Collaborative Project

GeoKnow - Making the Web an Exploratory for Geospatial Knowledge

Project Number: 318159 **Start Date of Project:** 2012/12/01 **Duration:** 36 months

Deliverable 4.1.1

Initial Release of the Spatial-Semantic Exploration Component

Dissemination Level	Public
Due Date of Deliverable	Month 12, 30/11/2013
Actual Submission Date	Month 12, 29/11/2013
Work Package	WP4, Spatial-Semantic Browsing, Visualisation and Authoring Interfaces
Task	T1
Type	Prototype Release
Approval Status	Approved
Version	1.0
Number of Pages	17
Filename	D4.1.1 Initial release spatial semantic exploration component.pdf

Abstract: Description of the initial prototype of *Mappify* and its underlying JavaScript library *Jassa*.

Project funded by the European Commission within the Seventh Framework Programme (2007 - 2013)

History

Version	Date	Reason	Revised by
0.0	03/11/2013	Initial Version	Jens Lehmann
0.1	07/11/2013	Structure	Claus Stadler
0.2	13/11/2013	Draft of Mappify	Patrick Westphal
0.3	15/11/2013	Draft of Jassa	Claus Stadler
0.4	20/11/2013	Finalized Mappify	Patrick Westphal
0.5	20/11/2013	Finalized Jassa	Claus Stadler
0.6	21/11/2013	Proof reading	Jens Lehmann
0.7	27/11/2013	Peer review	Jon Jay Le Grange
0.8	28/11/2013	Addressed peer review comments	Claus Stadler
1.0	28/11/2013	Approval	Jens Lehmann

Author List

Organization	Name	Contact Information
InfAI	Claus Stadler	cstadler@informatik.uni-leipzig.de
InfAI	Patrick Westphal	pwestphal@informatik.uni-leipzig.de
InfAI	Jens Lehmann	lehmann@informatik.uni-leipzig.de

Executive Summary

This deliverable describes the spatial-semantic browsing tool *Mappify* and the underlying re-usable *JAvascript Suite for Sparql Access (Jassa)* JavaScript library. Mappify allows one to easily create simple map applications based on RDF data retrieved from a SPARQL endpoint. For this purpose, Mappify defines a workflow and user interface for the creation of interactive maps. The approach of Mappify is summarized as follows: Via faceted search, a user defines one or more *concepts*. These are sets of items of interest, such as restaurants or monuments. Each concept can then be associated with a marker style for display on a map. Clicking a marker opens a popup with customizable content: The content itself is specified using an AngularJS HTML template which can be instantiated from JSON objects. The JSON is obtained from a declaration of which attributes of a concept to fetch from the SPARQL endpoint and how to perform the transformation to JSON. Mappify supports the export of the configuration and the generation of an HTML/JavaScript skeleton. The skeleton can be used e.g. as a base for a stand alone application or for embedding into a web page, such as for showing amenities near an event.

The Jassa library is an attempt of significantly easing interaction with SPARQL endpoints using multiple layers of abstraction. As such, it is not only an essential building block for Mappify, but it also bears the potential for use in a much wider range of applications. At the core, Jassa aims at offering a solid RDF API for building complex Semantic Web web applications. Therefore, Jassa's RDF API design closely resembles that of the well-known Apache Jena project. *Facete* is a Jassa component that offers several classes and utilities for SPARQL-based faceted search. *Sponate* is a component that abstracts access to a SPARQL endpoint using an interface similar to that of the popular JSON store *MongoDB*. Sponate mappings provide a declarative way for transforming SPARQL result sets into JSON documents. The Sponate engine has initial capabilities of rewriting queries posed against the Mongo-DB interface into SPARQL queries based on the Sponate mapping definitions.

D4.1.1 is a software prototype deliverable. Mappify and Jassa are publicly available at `https://github.com/GeoKnow/Mappify` and `https://github.com/GeoKnow/Jassa`, respectively.

A landing page for promoting current and future JavaScript libraries published as part of GeoKnow has been established at `http://js.geoknow.eu`.

Abbreviations and Acronyms

LOD	Linked Open Data
RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
JSON	JavaScript Object Notation
Jassa	JAvascript Suite for Sparql Access

Table of Contents

1 Introduction

This deliverable is comprised of two parts: The first one presents the Mappify application. The second one introduces the JAvaScript Suite for Sparql Access (Jassa), the Javascript library project that includes Facete and Sponate.

2 The Mappify Web Application

To provide means to easily explore and visualize spatial content retrieved from the Web of Data, new paradigms like faceted browsing have to be explored. Apart from this, new techniques need to be developed to combine semantic information with geospatial data and display them with state-of-the-art map rendering tools. Moreover, to support the reuse of certain application units and separate different concerns like browsing and presentation a software implementing these paradigms and techniques should consist of distinguishable components.

Mappify comprises different libraries and widgets supporting these aims. Hence a separate library for the facet based browsing component is used. The task of providing a unified model to access semantic and geospatial data is also solved by a dedicated software library. These libraries are consolidated in the *JAvascript Suite for Sparql Access (Jassa)*[1]. The actual geospatial data are displayed using the libraries of the OpenLayers project[2]. To also be able to apply such a modular design to the user interface the JavaScript framework AngularJS[3] was utilized.

In addition to the facet-based exploration of the SPARQL endpoint at hand, Mappify provides controls to define complex concepts describing sets of resources with certain property values that can be displayed on a map. This map is further constrained to a certain geospatial area of interest for the considered application context.

Mappify is Free Software provided under the Apache v2 license and can be retrieved under https://github.com/GeoKnow/Mappify. The following sections explain how to install Mappify and introduce the user interface components.

2.1 Installation

The Mappify tool is intended to be used as a single-page web application (SPA) and is thus implemented in client-side JavaScript running in the user's web browser. Accordingly, the Mappify JavaScript has to be delivered by a web server. Assuming that a web server is configured and running with a folder named `/srv/www` for serving web applications, and the programs NodeJS[4], npm[5] and the Bower package manager[6] are installed, the steps to make Mappify available are shown in the following listing.

```
1 cd /srv/www
2 git clone https://github.com/GeoKnow/Mappify
3 cd Mappify
4 bower install
```

Pointing your browser to http://localhost/Mappify/app should now show the Mappify user interface described in the following section.

2.2 Application User Interface Components

The Mappify user interface is divided into a concept configuration panel and the map preview as shown in Figure 1.

[1] https://github.com/GeoKnow/Jassa
[2] http://openlayers.org
[3] http://angularjs.org
[4] http://nodejs.org/
[5] https://npmjs.org/
[6] http://bower.io/

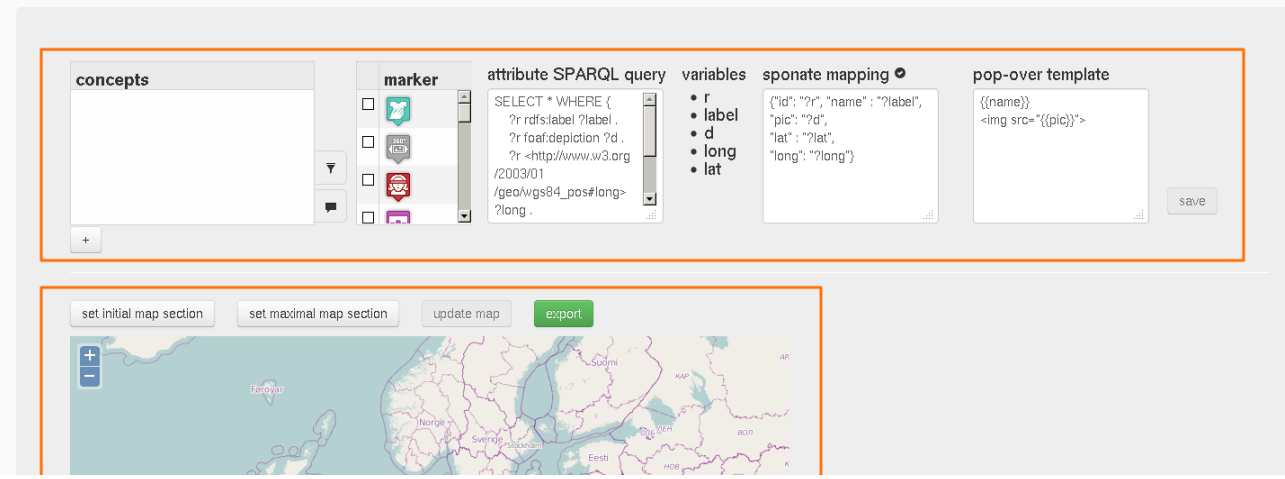
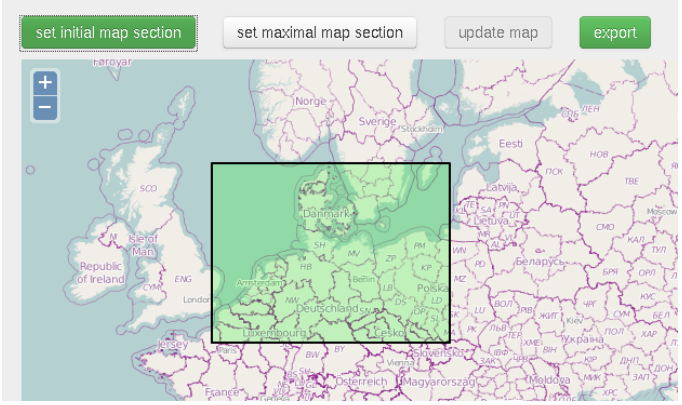
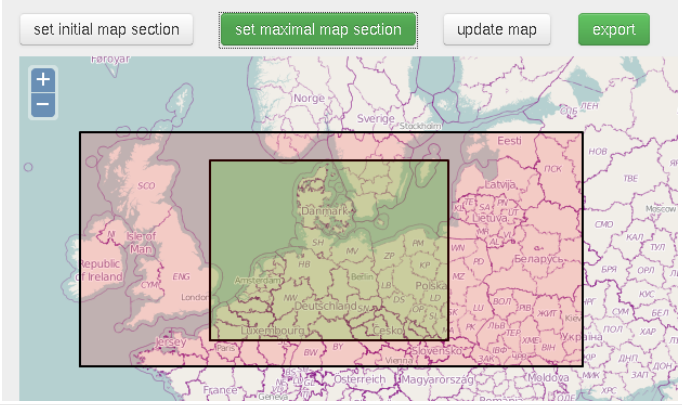
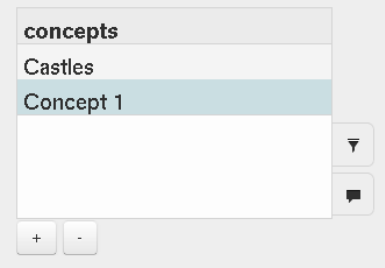


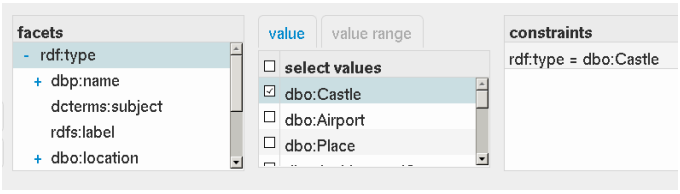
Figure 1: Screenshot of the Mappify user interface with highlighted concept configuration panel (upper box) and map preview area (lower box)

To set up a Mappify map configuration the following steps need to be applied using the user interface components as described in the following.

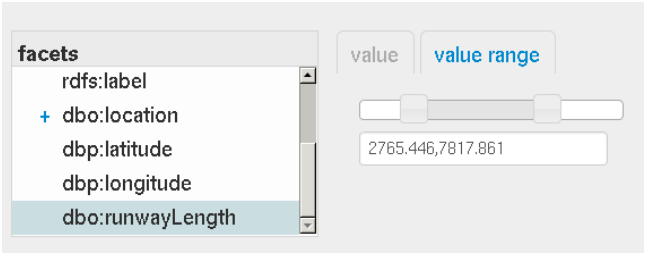
Depiction	Description
	In the map preview area, the initial and maximal map section areas can be drawn as corresponding bounding boxes. Drawing of the initial bounding box is enabled after activating the 'set initial map section' button. When generating the skeleton code e.g. with the purpose of embedding it in a web site, the map will be initialized to the specified area.
	To restrict the area within which resources are considered for display on the map, the 'set maximal map section' button is used. As with the initial map section, a bounding box can be drawn after the button's activation. The bounding box is then used to filter items in combination with any constraints set by the faceted search.



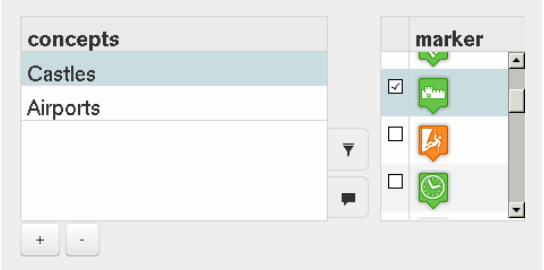
The concept configuration panel provides means to create and configure concepts to be displayed on the map. The first step is to create a concept. This is done by clicking on the '+' button. The new entry in the concept grid will have a default name that can be updated. Double-clicking the entry will show a form field to enter the new name. A concept can also be deleted by selecting it and then clicking the '-' button. Apart from these functionalities the concept grid is also used to switch between the constraint settings (filter symbol on the upper right) and the map appearance related settings (info pop-up symbol on the lower right). Depending on this choice the constraint or map appearance setting controls are shown on the right hand side.



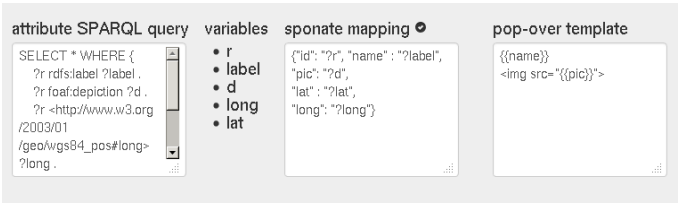
When choosing the constraint settings, an expandable facet tree is presented. Clicking on an arbitrary facet allows the user to set up value constraints for the selected facet. These are defined in the middle part by restricting the facet values to a set of resources or literals. The values presented there already reflect the overall constraint settings and thus exclude values that are not possible with respect to other constraints stored for the considered concept. The constraints already defined are summarized in the constraints grid on the right.



Another option is to restrict numerical facet values via the value slider available when clicking the 'value range' tab. This slider is only available if there are any numerical values and is then preset to the range of all values available for the selected facet with respect to existing constraints.

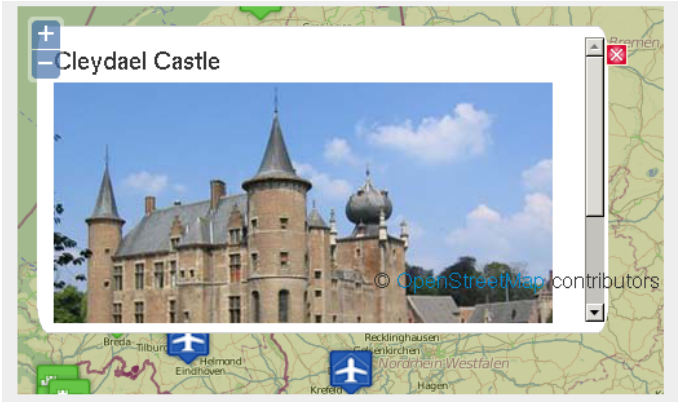


To configure how concepts should be displayed on the map, one has to switch to the map appearance settings. There, the first control provides a library of marker icons to chose from. The chosen marker icon will later be used to represent a resource on the map.



The remainder of the map appearance settings is dedicated to the info box popping up when clicking the marker icon. To specify what to display there, users can define their own Sponate mappings comprising a SPARQL query (text area field on the left), the actual mapping definition (text area field in the middle) and a pop-up template (text area field on the right). In the mapping definition Sponate variables are defined on the basis of the entered SPARQL query. These variables can then be used in the template together with arbitrary HTML markup. To be able to distinguish between both, the Sponate variables have to be enclosed in curly braces *{{...}}*. The *variables* overview shows an instantaneously updated list of SPARQL variables appearing in the query that can be used in the Sponate mapping definition.

Having defined the constraint and appearance related settings of a concept, all available resources satisfying the stored concept can be shown on the map. To trigger this the 'update map' button has to be clicked. Thereafter all resources appear, represented by the corresponding markers configured. The example shown displays castles and airports of north western Germany, the Netherlands and northern Belgium.



When clicking on a marker icon, its configured info text pops up replacing the Sponate variables of the underlying template with the corresponding values retrieved via an on-demand SPARQL query. After finishing the map set-up the 'export' button can be clicked to trigger the export and save the HTML snipped containing the necessary JavaScript and Mappify configurations.

3 The Jassa Library

Mappify makes use of two novel technologies for JavaScript-based interaction with SPARQL endpoints which are provided by the JAvascript Suite for Sparql Access (Jassa) library:

- *Sponate* is a SPARQL-to-JSON mapper with the goal of offering a MongoDB-like API[7] for accessing virtual JSON documents based on a SPARQL endpoint and a set of mapping definitions. Sponate is also in use on the http://js.geoknow.eu website, which lists all JavaScript libraries published as part of GeoKnow.
- *Facete* is a library for faceted search on SPARQL accessible data.

The Jassa code base can be used both on the client and the server side. The only difference between these settings lies in the dependencies that need to be included. Furthermore, Jassa provides several RDF and SPARQL foundation classes which are nearly identical to those of the excellent API of the Java-based Apache Jena project[8]. The rationale followed by Jassa is to exploit existing, well-known API designs, rather than to invent new ones. In the following we describe Jassa's dependencies, how to perform a client and server side set up, and the components (modules) it provides. Afterwards we give a brief introduction to Sponate and Facete and code samples for their use.

3.1 Dependencies

Jassa's dependencies and why they are needed are listed below.

- *jquery*[9]: Asynchronous requests are based on jQuery's $.ajax
- *prototype*[10]: The *Class* object is needed from prototype.
- *underscore*[11]: Provides several utility functions for working with (associative) arrays.
- *underscore.string*[12]: Provides several utility functions for working with strings.
- *xmlhttprequest*[13]: NodeJs only; i.e. not needed for browser based set up. This library emulates the browser's XMLHttpRequest object, and makes jQuery's $.ajax work from nodejs. Must be included before jQuery.

3.2 Installation

Jassa can be obtained via npm:

```
1 npm install jassa
```

Alternatively, it can be built from source:

[7] http://docs.mongodb.org/manual/reference/operator/nav-query/
[8] http://jena.apache.org
[9] http://jquery.com/
[10] http://prototypejs.org/
[11] http://underscorejs.org/
[12] https://github.com/epeli/underscore.string
[13] https://npmjs.org/package/xmlhttprequest

```
git clone https://github.com/GeoKnow/Jassa.git
cd Jassa/jassa-js/
mvn package
```

This generates the Jassa library and its minified version under:

```
./target/jassa/webapp/resources/js/jassa.js
./target/jassa/webapp/resources/js/jassa.min.js
```

3.3 Client-side Set Up for Browsers

Once all necessary dependencies are obtained, reference them in an HTML file similar to the example below. Adjust the paths and versions to your needs.

```
<html>
  <head>
    <script src="resources/libs/jquery/1.9.1/jquery.js"></script>
    <script src="resources/libs/underscore/1.4.4/underscore.js"></script>
    <script src="resources/libs/underscore.string/2.3.0/underscore.string.js">
    </script>
    <script src="resources/libs/prototype/1.7.1/prototype.js"></script>

    <script src="resources/libs/jassa/0.5.0/jassa.js"></script>

    <script type="text/javascript">
      _.mixin(_.str.exports());

      // The Jassa object is now readily available
    console.log("The Jassa object: ", Jassa);
    </script>
  </head>
</html>
```

3.4 Server-side Set Up using NodeJS

Installing Jassa via npm also downloads its dependencies. In the application, the referenced modules need to be set up as follows:

```
var XMLHttpRequest = require("xmlhttprequest").XMLHttpRequest;

$ = require('jquery');

$.support.cors = true;
$.ajaxSettings.xhr = function () {
    return new XMLHttpRequest;
}

require('prototype');

_ = require('underscore');
_.str = require('underscore.string');

```

```
_.mixin(_.str.exports());

var Jassa = require('jassa');
```

3.5 The Jassa Architecture

At present, Jassa comprises the following modules:

- *util*: A util module. Contains collections, such as HashMap and HashSet.
- *rdf*: The module that holds core classes related to RDF.
- *vocab*: A module for vocabularies, expressed in terms of classes of the rdf module.
- *sparql*: A module for core classes related to SPARQL. Builds upon the prior modules.
- *service*: Abstraction layer for SPARQL endpoints.
- *facete*: A faceted search module.
- *sponate*: A SPARQL-to-JSON mapper. Particularly powerful in combination with the generation of web frameworks that offer a clean separation between DOM and application logic, such as angularjs and Ember.js.

In the following, the foundation classes, facete and sponate are briefly explained.

3.6 Foundation Classes

The Jassa core classes aim to serve as a solid foundation for JavaScript-based Semantic Web applications.

The *rdf* module provides the *Node* class for encapsulating RDF terms. In contrast to approaches that are based on plain JSON, a Jassa node object provides methods such as *isLiteral()* and *getLiteralDatatypeUri()*. Furthermore, the *toString()* method is implemented to yield meaningful string representations.

The *sparql* module contains several classes for the syntactic representation of SPARQL queries. Their use is demonstrated below.

```
var rdf = Jassa.rdf;
var sparql = Jassa.sparql;

var query = new sparql.Query();
var s = rdf.NodeFactory.createVar("s");
var p = rdf.NodeFactory.createVar("p");
var o = rdf.NodeFactory.createVar("o");

var triple = new rdf.Triple(s, p, o);

query.setElement(new sparql.ElementTriplesBlock([triple]));
query.setResultStar(true);
query.setLimit(10);

console.log("QueryString: " + query);
```

Listing 1: Programmatic creation of the SPARQL query "Select * { ?s ?p ?o } Limit 10"

The *facete* faceted search module highly depends on programmatic creation and optimization of queries.

The *service* module defines the following interfaces:

- *QueryExecutionFactory*: A factory class for QueryExecution objects. May provide common settings for all created QueryExecution objects.
- *QueryExecution*: A class that supports starting the execution of a query
- *ResultSet*: A result set is essentially an iterator over a collection of SPARQL bindings.

```
var service = Jassa.rdf;

var qef = new service.QueryExecutionFactoryHttp(
              "http://dbpedia.org/sparql",
              ["http://dbpedia.org"]
          );

var qe = qef.createQueryExecution("Select * { ?s ?p ?o } Limit 10");
qe.setTimeout(5000); // milliseconds

qe.execSelect()
    .done(function(rs) {
        while(rs.hasNext()) {
            var binding = rs.nextBinding();
            console.log("Got binding: " + binding);
        }
    })
    .fail(function(err) {
        console.log("An error occurred: ", err);
    });
```

3.7 Sponate

Sponate is a SPARQL-to-JSON mapper that offers an API similar to that of MongoDB based on a set of mapping definitions. The example below shows how to create a Sponate mapping and how to query it.

```
var service = Jassa.service;
var sponate = Jassa.sponate;

var prefixes = {
  'dbpedia-owl': 'http://dbpedia.org/ontology/',
  'dbpedia': 'http://.org/resource/',
  'rdfs': 'http://www.w3.org/2000/01/rdf-schema#',
  'foaf': 'http://xmlns.com/foaf/0.1/'
};

var qef = new service.QueryExecutionFactoryHttp('http://dbpedia.org/sparql', ['
    http://dbpedia.org']);

var store = new sponate.StoreFacade(qef, prefixes);

store.addMap({
  name: 'castles',
```

```
  template: [{
    id: '?s',
    name: '?l',
    depiction: '?d',
    owners: [{
      id: '?o',
      name: '?on'
    }]
  }],
  from: '?s a dbpedia-owl:Castle ; rdfs:label ?l ; '
       + 'foaf:depiction ?d ; dbpedia-owl:owner ?o .'
       + '?o rdfs:label ?on .'
       + 'Filter(langMatches(lang(?l), "en"))'
       + 'Filter(langMatches(lang(?on), "en"))'
});

var filterText = 'tle'; // The search string
var criteria = {name: {$regex: filterText}};
var flow = store.castles.find(criteria).limit(10).skip(10);

flow.asList()
  .done(function(docs) {
      _(docs).each(function(doc) {
          console.log("JSON document: " + JSON.stringify(doc));
      });
  })
  .fail(function(err) {
       console.log("An error occurred: ", err);
  });
```

3.8 Facete

Facete is a module that builds upon all prior modules and offers SPARQL-based faceted search.

Facete is conceptually based on three fundamental constructs:

- *Concept*: A concept is a pair comprised of a SPARQL graph pattern and a variable thereof.
- *Path*: A list of steps, whereas a step is comprised of an RDF property name and a direction. A step's direction is either *forward* or *backward* and specifies whether the step leads to the set of resources occurring in the subject or the object position of that property.
- *FacetNode*: A facet node represents a variable mapping for a given path. Equivalent paths are always mapped to the equivalent variables.

Facete is configured with the following essential configuration:

- The *Base Concept* specifies the initial set of resources on which the faceted search is performed.
- The *Constraint Manager* a container for constraints.
- The *Root Facet Node*: The root node of a tree structure with the purpose of mapping paths to SPARQL variables and vice versa.

The example below demonstrates the programmatic set up:

```
var constraintManager = new facete.ConstraintManager();

var baseVar = rdf.NodeFactory.createVar("s");
var baseConcept = facete.ConceptUtils.createSubjectConcept(baseVar);
var rootFacetNode = facete.FacetNode.createRoot(baseVar);

// Based on above objects, create a provider for the configuration
// which the facet service can build upon
var facetConfigProvider = new facete.FacetGeneratorConfigProviderIndirect(
  new facete.ConceptFactoryConst(baseConcept),
  new facete.FacetNodeFactoryConst(rootFacetNode),
  constraintManager
);

var fcgf = new facete.FacetConceptGeneratorFactoryImpl(facetConfigProvider);
var facetConceptGenerator = fcgf.createFacetConceptGenerator();


var expansionSet = new util.HashSet();
expansionSet.add(new facete.Path());

var facetService = new facete.FacetServiceImpl(qef, facetConceptGenerator);
var facetTreeService = new facete.FacetTreeServiceImpl(facetService,
    expansionSet);

facetService.fetchFacets()
    .done(function(facetTree) {
        console.log("FacetTree: " + JSON.stringify(facetTree));
    })
    .fail(function(err) {
        console.log("An error occurred: ", err);
    });
```

4 Conclusions and Future Work

The Mappify prototype currently provides a web-based user interface to explore and visualize RDF data and to generate a code skeleton based on the user supplied configuration. Mappify is built using state-of-the-art web technologies, such as AngularJS and the Jassa library. Jassa is our effort to offer advanced and reusable JavaScript building blocks for complex client and/or service Semantic web applications. For both Mappify and Jassa there exist several routes for future developments and improvements.

In regard to Mappify, a weakness is that the user-friendliness of its user interface declines as the complexity of the data and mappings increase. For this reason, we intend to rework the layout and some of the controls. Furthermore, although Mappify supports the generation of a code skeleton based on the user's configuration, it currently lacks support for saving and restoring projects. In regard to the spatial features, currently only support for the WGS84[14] vocabulary is implemented. In the future, we plan to additionally support GeoSPARQL[15] and GeoVocab[16]. Another topic worth investigating is, how Mappify can be extended to deal with geospatial traces, such as displaying the trace of a sightseeing tour together with taken pictures.

Regarding Jassa, work will be focused on improving its components, especially *Facete* (faceted search module) and *Sponate* (SPARQL-to-JSON mapper). Besides bug fixes, a general task is to improve all the component's scalability by means of a (common) caching system. Facete could be further enhanced with new facet retrieval strategies which may increase its performance. Sponate would benefit from support for creating JSON documents from unions of Sponate mappings, and an extended set of query operators. Finally, in the course of the GeoKnow project, the components are planned to be evaluated along different dimensions, such as scalability, ease-of-use/friendliness and fitness-for-use.

[14] http://www.w3.org/2003/01/geo/wgs84_pos#
[15] http://www.opengeospatial.org/standards/geosparql
[16] http://geovocab.org/

5 Licences

Mappify and Jassa are freely available under the Apache v2 license.