diagram of Weather API service. When user searches for particular city, the "getWeather" function will be triggered [2]. This function will issue an async get request to Weather API portal with the city coordinates and city name [2]. The Weather API will return the real time weather forecast data of that particular city [3]. The retrieved result will be shown to the user. Afterwards, the results will be sent to the server application. This data will be stored to the GraphDB as RDF Blank-node [6]. Then the server application will respond to the front app with status code 200, if everything is successful [7].

Create new city RDF data

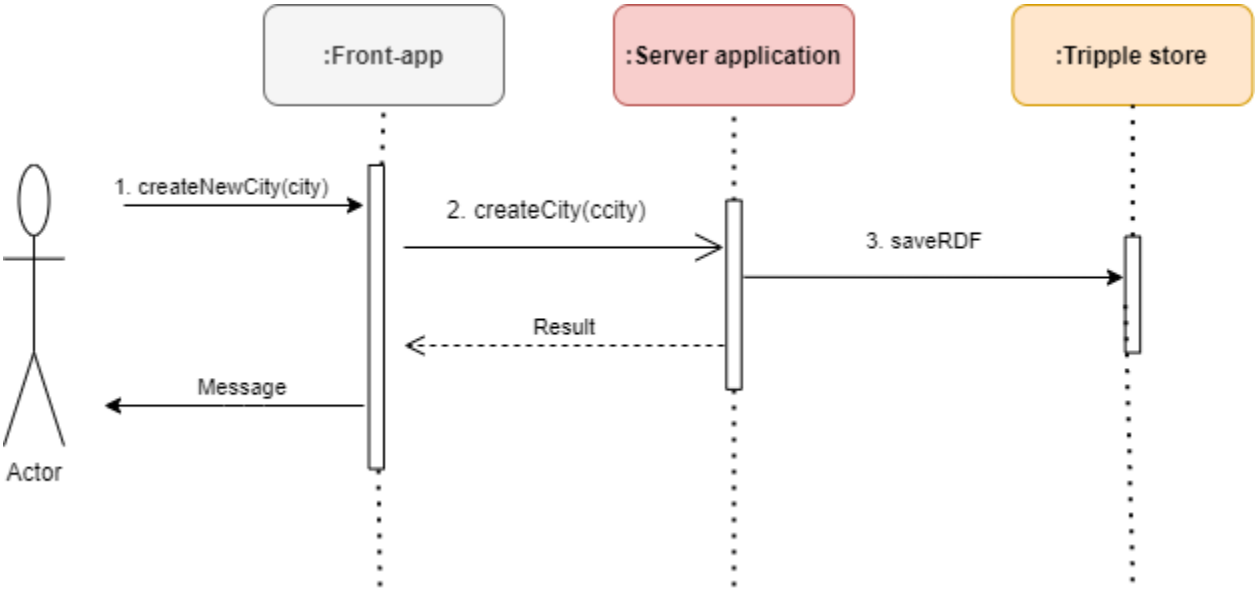


Figure 7:Create new RDF data from Graphical user interface

User would be able to create new City data using Graphical user interface. User will be required to insert different data based on the given text fields such as city name, city IRI and city coordination etc. This data will be posted to the server application [2]. Server application will then prepare RDF graph, based on the given data and store it to the GraphDB [3]. If everything is successful, then the server application will respond with 200 status code and the success message will be shown to the user.

Implementation Overview

Package overview

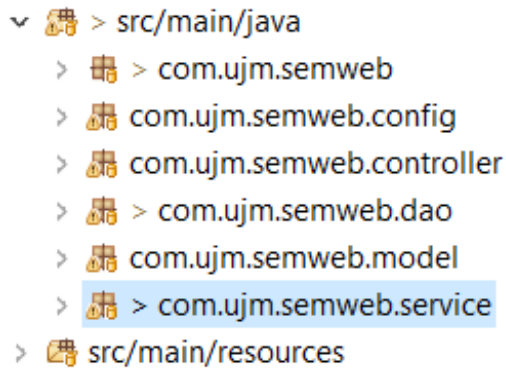


Figure 8: Spring boot package overview

The Spring boot application contains different packages based on the functionalities. The "*com.ujm.semweb.config*" package contains Swagger configuration class, and DumpData class (See below for details of DumpData class). API endpoints are defined in controller class. These controller classes were dropped inside "*com.ujm.semweb.controller*" package. We implemented different controllers for different entities based on "SOLID" architecture principle. "*com.ujm.semweb.dao*" package contains components that are responsible for interact with triple store (Data Access Objects). The model entity classes were created inside "*com.ujm.semweb.model*" package. Also each entity contains corresponding service classes that exists inside "*com.ujm.semweb.service*" package.

Functional Overview

```
private  String GRAPH_REPO_QUERY =
        "http://localhost:7200/repositories/SemWeb";
private String GRAPH_REPO_UPDATE =
        "http://localhost:7200/repositories/SemWeb/statements" ;
@PostConstruct
public void dumpData() {
    LOG.info("RDF DATA STORING PROCESS INITIATED");
    dumpTrainStationGraphV2();
    dumpCitiesGraphV2();
    dumpHospitalGraph();
    dumpBiksStationGraphRennes();
    dumpBiksStationGraphLyon();
    LOG.info("RDF DATA STORING PROCESS COMPLETED");
}
```

Figure 9: Dump-data function

This "dumpData" function is configured to get triggered when the web application gets started (@PostConstruct annotation will do this in Spring Boot application). This function contains sub-functions to store city data, train station data, hospital data and bike station data to triple store. The data files (CSV files) are available in "resource" folder. These functions will read the data files and prepare RDF graph and store it to the triple store with the given endpoint (at top we define global variables for GraphDB end points).

This class is present in "*com.ujm.semweb.config*" package.

Get all stations by city name

```
public List<RailwayStation> getAllStationByCityName(String cityName) {
    List<RailwayStation> railwayStations=new ArrayList<>();
    String graphQuery=""
            + "PREFIX a: <https://www.wikidata.org/wiki/Property:> "
            + "PREFIX wd:<https://www.wikidata.org/wiki/> "
            + "PREFIX schema:<http://www.w3.org/2000/01/rdf-schema#> "
            + "PREFIX schemaOrg:<https://schema.org/> "
            + "SELECT DISTINCT ?comment ?stationComment ?city  ?station ?stationLabel ?stationCoordination ?branchCode {"
            + "         ?city  a:P31 wd:Q484170 ; "
            + "             schema:label \""+cityName.toUpperCase()+"\";"
            + "             schema:comment ?comment ."
            + "         ?station  a:P31 schemaOrg:TrainStation ; "
            + "                 a:P625 ?stationCoordination; "
            + "                 schemaOrg:branchCode  ?branchCode; "
            + "                 a:P131  ?city; "
            + "                 schema:comment ?stationComment; "
            + "                 schema:label ?stationLabel . "
            + "}";
     QueryExecution queryExecution = prepaerQueryExecution(graphQuery);
     LOG.info("EXECUTING SPAREQL QUERY");
     try {
     for (ResultSet results = queryExecution.execSelect(); results.hasNext();) {
          QuerySolution qs = results.next();
          RailwayStation station=new RailwayStation();
          station.stationUri=qs.get("?station").toString();
          station.coordination=qs.get("?stationCoordination").toString();
          station.cityUri=qs.get("?city").toString();
          station.cityLabel=cityName;
          station.branchCode=qs.get("?branchCode").toString();
          station.stationLabel=qs.get("?stationLabel").toString();

          station.comment=qs.get("?stationComment").toString();
          station.instanceOf="Q484170";
          railwayStations.add(station);
        }
        queryExecution.close();
        return railwayStations;
```

Figure 10:Get railway stations by city name function

The above image shows the Java function to retrieve all railway stations by city name. When user search a city, the city name will be passed to the web server application and this function will get executed. This function is existing in "*StationDao.java*" Class. This function will prepare a SPARQL query and concatenate the city name. The GraphDB connection object will return all the matching railway station triplets in query results. This query results will be processed to prepare railway station objects and sent back to the front app.

Save train station timetable data

```
try {
for(RailwayStation railwayStation: railwayStations) {
       String stationQid=railwayStation.stationUri;
       //SETTING UP AVAILABILITY BLANK NODE
       timeTableGraph+="<"+stationQid+"> "
       +"<"+model.createProperty(custom_ontology+"hasTimeTable").toString()+"> "
       +" [ <"+a+"> "
       +"<"+model.createProperty(custom_ontology+"TrainStationTimeTable").toString()+"> ; "

       +"<"+model.createProperty(custom_ontology+"hasStopName").toString()+"> "
       +" \""+railwayStation.stopPoint+"\"@en ; "
       //SettingUp-parent_station
       +"<"+model.createProperty(custom_ontology+"definedBy").toString()+"> "
       +" \""+railwayStation.timeTableNetwork+"\"@en ; "
     //SettingUp-trip_id
       +"<"+model.createProperty(custom_ontology+"hasTripId").toString()+"> "
       +" \""+railwayStation.tripId+"\" ; "
       //SettingUp-Direction
       +"<"+model.createProperty(custom_ontology+"hasDirection").toString()+"> "
       +" \""+railwayStation.timeTableDirection+"\" ; "
     //SettingUp-arrival_time
       +"<"+model.createProperty(custom_ontology+"arrivingAt").toString()+"> "
       +" \""+railwayStation.arrivingTime+"\"^^<http://www.w3.org/2001/XMLSchema#dateTime> ; "
       //RecordedAt
       +"<"+model.createProperty(custom_ontology+"recordedAt").toString()+"> "
              +" \""+formatter.format(date).toString()+"\"^^<http://www.w3.org/2001/XMLSchema#dateTime> ;"
       //SettingUp-departure_time
       +"<"+model.createProperty(custom_ontology+"departingAt").toString()+"> "
       +" \""+railwayStation.departingTime+"\"^^<http://www.w3.org/2001/XMLSchema#dateTime> ; "
       //SettingUp-route_id
       +"<"+model.createProperty(custom_ontology+"hasRouteId").toString()+"> "
       +" \""+railwayStation.tripId+"\" ; "
       //SettingUp-agency_id
       +"<"+model.createProperty(custom_ontology+"serviceProvidedBy").toString()+"> "
       +" \""+railwayStation.timeTableNetwork+"\"@en  "
       +"] . ";
       if(count%2==0) {
           timeTableGraph+="}";
         LOG.info(timeTableGraph);
         LOG.info("STORING RDF DATA TO DB AT >>>>>> "+count);
         saveToGraphDb(timeTableGraph);
         LOG.info("SUCCESSFULLY STORED THE GRAPH DATA>>>>>>");
         timeTableGraph="INSERT DATA {";
```

Figure 11: Save real time timetable of railway station code

Above figure shows the partial code of `saveRealTimeRdf` function (Save railway station timetable function). This function present in "*StationDao.java*" Class. This function will get list of timetable data and prepare an RDF graph as blank-node and store it to the GraphDB. Above figure shows code example of how RDF-Blak-Node graph is prepared based on particular Railway Station IRI. **This complete graph is prepared with the use of our own ontology graph (OWL).**

Get timetable data of a station

```
public List<RailwayStation> getAllStationStatisticTimeTableDataByCityName(String cityName) {
    List<RailwayStation> railwayStations=new ArrayList<>();
    String graphQuery="PREFIX a: <https://www.wikidata.org/wiki/Property:> "
            + "PREFIX wd:<https://www.wikidata.org/wiki/> "
            + "PREFIX schema:<http://www.w3.org/2000/01/rdf-schema#> "
            + "PREFIX schemaOrg:<https://schema.org/> "
            + "PREFIX dbo: <http://dbpedia.org/ontology/>"
            + "PREFIX dbp: <http://dbpedia.org/property/>"
            + "PREFIX custom_ontology:<http://www.semanticweb.org/dhayananth/ontologies/2020/11/untitled-ontology-7#>"
            + ""
            + "SELECT DISTINCT ?comment ?stationComment ?city  ?station ?stationLabel ?stationCoordination ?branchCode ?departingAt ?arrivingAt "
            + "?hasDirection{"
            + "    ?city  a:P31 wd:Q484170 ; "
            + "    chema:label \""+cityName.toUpperCase()+"\" ;"
            + "   schema:comment ?comment ."
            + "    ?station  a:P31 schemaOrg:TrainStation ; "
            + "              a:P625 ?stationCoordination; "
            + "              schemaOrg:branchCode  ?branchCode; "
            + "              a:P131  ?city; "
            + "              schema:comment ?stationComment; "
            + "              schema:label ?stationLabel "
            + "            OPTIONAL{"
            + "                "
            + "             ?station  custom_ontology:hasTimeTable ?TrainStationTimeTable;"
            + "                                    custom_ontology:arrivingAt ?arrivingAt;"
            + "                                    custom_ontology:hasDirection ?hasDirection;"
            + "                                    custom_ontology:departingAt ?departingAt."
            + "               }"
            + "}"
            + "";
     QueryExecution queryExecution = prepaerQueryExecution(graphQuery);
     LOG.info("EXECUTING SPAREQL QUERY");
     try {
     for (ResultSet results = queryExecution.execSelect(); results.hasNext();) {
          QuerySolution qs = results.next();
          RailwayStation station=new RailwayStation();
          station.stationUri=qs.get("?station").toString();
          station.coordination=qs.get("?stationCoordination").toString();
          station.cityUri=qs.get("?city").toString();
          station.cityLabel=cityName;
          station.branchCode=qs.get("?branchCode").toString();
          station.stationLabel=qs.get("?stationLabel").toString();
```

Figure 12: Get timetable data of railway stations

Above code shows the implementation of GET timetable data of all stations by city name. This function will prepare a SPARQL query by concatenating city name. The GraphDB will return all timetable RDF Blank-Node data as query result. This query results will be further processed to create list of train stations with it's real time data. Only partial part of the code is available in above image. Full code can be accessed in "*StationDao.java*" Class.

Save bike stations real-time data

```
Model model =ModelFactory.createDefaultModel();
String bikeStationGraph="INSERT DATA {";
int count=0;
LOG.info("PREPARING CITY RDF DATA");
try {
for(BikeStation bikestation: bikeStations) {
      String bikeStationQid=bikestation.uri;
      //SETTING UP AVAILABILITY BLANK NODE
      bikeStationGraph+="<"+bikeStationQid+"> "
          +"<"+model.createProperty(custom_ontology+"hasAvailability").toString()+"> "
          +" [ "
/                 Setting up blank node type
              + "<"+property+"P31> "
              +"<"+model.createProperty(custom_ontology+"Availability").toString()+"> ; "
  //          Setting up recorded time
              +"<"+model.createProperty(custom_ontology+"recordedAt").toString()+"> "
              +" \""+bikestation.recordedAt+"\"^^<http://www.w3.org/2001/XMLSchema#dateTime> ;"
/             Setting up available
              +"<"+model.createProperty(custom_ontology+"numberOfAvailability").toString()+"> "
              +" \""+bikestation.availability+"\"^^<http://www.w3.org/2001/XMLSchema#int> "
          + "] . ";
      if(count%2==0) {
          bikeStationGraph+="}";
        LOG.info(bikeStationGraph);
        LOG.info("STORING RDF DATA TO DB AT >>>>>> "+count);
        saveToGraphDb(bikeStationGraph);
        LOG.info("SUCCESSFULLY STORED THE GRAPH DATA>>>>>>");
        bikeStationGraph="INSERT DATA {";
      }
      count++;
   }
   bikeStationGraph+="}";
   LOG.info(bikeStationGraph);
   LOG.info("STORING RDF DATA TO DB>>>>>>");
   saveToGraphDb(bikeStationGraph);
   LOG.info("SUCCESSFULLY STORED THE GRAPH DATA>>>>>>");
}
catch(Exception e) {
```

Figure 13: Save bike station real time data

The above figure shows how the bike station real time data has been stored in the GraphDB. This is partial code of the function and full code can be accessed in "*BikeStationDao.java*" Class. Bike station real time data will be prepared as RDF Blank-Node with corresponding "Bike station IRI". **Most of the predicates are defined in our own ontology graph (OWL).**

Retrieve bike station real time data

```
graphQuery="PREFIX a: <https://www.wikidata.org/wiki/Property:> "
        + "     PREFIX wd:<https://www.wikidata.org/wiki/> "
        + "     PREFIX schema:<http://www.w3.org/2000/01/rdf-schema#> "
        + "     PREFIX schemaOrg:<https://schema.org/> "
        + "     PREFIX dbo: <http://dbpedia.org/ontology/>"
        + "     PREFIX dbp: <http://dbpedia.org/property/>"
        + "     PREFIX custom_ontology:<http://www.semanticweb.org/dhayananth/ontologies/2020/11/untitled-ontology-7#>"
        + "     SELECT DISTINCT "
        + "     ?cityComment  "
        + "     ?city ?label"
        + "     ?name  ?bikestation ?capacity ?brand ?recorderAt ?coordinate ?numOf{"
        + "    ?city  a:P31 wd:Q484170 ; "
        + "           schema:label \""+cityName.toUpperCase()+"\" ;"
        + "           schema:comment ?cityComment ."
        + "     ?bikestation  a:P31 dbo:Station ;"
        + "                  a:P131 ?city ; "
        + "                   a:P1192 ?name ;  "
        + "                  dbp:storage ?capacity;"
        + "                  a:P625 ?coordinate ."
        + "      OPTIONAL{"
        + "            ?bikestation custom_ontology:hasAvailability ?availability."
        + "            ?availability a:P131 custom_ontology:Availability;"
        + "                         custom_ontology:numberOfAvailability ?numOf;"
        + "                        custom_ontology:recordedAt ?recorderAt."
        + "            }"
        + " OPTIONAL{"
        + "                ?bikestation schemaOrg:brand ?brand . "
        + "            }"

        + "     }";
 QueryExecution queryExecution = prepaerQueryExecution(graphQuery);
 LOG.info("EXECUTING SPAREQL QUERY");
 try {
 for (ResultSet results = queryExecution.execSelect(); results.hasNext();) {
      QuerySolution qs = results.next();
      BikeStation bikeStation=new BikeStation();
      bikeStation.uri=qs.get("?bikestation").toString();
      bikeStation.label=qs.get("?name").toString();
      bikeStation.cityUri=qs.get("?city").toString();
      bikeStation.cityName=cityName;
      bikeStation.brand="VERT";
      bikeStation.address="N/A";
```

Figure 14: Retrieve bike station real time data

Above image shows the code for retrieving bike stations along with it's real time data of "Rennes" city. The SPARQL query will retrieve all bike stations of "Rennes" along with it's real time data that has been stored as RDF Blank-Node such as "availability" and "recordedAt". The query result will be processed further to prepare the object list and send it to the corresponding controller.

Front-app implementations

Get SNCF API data

```
getRealTimeTrainSNCF(areaCode): Observable<HttpResponse<any>> {
    let todayDate = new Date();
    let preparedDateTime = todayDate.getFullYear() + (todayDate.getMonth() + 1) + todayDate.getDate()
        + "T" + todayDate.getHours() + todayDate.getMinutes() + todayDate.getSeconds();
    let headers = new HttpHeaders();
    headers = headers.set('Authorization', environment.sncf);
    let sncfUrl = "https://api.sncf.com/v1/coverage/sncf/stop_areas/" + areaCode + "/departures?datetime=" + preparedDateTime;
    return this.http.get<any>(sncfUrl, { headers: headers, observe: 'response' });
}
//SaveRealTimeDataAsRDF
postStation(stations) {
    return this.http.post<any>(
        this.baseUrl + "station/upload", stations, { observe: 'response' });

}
```

Figure 15: Get request to SNCF Portal

Above figure shows code snippet from Angular application. “getRealTimeTrainSNCF” function will get real time timetable data of specific railway station. For this GET request, we attach the branch code and authorization key that is available in environment file. The second function “postStation” will send this real time data to “Server-application” to store it in GraphDB.

```
updateRealTimeDataTimeTable(data, uri) {
  let timeTable = [];
  data.forEach(element => {
    timeTable.push({
      stationUri: uri,
      stopPoint: element.stop_point.lable,
      commercialModes: element.stop_point.commercial_modes.name,
      transportMean: element.stop_point.physical_modes.name,
      timeTableDirection: element.display_informations.direction,
      arrivingTime: element.stop_date_time.base_arrival_date_time,
      departingTime: element.stop_date_time.base_departure_date_time,
      timeTableNetwork: element.display_informations.network,
      timeTableLabel: element.display_informations.label,
      tripId: element.route.direction.id
    })
  });
  this.stationService.postStation(timeTable)
    .subscribe(res => {
      console.log(res)
    }, error => {
      console.log(error)
    })
}
```

Figure 16: Modelling the timetable data

Above code snippets shows how the modelling of timetable data is made by using JavaScript. This prepared json will be then posted to the server application using “postStation” function discussed in previous figure (*figure 14*).

Preparing JSON-LD

```
prepareAndInsertStationJsonLd(data) {
  let stationSchema = [];
  data.forEach(element => {
    let schema =
    {
      "@context": "https://schema.org",
      "@type": "TrainStation",
      "name": element.stationLabel,
      "location": {
        "@type": "Place",
        "name": element.cityLabel,
        "address": {
          "@type": "PostalAddress",
          "addressLocality": element.cityLabel,
          "addressRegion": "FR",
          "addressCountry": "FR"
        }
      },
      "identifier": element.stationUri,
      "branchCode": element.branchCode,
      "organizer": {
        "@type": "Organization",
        "name": "SNCF",
      },
      "Language": "French",
      "description": `${element.stationLabel} is an french railway station located in the city of ${element.cityLabel}`,
      "url": `https://www.google.com/search?q=${element.cityLable} ${element.stationLabel}`
    };
    stationSchema.push(schema);
    this.jsonLdTags.push(schema);
```

Figure 17: Preparing JSON-LD

Above image shows the code snippet for preparing JSON-LD object from Angular application. The function will get array of railway station data and loop through that array in order to prepare the JSON-LD object for each railway stations. Then the prepared object will be passed to the JSON-LD service class in order to insert the object in appropriate tag of DOM element. This part is done by following function (see below figure).

```
insertSchema(schema: Record<string, any>, className = 'structured-data'): void {
    let script;
    let shouldAppend = false;
    if (this._document.head.getElementsByClassName(className).length) {
        script = this._document.head.getElementsByClassName(className)[0];
    } else {
        script = this._document.createElement('script');
        shouldAppend = true;
    }
    script.setAttribute('class', className);
    script.type = JsonLDService.scriptType;
    script.text = JSON.stringify(schema);
    if (shouldAppend) {
        this._document.head.appendChild(script);
    }
}
```

Figure 18: Insert JSON-LD to DOM element

OWL Implementation

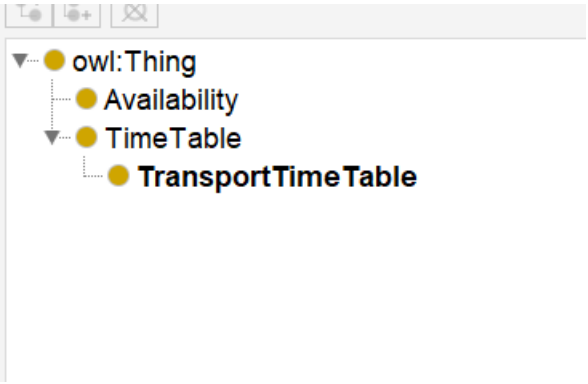


Figure 19: Schema classes

We defined three owl classes for our project using Protege tool. The "TransportTimeTable" is sub class of "TimeTable" class. This class contains transport timetable specific properties and "Timetable" class contains generic timetable related properties.

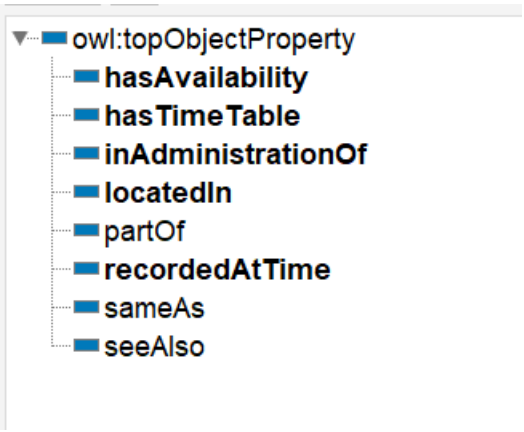


Figure 20: Object properties

These are the defined object properties in our own ontology. Each property belongs to specific schema class. Full details of this properties can be viewed in provided ontology file. These object properties used to make relation between two "**Entities**" (Subject and Object both are entities).

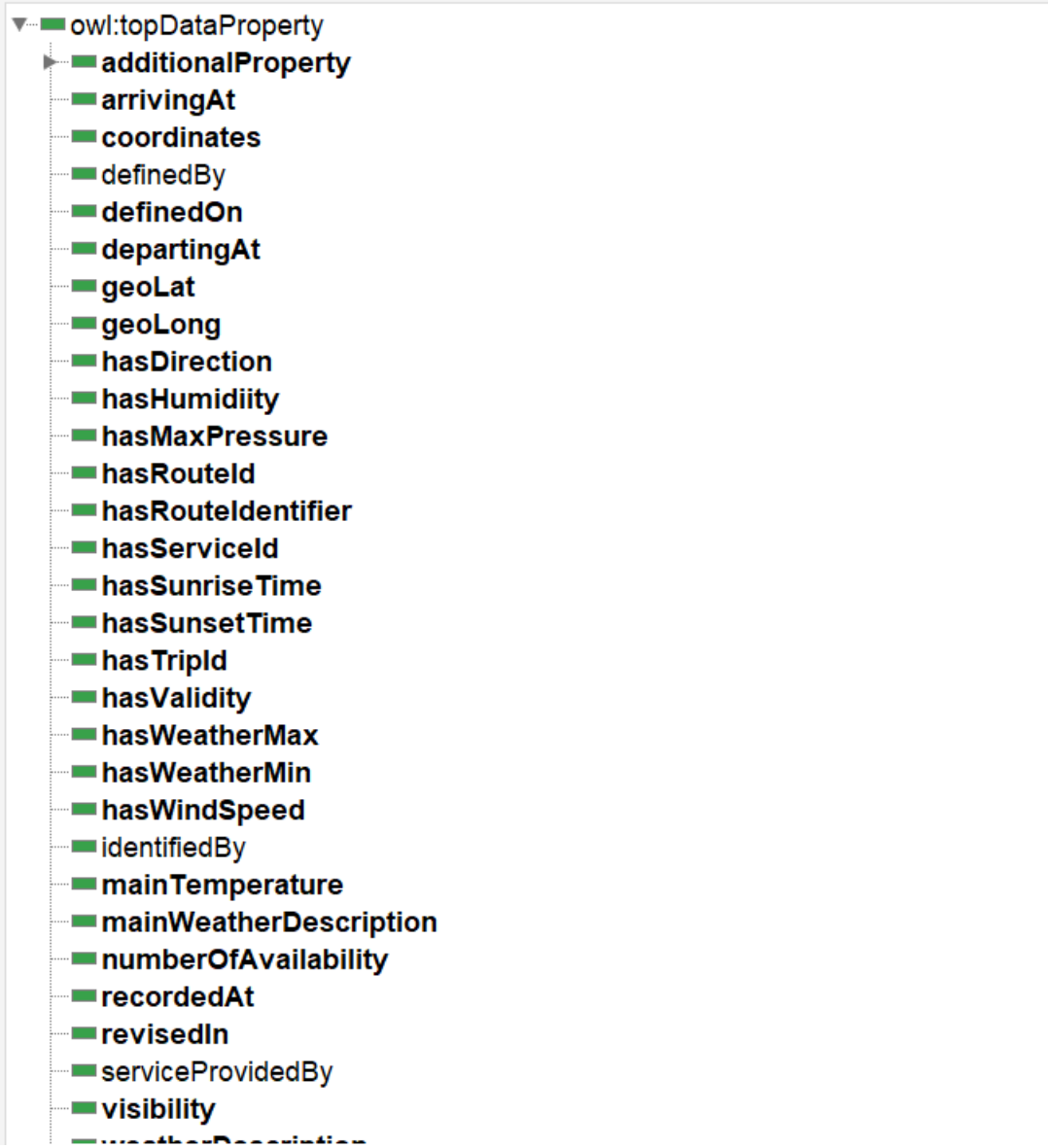


Figure 21: Datatype properties

We define different data type properties that helps to make relation with "**Entity**" (Subject) and "**Literals**" (Object).

Configuration support

How to get started with the Project

1. To run the program Java 11 or greater version must be installed in the computer
2. Download the Application file from Git repositories (front-app & server-application repositories need to be downloaded) (Here is the link : click-here)
3. Import the project to Eclipse as Maven Project. (Here is the link : click-here)
4. Install the GraphDB (see instruction below)
5. Import custom_ontology.ttl file to the graphDB in import section.
6. Update the application properties with relevant path of GraphDB endpoint for both UPDATE, and QUERY. (in `application.properties` and `cityDao.java` files)
7. Run the java application
8. Run the front-app (see below for instruction)
9. In the User interface there are Specific buttons which allows to see what the application is performing.

Installation GraphDB

To find out the detailed ways to follow the above process, please click-here

Run the front-app

- Requirement
  - NodeJS (click here to see how to install)
- Clone the repository (front-app)
  - https://github.com/ujm-closed
- Open command window in root of the front-app folder and type "*npm install –save*". This command will pull all necessary npm packs from online repository to your local computer.
- Type "ng serve" in the same command window.
- Access the application in "http://localhost:4200/" address.

References

3. Apache jena org. (2020). *Apache jena*. Retrieved from Apache jena : https://jena.apache.org/
4. BROWN, K. (2019, 11 13). *What Is GitHub, and What Is It Used For?* Retrieved from How to geek: https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/
5. GraphDB org. (2020). *GraphDB General*. Retrieved from Ontotext GraphDB: https://graphdb.ontotext.com/documentation/free/index.html
6. JSON-LD org. (2018). *JSON for Linking Data*. Retrieved from JSON-LD: https://json-ld.org/
7. MDN contributors. (2020, 03 01). *The WebSocket API (WebSockets)*. Retrieved from MDN Web docs: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
8. Protege team. (2020). *Protege*. Retrieved from Protege: https://protege.stanford.edu/
9. W3C Org. (2014). *SEMANTIC WEB*. Retrieved from W3C: https://www.w3.org/standards/semanticweb/#:~:text=The%20term%20%E2%80%9CSemantic%20Web%E2%80%9D%20refers,SPARQL%2C%20OWL%2C%20and%20SKOS.

Code Reference

JSON LD and Angular : https://medium.com/javascript-in-plain-english/how-to-use-json-ld-for-advanced-seo-in-angular-63528c98bb91

Apache Jena to GraphDB : https://graphdb.ontotext.com/documentation/free/using-graphdb-with-jena.html

Library Reference

Configuration file : https://www.codejava.net/coding/reading-and-writing-configuration-for-java-application-using-properties-class

Spring boot : https://spring.io/projects/spring-boot

Jackson.jar : https://github.com/FasterXML/jackson

graphdb :https://graphdb.ontotext.com/