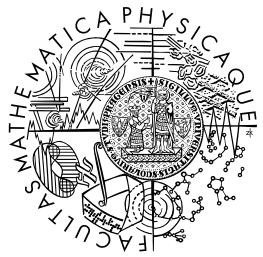


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Vladislav Vancák

Graph data visualizations with D3.js libraries

Department of Software Engineering

Supervisor of the bachelor thesis: doc. Mgr. Martin Nečaský, Ph.D.
Study programme: Computer Science
Study branch: IPSS

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 20.07.2017 signature of the author

Title: Graph data visualizations with D3.js libraries

Author: Vladislav Vancák

Department: Department of Software Engineering

Supervisor: doc. Mgr. Martin Nečaský, Ph.D., Department of Software Engineering

Abstract: Data visualisations help us process large volumes of complex data faster. The goal of this thesis is to implement visualizer plugins into the Linked Pipes Visualization Assistant – a platform for visualising graph data published in the RDF format. The Assistant automatically analyses data sources and matches them against available visualizers to create interactive views. All visualizers allow configuring the visualisation, saving the configuration and publishing created views. The first group of visualizers works with time-oriented data, viewing instants and intervals on a timeline chart. The second group of visualizers views map-oriented data on the Google Maps. Resources with coordinates are visualised using markers, while resources with coordinates and quantifiers are shown as circles (the circle radius corresponds to the quantifier).

Keywords: RDF, graph data, LinkedPipes, visualization

I would like to thank my thesis supervisor, doc. Mgr. Martin Nečaský Ph.D. for all his help and guidance. Furthermore, I would like to express my gratitude to the technical consultant of this thesis, RNDr. Jiří Helmich for his advice.

I would like to thank my family and friends for their continuous support during my Bachelor studies.

Contents

Introduction

The amount of data available online is already immense and keeps growing. Using charts or graphs to visualise large volumes of complex data has proven much easier than using tables, spreadsheets or reports - mainly because our ability to identify patterns and correlations when dealing with numbers is much poorer than our ability to recognise and compare various shapes or colours (also known as the picture superiority effect).

However, a task of creating an interactive visualisation based on a data set usually requires a developer, who will understand the data structure and implement the visualizer. In an attempt to simplify this process, *visualisation assistants* allow creating such interactive views by non-developers. A visualisation assistant should be able to analyse the source data and create views without requesting users to understand the underlying data structure.

Linked Data is a method of publishing structured data so that it can be queried using semantic queries. Information is described in the Resource Description Framework (RDF), which forms a directed, labelled graph, where the edges represent the named link between two resources, represented by the graph nodes. The method allows expressing any information by extending the raw data with machine-readable semantic meta-data.

A *Linked Data driven visualisation assistant* benefits from working with such extended data, as it can understand it better. Therefore the user is allowed to generate more versatile views, tailored for all different kinds of data.

The Linked Pipes Visualisation Assistant is a Linked Data driven visualisation assistant developed within projects of the *Department of Software Engineering of the Charles University*. To create a new interactive view, the user starts by selecting a source of RDF data. The assistant analyses the data and offers a list of visualizers that can visualise the data. The user selects the visualizer he is interested in and uses it to generate a new view. The assistant allows the user to configure the view and then publish it.

Goal of this thesis Since the RDF data can be very versatile in its structure, the Linked Pipes Visualisation Assistant is only as powerful as its visualizers. By introducing new visualizers, we will significantly enhance the Assistant's capability of visualising various data sources. Firstly, we will implement a set of timeline-based visualizers. We will implement a new TimeLine chart, using the D3.js libraries in the process. Secondly, we will use an existing Google Maps chart to create a set of map-based visualizers.

Thesis structure Chapter 1 covers an introduction to the Linked Data. The chapter provides a quick overview of concepts and offers a quick introduction to the technologies used in this thesis, necessary for a good understanding of the text.

Chapter 2 offers requirements analysis. For each of the required visualizers, we introduce the underlying data structure, required configurations and provide a few UI mockups to illustrate the desired result view.

Chapter 3 examines the platform, where the visualizers will be integrated. We introduce the Linked Data Visualization Model (LDVM) and the model's implementation in the LDVMi platform. After that, we show the Linked Pipes Visualisation Assistant, built on top of the LDVMi, give a quick overview of the Assistant's architecture and finish with a process of integrating new visualizers.

Chapter 4 deals with the implementation of required visualizers, described the implemented components and goes through the use cases.

Chapter 5 tests the implementation and shows the impact of various visualizer components and configurations on the application's performance.

1. Preliminaries

This chapter introduces the reader to the Linked Data. We will start with the basic concepts, but the main focus will be on presenting the Resource Description Framework (RDF) - we will go through RDF vocabularies, available serialization formats and the SPARQL query language.

1.1 Linked Data

The term Linked Data was introduced in 2006 by Tim Berners-Lee [1], the director of the World Wide Web Consortium, as a method of publishing structured data. To make the data readable by computers, the main principles are:

1. Use URIs as names for things
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)
4. Include links to other URIs, so that they can discover more things.

Tim Berners-Lee gave a presentation about linked data at 2009 TED conference [2], presenting altered version of the main principles and making the concept more popular among the public.

The recommendations for publishing linked data can be found at the official website[1].

1.2 Resource Description Framework

Resource Description Framework (RDF) [9] is a model for describing linked data. All information is stored using **triplets**. Each triplet consists of a subject, a predicate and an object.

The following example shows a triplet describing a relation "Person owns Company":

```
subject   http://example.org/people/person1
predicate http://schema.org/owns
object    http://example.org/companies/company1
```

[1] http://linkeddata.org/

While subject and predicate are always URIs, an object can also be a literal:

```
subject   http://example.org/book/book1
predicate http://purl.org/dc/elements/1.1/title
object    "SPARQL Tutorial"
```

Using such triplets, we can create a graph, where subjects and objects represent vertices, while predicates represent directed edges. Note, that a single URI can represent an edge and a vertex at the same time. Using that, we can introduce RDF **properties**, which represent relationships between resources. We will describe them as other resources (i.e. URI in subjects/objects) and then use them to represent relations (i.e. URI in predicates).

1.2.1 Vocabularies

While users can always publish data using properties they create, it is more common to use predefined ones (if available). An RDF vocabulary provides definitions of such properties and describes relationships between them and other resources.

Every vocabulary is identified by its **namespace**, represented by an URI (also called a namespace name). All properties described in the vocabulary are in that namespace (i.e. their URIs start with the namespace name). To make triplets shorter and cleaner, most of the data is described using **prefixes**. First, we define a prefix representing a namespace. We can now refer to properties from the vocabulary relatively to that namespace root, using a [prefix]:[relative_url] notation. Table 1.1 shows some of the most commonly used prefixes we will need in this text.

prefix	**namespace**	**name**
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#	RDF
rdfs	http://www.w3.org/2000/01/rdf-schema#	RDF Schema
xsd	http://www.w3.org/2001/XMLSchema#	XML Schema Definition
skos	http://www.w3.org/2004/02/skos/core#	SKOS
s	http://schema.org/	Schema
dct	http://purl.org/dc/terms/	DCMI Metadata Terms
gr	http://purl.org/goodrelations/v1# vocabulary	Good Relations

Table 1.1: Well-known vocabularies used in this thesis with prefixes we will use.

1.2.2 Serialization

As the RDF is just an abstract model, there are multiple ways of serializing the triplets.

HTML+RDFa [12] allows marking up data in HTML documents, thus making the web page machine- and human-readable at the same time,

RDF/XML [10] (sometimes just RDF) stores the triplets in XML files,

JSON-LD[2] serializes RDF data into the JSON format,

Turtle [11] (Terse RDF Triple Language) introduces a new file format, usually identified by .ttl suffix.

Turtle file is focused to represent data directly in triplets. The turtle syntax allows to define a prefix, which can be used instead of the URI it represents. This is making stored triplets shorter and therefore more readable. To make files even shorter, one subject may be assigned multiple predicates and objects. Such constructs are separated by semicolons, while dots separate two subject definitions.

Following example shows us a simple turtle syntax usage:

```
@prefix somePrefix: <http://www.example.net/schemas/relationships/> .

<http://example.org/#PersonA>
        somePrefix:enemyOf <http://example.org/#PersonB> ;
        somePrefix:friendOf <http://example.org/#PersonC> .

<http://example.org/#PersonB>
        somePrefix:friendOf <http://example.org/#PersonC>.
```

1.3 SPARQL

Now, since we have a way of serializing RDF data, we will need a way to access it using queries. With relational databases unable to directly store graph data, **Triplestores**[3] were created to serve this purpose.

To write queries, the SQL will not be enough either. The triplestores allow querying data using **SPARQL** [8]. The language consists of:

PREFIX - similar to @prefix in Turtle, to make the query more readable

[2] https://json-ld.org/

[3] https://en.wikipedia.org/wiki/Triplestore

SELECT to retrieve table-structured data

CONSTRUCT to retrieve triplets

ASK to determine whether any matching data exist

WHERE contains a set of conditions. Conditions can either be in form of triplets in Turtle format, or functions like *FILTER*, *(NOT) EXISTS* etc. Variables are also available - their names start with a question mark.

GROUP BY, *LIMIT* and others.

The following example shows a SPARQL query, which retrieves a table containing URI, longitude and latitude of available coordinates.

```
PREFIX s: <http://schema.org/>

SELECT ?coordinates ?longitude ?latitude
WHERE {
?coordinates s:longitude ?longitude ;
        s:latitude ?latitude .
}
GROUP BY ?coordinates ?longitude ?latitude
LIIMT 100.
```

The complete documentation of query language can be found at the official W3C site [8]. The section should, however, be enough to understand the basic principles and give the reader an insight needed to understand queries in this text.

2. Requirements analysis

In this chapter, we will go through required visualizer plugins. Firstly, we will introduce general requirements - labels, comments, colours and language support for visualised resources. Secondly, we will go through required Time Line visualizers and Google Maps visualizers, specifying visualised data formats, use cases and configuration requirements along the way.

2.1 General requirements

Although all RDF resources are identified using URIs, they are often extremely long, unreadable and searching among them is difficult. Therefore the application introduces Labels to describe resources in a more user-friendly way. Furthermore, to give the user more information, each visualised resource should also provide a comment, containing a short description.

2.1.1 Labels

In the RDF graph, Labels are stored as literals. To distinguish them from other literals, we have to introduce a set of properties used to link only Labels. When finding a suitable Label, the visualiser will look for triplets with those properties in predicates. In addition to that, properties will have to be ordered by relevance - and when multiple labels are available, the most relevant one is returned. In this thesis, we will use following properties:

1. rdfs:label
2. dct:title
3. skos:prefLabel
4. s:name
5. gr:legalName

2.1.2 Comments

Like labels, we will also use a set of properties to distinguish comments from other literals. A comment for a visualised resource should provide a short description of the resource. In this thesis, we will use following properties:

1. rdfs:comment

2. dct:description

3. skos:definition

4. skos:note

2.1.3 Dereferencing

Visualizers get all data from user-defined data sources. If a data source contains Label/Comment, we will display the available value directly. However, in a case of missing values, we can afford to look at them differently - since Labels and Comments do not significantly change the visualisation, we can try to fetch additional data. This scenario provides an opportunity to benefit from working with Linked Data - As the principles (in section 1.1) state, we should be able to get some information about resources by looking up ("dereferencing") its URI.

Algorithms for obtaining labels and comments will therefore have following structure:

1. Try fetching data from the source graph(s).

2. Try dereferencing resource URI and search for data within response.

3. Label / Comment not available.

2.1.4 Language support

Literals can contain language information stored by appending @language shortcut after the string - For example "This is a comment"@en is a valid English literal. This feature gives us an opportunity to monitor available languages and to allow users to select the visualisation language.

All visualizers should support this feature if multiple languages are available. However, missing language information should not influence the validity of input data.

2.1.5 Data descriptors

Each visualizer will be able to visualise resources of a pre-defined data structure. This structure will be described using *data descriptors*. A data descriptor is defined via an *SPARQL ASK* query. This query will be run against input data graph(s). If the ASK returns *true*, the graph(s) contain described data structure and can be visualised using the corresponding visualizer. We

will provide a data descriptor for every required visualizer in the following section.

2.1.6 Colours

For the visualizers that use shapes to visualise resources, we can use the colour of objects on the chart. Visualizers use this to separate shown resources:

Stroke colour is used to separate *properties* used to link visualised resources to the rest of data (e.g. predicate between Thing with Interval and the Interval).

Fill colour is used to separate *types* of visualized objects. A type of an RDF resource can be obtained via the *rdf:type* property.

2.2 Time Line

The first group of visualizers work with time-oriented data and displays it on a Time Line chart. Upon clicking on visualised resources a separate window, containing information about the selection, should be shown.

The visualised data can be further divided into Instant-oriented and Interval-oriented. **Instants** have one linked resource describing *date and time*, while **Intervals** have two linked resources - a *start* and an *end* datetimes. Every visualizer is therefore available in two separate versions - one for instants and one for intervals.

Figure 2.1 shows proposed mock-ups for the instants and intervals charts.

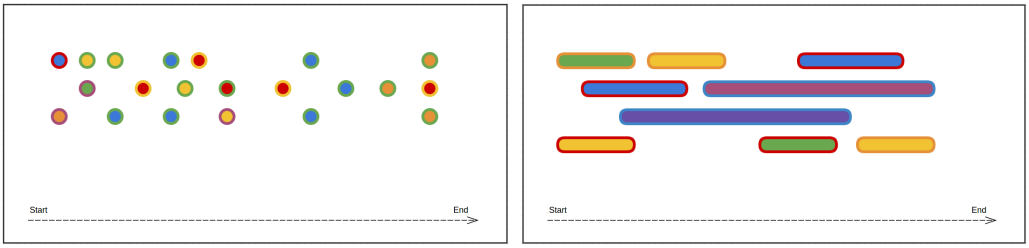


Figure 2.1: Mock-ups of instants(left) and intervals(charts). X-axis represents time; Y-axis is used to minimise overlapping of visualised objects. Different colours are used to divide data by type(fill) and by used properties(stroke).

2.2.1 Instants / Intervals

This is the basic visualizer, showing pure time-based resources. *Instants* visualizes resources with a date-time property. *Intervals* visualizes resources with start and end date-time properties.

Clicking on instants will show the date-time in the info window. Clicking on intervals will show the start and the end of the interval. Both will display information about the selected RDF resource (URI, Label, Comment).

The configuration requirements are:

- Selecting **time range** (begin-end) of visualised intervals. The visualizer is supposed to load time records from that range. Figure 2.2 shows mock-up of the Time Range Selection UI component.
- **Zooming** the chart reduces the range of visualised dates. Note, that unlike the previous feature, zooming does not reload data. Zoom will be available via mouse scroll wheel.
- **Moving the zoom**ed window - as the zoom does not reload data, we can use the reduced time range as a magnified view over the whole loaded time range. Users can move the window by dragging the chart.

The data descriptor for the *instants* visualisation is

```
PREFIX time: <http://www.w3.org/2006/time#>

ASK {
  ?instant time:inDateTime ?dateTimeDescription .
}
```

The data descriptor for the *intervals* visualisation is

```
PREFIX time: <http://www.w3.org/2006/time#>

ASK {
  ?interval time:hasBeginning ?beginning ;
    time:hasEnd ?end .

  ?beginning time:inDateTime ?dateTimeDescriptionBeginning .
  ?end time:inDateTime ?dateTimeDescriptionEnd .
}
```

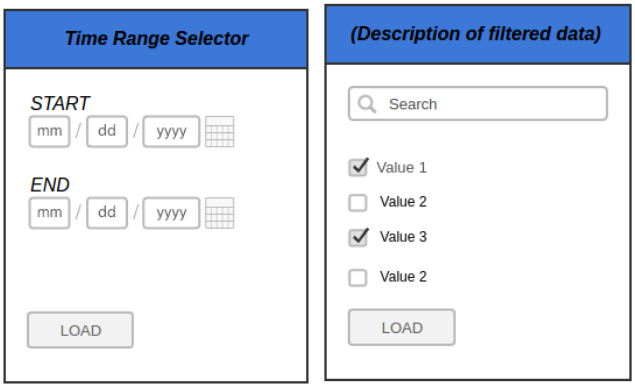


Figure 2.2: Configuration UI components mock-ups. Time Range Selector on the left - a user selects start and end of a time range and loads time records only in that interval. The Resource-Filtering window on the right enables filtering visualised resources. The user selects a subset of available resources and reloads the visualised data according to the selection. Note, that the visualised resources can represent anything in the final visualisation - for example, we will use this component to filter properties used to link loaded resources. For each visualised resource, its Label is shown. The component supports searching amongst visualized resources.

2.2.2 Things with Instants / Intervals

Visualises resources, that have instants or intervals linked to them. Such format of data can represent, for example, a touring exhibition, which has multiple exposition dates/intervals.

Clicking on the instants/intervals shows (in the info window) information about the selected instant or interval and all linked things. Information about property used to link the record to the instant/interval is added as well.

In addition to configurations supported by instants/intervals visualizer, these visualizers have to support:

- Filtering **types** of visualised things - a user can select which types of things do they want to load. Component shown in figure 2.2 will be used for this purpose.
- Filtering **properties** used to link instants/intervals - the user is allowed to filter things linked to instants/intervals using a selected subset of properties. The Resource-Filtering component shown above will be used to support this feature as well.

The data descriptor for the *instant-based* visualisation is

```
PREFIX time: <http://www.w3.org/2006/time#>

ASK {
  ?thing ?hasTemporalAbstraction ?instant .

  ?instant time:inDateTime ?dateTimeDescription .
}
```

The data descriptor for the *interval-based* visualisation is

```
PREFIX time: <http://www.w3.org/2006/time#>

ASK {
  ?thing ?hasTemporalAbstraction ?interval .

  ?interval time:hasBeginning ?beginning ;
    time:hasEnd ?end .

  ?beginning time:inDateTime ?dateTimeDescriptionBeginning .
  ?end time:inDateTime ?dateTimeDescriptionEnd .
}
```

2.2.3 Things with Things with Instants / Intervals

Visualises the next level of abstraction, resources that have things with instants or intervals linked to them. An example of such data format is laws with versions; where each version can have multiple validity intervals.

Clicking on the instants/intervals shows (in the info window) information about the selected instant or interval and all linked things. Information about property used to link the record to the instant/interval is added as well.

In addition to configurations supported by things with instants/intervals visualizer, these visualizers have to support:

- Filtering **visualised things** - a user can select which things do they want to load. Component shown in figure 2.2 will be used for this purpose.
- Filtering **properties** used to link thing with instant / interval - the user is allowed to choose a subset of properties. The Resource-Filtering component shown above will be used to support this feature as well.

The data descriptor for the *instant-based* visualisation is

```
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX time: <http://www.w3.org/2006/time#>

ASK {
  ?outer_thing ?hasAbstraction ?inner_thing .
  ?inner_thing ?hasTemporalAbstraction ?instant.

  ?instant time:inDateTime ?dateTimeDescription .
}
```

The data descriptor for the *interval-based* visualisation is

```
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX time: <http://www.w3.org/2006/time#>

ASK {
  ?outer_thing ?hasAbstraction ?inner_thing .
  ?inner_thing ?hasTemporalAbstraction ?interval.

  ?interval time:hasBeginning ?beginning ;
    time:hasEnd ?end .

  ?beginning time:inDateTime ?dateTimeDescriptionBeginning .
  ?end time:inDateTime ?dateTimeDescriptionEnd .
}
```

2.3 Google Maps

The second group of visualizers work with map-oriented data. The visualizers have a similar structure to the Time Line, introducing a basic visualizer just for coordinates and then linking resources to it. Upon clicking on visualised objects a window containing additional information about visualized resource should appear.

The visualized data can be divided into **pure geo-coordinates** containing only a latitude and a longitude and **coordinates with quantifier** linked to them. Coordinates and Places visualizers are in the first group, while the second one contains Quantified Places and Quantified Things with Places. The first group will be visualised using Google Maps **markers**. The second

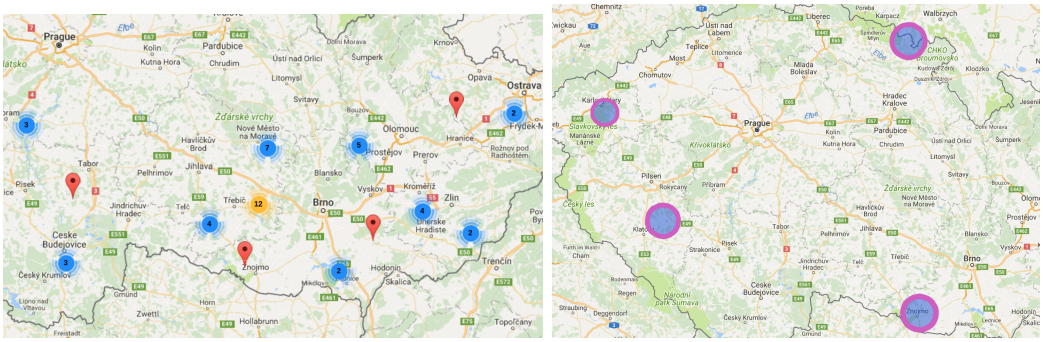


Figure 2.3: Google Maps visualizer mock-ups: Markers (left) and Circles(right) on the map, where coordinates correspond to the Marker position or Circle centre. Circle radius is calculated from the quantifier linked to the viewed resource.

group will be visualized using **circles**, with radius corresponding to size of the linked quantifier.

Figure 2.3 shows mock-ups of Goole Maps visualizers. The Resource-Filtering component (same as for timeline, shown in figure 2.2) will be used to filter and reload data according to the configuration. Markers (on the left) will be used to visualize pure geo-coordinates, while Circles (on the right) will be used to visualise coordinates with quantifiers. Note: 1. circles will also use colors to distinguish different data types and used predicates, 2. markers will support clustering (blue and yellow circles on the visualisation).

2.3.1 Coordinates

The first visualizer requires only longitude and latitude and visualises them on the map using markers and markers clusters. Clicking on markers shows information about the coordinates - with Label and Comment.

The configuration requirements are:

- **Zooming** the map using mouse scroll wheel, like on the regular Google Maps. Zooming out groups markers into clusters, making the visualisation less overcrowded.

- **Moving** the map by dragging it. Works exactly like on the regular Google Maps as well.

The data descriptor for this visualisation is

```
PREFIX s: <http://schema.org/>

ASK {
  ?coordinates s:longitude ?longitude ;
    s:latitude ?latitude .
}
```

2.3.2 Places

A place is defined as a resource with coordinates. Clicking on markers shows information about the place and coordinates - URI, Label, Comment for every visualized resource.

In addition to configurations supported by coordinates, the visualizer has to support filtering **types** of visualised places - a user can select which types of places do they want to load, (e.g. load just cities). The resource filtering component shown in figure 2.2 will be used for this purpose.

The data descriptor for this visualisation is

```
PREFIX s: <http://schema.org/>

ASK {
  ?place s:geo ?coordinates .
  ?coordinates s:longitude ?longitude ;
    s:latitude ?latitude .
}
```

2.3.3 Quantified Places

Like the Places visualizer, but each place is also linked to a quantifier. Data is visualised using circles, where radius corresponds to the linked quantifier. Clicking on circles shows information about the quantifier, place and coordinates - with Label and Comment for each shown resource.

In addition to configurations supported by coordinates, the visualizer has to support:

- Filtering **types** of visualized places - a user can select which types of places do they want to load. Component shown in figure 2.2 will be used for this purpose.

- Filtering **properties** used to link quantifiers - the user is allowed to filter places linked to quantifiers using a selected subset of properties. The Resource-Filtering filtering component shown above will be used to support this feature as well.

The data descriptor for this visualisation is

```
PREFIX s: <http://schema.org/>

ASK {
  ?place s:geo ?coordinates ;
    ?hasQuantifiedAbstraction ?quantifier .

  ?coordinates s:longitude ?longitude ;
    s:latitude ?latitude .

  FILTER(ISNUMERIC(?quantifier))
}
```

2.3.4 Quantified Things with Places

Visualizes things which have quantifiers and places linked to them. Again, data is visualised using circles, where the radius corresponds to the linked quantifier.

Clicking on circles shows information about the thing, quantifier, place and coordinates - with Label and Comment for each shown resource.

In addition to configurations supported by places, the visualizer has to support:

- Filtering visualised **things** - a user can select which things they want to load. Places and Quantifiers will be loaded only for selected things. Component shown in figure 2.3 will be used for this purpose.

- Filtering properties used to link **places** - the user can select which properties are allowed to link places and load data using only those properties. Component shown in figure 2.3 will be used for this purpose as well.

- Filtering properties used to link **quantifiers** - just like places, we will allow the user to choose a subset from the available properties used to link quantifiers and load resources using the selection. Like in the previous configurations, we will use the record-filtering component.

The data descriptor for this visualisation is

```
PREFIX s: <http://schema.org/>

ASK {
  ?thing ?hasSpatialAbstraction ?place ;
    ?hasQuantifiedAbstraction ?quantifier .

  ?place s:geo ?coordinates ;

  ?coordinates s:longitude ?longitude ;
    s:latitude ?latitude .

  FILTER(ISNUMERIC(?quantifier))
}
```

3. Platform overview

The following chapter introduces the platform for the visualizer plugins. We will begin with the Linked Data Visualization Model implementation; After that, we will introduce the LinkedPipes Visualization Assistant, take a look at the use cases and architecture. We will also cover used technologies and standards of the code structure.

Although we will not introduce any new components or functionality in this section, we will point out parts of the platform, where we will need to add new or modify existing application modules.

The last section describes a process of adding new visualizers. We will also provide a summary of the modules, which require adding new components or extending the functionality of its current components (mentioned above). That should give us a solid foundation to use in the Implementation chapter.

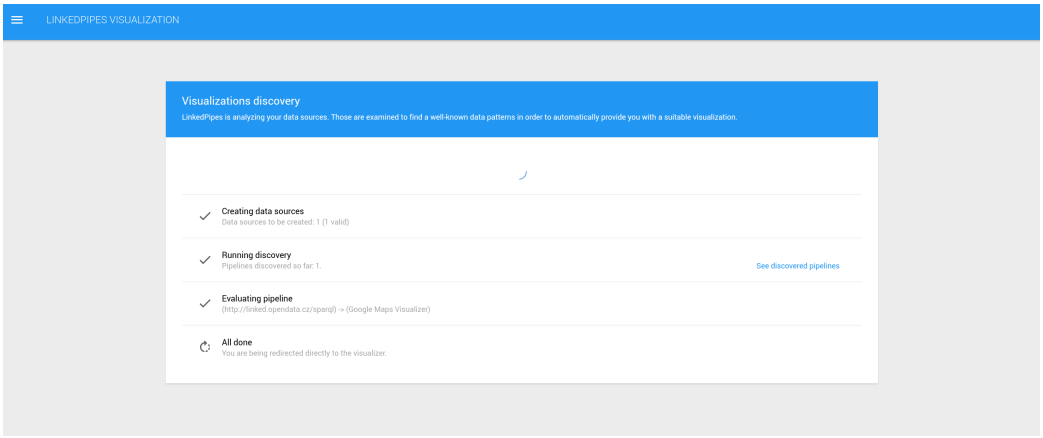
3.1 LDVMi

Linked Data Visualization Model (LDVM) [3] is an abstract model of handling the visualisation process via **pipelines** - the application takes any Source Data (not necessarily an RDF file), applies a series of transformations and then hands the output to a visualizer component, which then produces a visual representation of given data. In addition to that, the model introduces an important concept of **compatibility** - each component of the model is assigned an input descriptor, determining what data it can process.

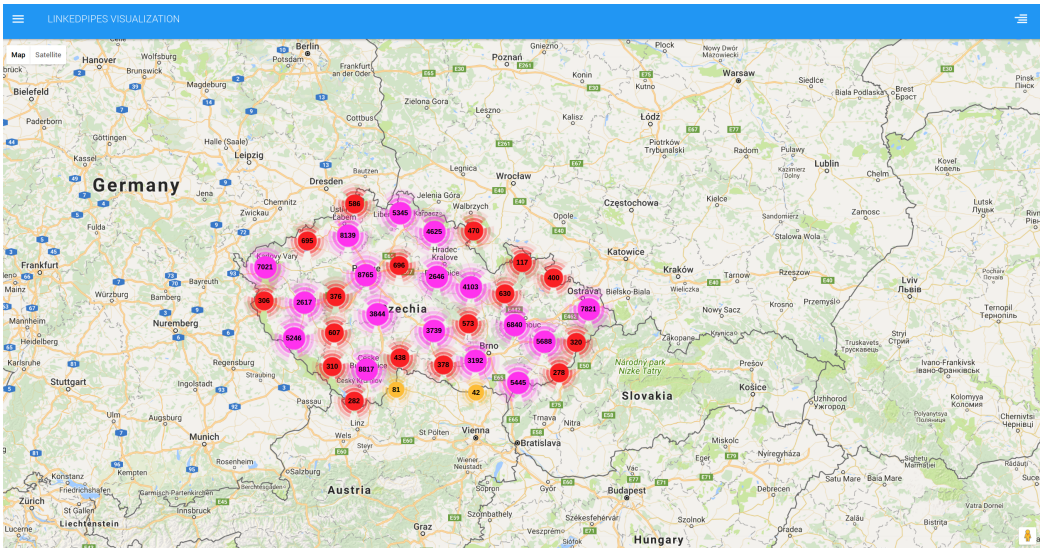
Linked Data Visualization Model implementation (LDVMi) [6] is an implementation of this abstract model. The application itself uses the Model-View-Controller (MVC)[1] model. We will take a closer look at the application architecture in the Assistant part.

After a data source is inserted, the application starts pipeline discovery. Using input descriptors for all visualizers, the application is attempting to combine registered transformations to achieve required data formats.

[1] https://en.wikipedia.org/wiki/Model-view-controller



If more pipelines were found, the user is allowed to choose which does he want to execute. After that, the user is then redirected to the chosen visualisation.



As the purpose of this thesis is to implement visualizer plugins into the Linked Pipes Visualisation Assistant, there is no need to introduce the full internal structure of the LDVMi. However, the Assistant uses some of the original features.

Most of the functionality is shared internally - and since the Assistant is implemented to act as a separate application, the user will not notice that. However, one thing that remains is registering new visualizers.

3.1.1 Registering new visualizers

To add a visualizer to the Assistant, we will have to register it in the LDVMi first. LDVMi requires each component to be a separate RDF resource. We

will need to describe visualizers as **resources**.

```
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema\#> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix ldvm: <http://linked.opendata.cz/ontology/ldvm/> .

@prefix v-googlemaps-quantified-places:
<http://linked.opendata.cz/ontology/ldvm/visualizer/googlemaps-quantified-places/> .
@prefix v-googlemaps-quantified-places-r:
<http://linked.opendata.cz/resource/ldvm/visualizer/googlemaps-quantified-places/> .
```

Then we need to define a visualizer **Configuration**:

```
v-googlemaps-quantified-places-r:GoogleMapsVisualizerTemplate
    a ldvm:VisualizerTemplate ;
    rdfs:label "GoogleMaps quantified places values visualizer"@en;
    rdfs:comment "Visualizes places with values and coordinates"@en;
    ldvm:componentConfigurationTemplate v-googlemaps-quantified-places-r:Configuration ;
    ldvm:inputTemplate v-googlemaps-quantified-places-r:Input ;
    ldvm:feature v-googlemaps-quantified-places-r:GoogleMapsFeature .

v-googlemaps-quantified-places:GoogleMapsVisualizerConfiguration
    a rdfs:Class ;
    rdfs:label "Google Maps quantified places Visualizer Configuration"@en;
    rdfs:subClassOf ldvm:ComponentConfiguration .

v-googlemaps-quantified-places-r:Configuration
    a v-googlemaps-quantified-places:GoogleMapsVisualizerConfiguration ;
    dcterms:title "Default Configuration" .
```

And finally, the **Input Descriptor**. Note the SPARQL ASK query at the end of following code - That is the query used to perform the data compatibility check during the pipeline execution. When implementing visualizers, we will put the input data descriptors, which were introduced in the previous chapter, in that place.

```
v-googlemaps-quantified-places-r:Input
    a ldvm:InputDataPortTemplate ;
    dcterms:title "Data described using RGML vocabulary" .

v-googlemaps-quantified-places-r:GoogleMapsFeature
    a ldvm:MandatoryFeature ;
    dcterms:title "GoogleMaps input descriptor" ;
    ldvm:descriptor v-googlemaps-quantified-places-r:GoogleMapsDescriptor .

v-googlemaps-quantified-places-r:GoogleMapsDescriptor
    a ldvm:Descriptor ;
    dcterms:title "GoogleMaps places with values presence check" ;
    ldvm:query """
        PREFIX s: <http://schema.org/>

        ASK {
            ?place s:geo ?coordinates ;
                ?hasQuantifiedAbstraction ?quantifier .

            ?coordinates s:longitude ?longitude ;
                s:latitude ?latitude .

            FILTER(ISNUMERIC(?quantifier))
     }
    """ ;
    ldvm:appliesTo v-googlemaps-quantified-places-r:Input .
```

Together, this gives us an RDF visualizer component definition. If we compare data descriptors (or simply look at the visualizer name), we will notice that this is actually a definition of Google Maps - Quantified Places visualizer. We will have to create such descriptors for all required visualizers.

Now that we have the definition, we will insert it into the LDVMi platform. Components can be added via LDVMi Home ⇒ Components ⇒ Add New. A screen shown in 3.1 should appear. We will be uploading our visualizer definitions here.

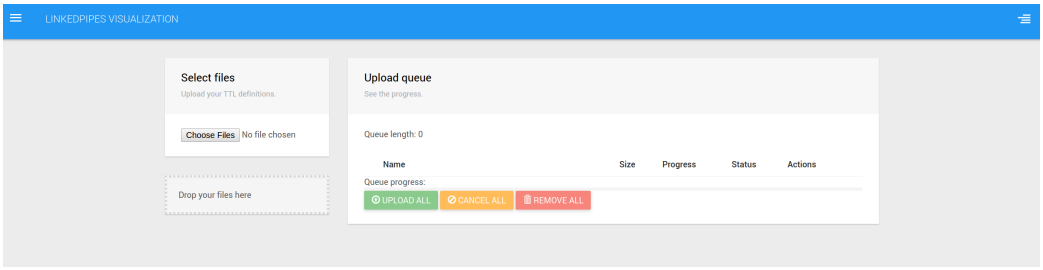


Figure 3.1: LDVMi: Adding components into the application. Visualizers used in the Assistant have to be imported as LDVMi visualizer components.

After a successful upload, the work in the LDVMi application is done. Although we will use many shared features/components, we will consider them parts of the Assistant.

3.2 Linked Pipes Visualisation Assistant

The Linked Pipes Visualization Assistant (or just Assistant) [7] allows users to create, configure and publish visualisations (or views) based on input data sets. The Assistant is integrated into the LDVMi but acts as a standalone application. This section will first show the Assistant from user's point of view. In the second part, we will proceed with the architecture of the platform.

3.2.1 Use cases

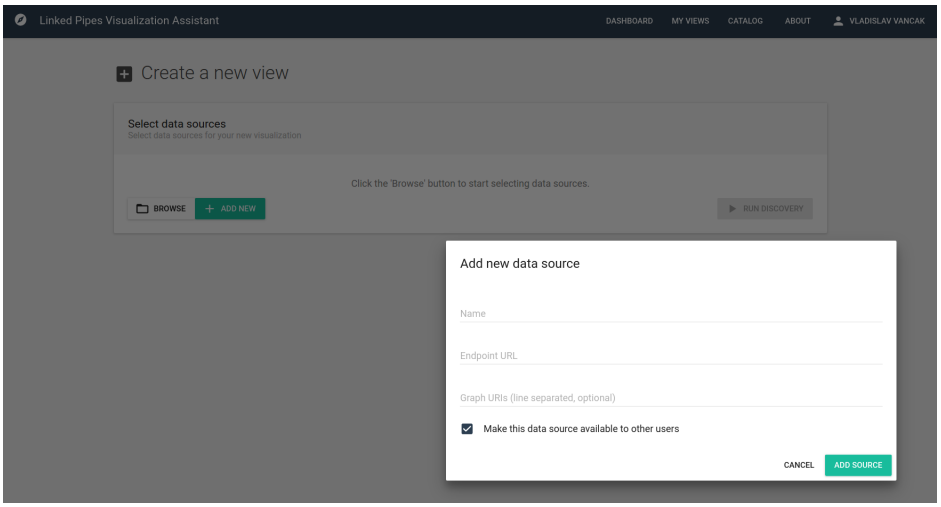
As we have already mentioned, the purpose of the Assistant is to create visualisations of RDF data.

By default, a newly created view is marked as private, where only the creator has access to it. In this state, the creator is expected to configure the visualisation. To make a configured view public, the application introduces **publishing** of created views.

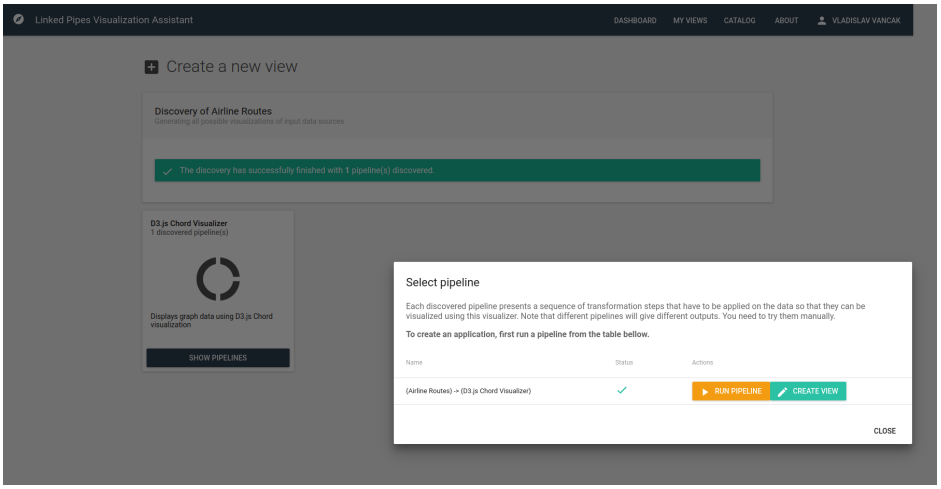
A published view is a visualisation together with a configuration. It is accessible without the need of entering user credentials and from outside of the Assistant (i.e. no Assistant UI). Published views should no longer support significant changes of the configuration.

Creating new views is possible from the home page or from the Dashboard, which allows users to manage all created views.

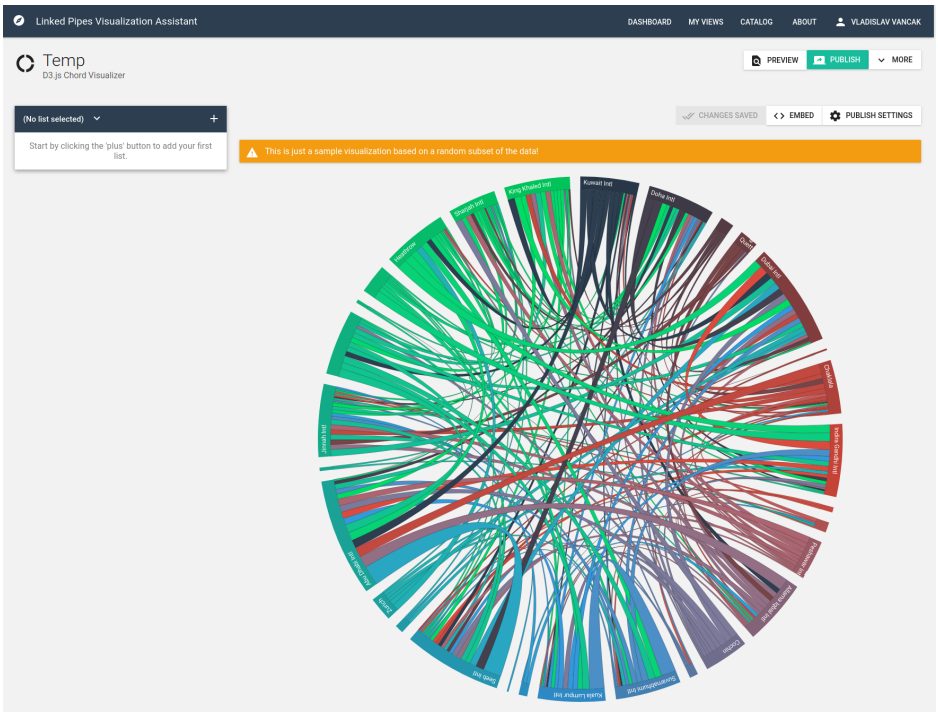
1. Selecting a data source: Users can choose an existing data source (i.e. previously added), or add a new one. A data source consists of an SPARQL endpoint and optional graph URIs.



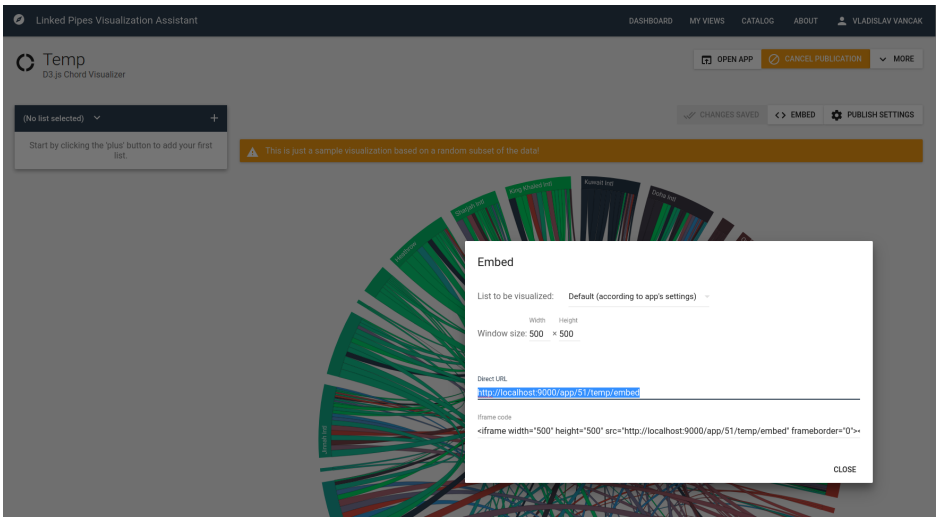
2. A pipeline discovery similar to the LDVMi pipeline determines which visualizers can be used, and offers a selection to the user. Upon selecting a visualizer, the user has to **run the pipeline** and then **create a view**.



3. The user is redirected to the visualizer, where he is expected to configure the view and save the configuration.



4. With the configuration saved, the user can now publish the created view. The application allows accessing the visualisation via direct URL, or a view can be embedded into another webpage.



5. The published view is now accessible without the creator's credentials. A published view also does not contain any Assistant interface. An example of such view is shown in 5

Figure 3.2: Result visualisation. The Assistant user interface is no longer needed. The configuration created in the Assistant is used and can not be changed from this view. The view is accessible without the creator's Assistant profile. The visualisation can also be embedded into web pages.

3.2.2 Architecture

Like the LDVMi, the Assistant uses the server-client model, where the client side runs in a web browser, while the backend runs on a (remote) server.

The frontend is implemented using **Single-Page Applications** (SPA)[2]. In our case, that means that the frontend acts as a standalone application running in the web browser. After a webpage is displayed, the application uploads the page dynamically, using REST API calls[3] to communicate with the backend running on the server.

As the Assistant itself is integrated into the LDVMi, it preserves the original MVC structure. The architecture diagram is shown in figure 3.3.

Models are a source of data. They provide queries to get data from SPARQL endpoints and data extractors to extract valid data from the returned table.

Controllers take care of all API calls. When such call is received, a corresponding controller is called to handle the request. Controllers have

[2]https://en.wikipedia.org/wiki/Single-page_application
[3]https://en.wikipedia.org/wiki/Representational_state_transfer

access to all models and views, as they use them to provide data and user interface to the application.

Views provide the user interface, i.e. the Frontend.

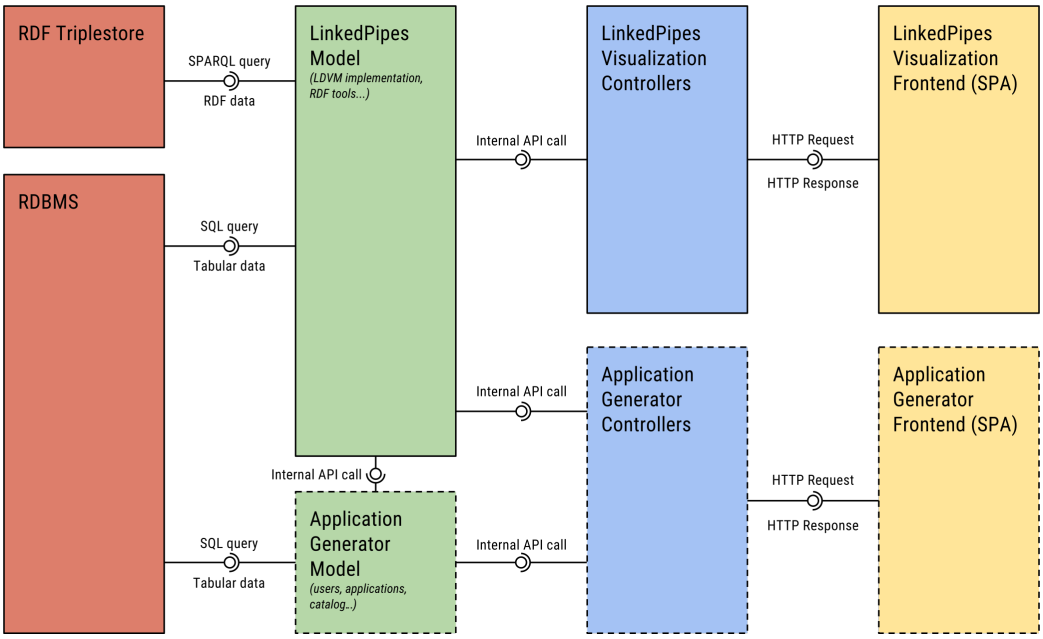


Figure 3.3: Assistant architecture. Blocks with solid borders are part of the original LinkedPipes Visualization, while dashed borders are parts of the Assistant. Note, that in the Assistant thesis, a term Application Generator is used to refer to the Assistant. Image source: The original Assistant thesis [7]

3.3 Technologies

The following section provides an overview of used technologies. We will start with the backend of the application and move onto the frontend.

3.3.1 Back-end

The backend implementation is fairly easy - its functionality consists mainly of creating SPARQL queries, executing them, extracting result data (possibly transforming it) and sending it back to the client.

This part of the application is written in Scala[4] and powered by the Play framework[5]. To work with RDF data, Apache Jena[6] is used.

The Assistant-only parts can be distinguished by separate package names. Controllers use the *controllers.assistant*. Models use *model.assistant* as their root packages.

The only place where the LDVMi and the Assistant can not be properly separated is *routes configuration*. This part of the application specifies target URIs and HTTP methods [4] available for REST API calls, along with mapping routes to the controller methods. These configuration cannot be separated, as there is only one root URL. Therefore the methods are separated just visually (i.e. two "sections" of the configuration file), where the assistant URIs usually contain */assistant* in their path.

3.3.2 Front-end

The frontend is a bit more complicated from the technological point of view, as it combines many of currently available web technologies to correctly load and visualise input data. As we will only implement the visualizer plugins, we will list the most relevant packages. To get the full frontend development stack, see the Assistant implementation thesis [7, attachment A].

The frontend is implementing in JavaScript, using ECMAScript 2015 (ES6) [5] standard. To manage package dependencies, the application uses npm[7] package manager. The most important libraries used are Redux, React and Reselect.

Redux

Redux[8] is a library for managing the application state. An application is expected to create:

Actions represent "what is happening"

Reducers define how the state is changed when Actions are called

Store holds the application state and updates it.

[4]https://www.scala-lang.org/
[5]https://www.playframework.com/
[6]https://jena.apache.org/
[7]https://www.npmjs.com/
[8]http://redux.js.org/

The application has access to Action Creators, which *Dispatch* corresponding actions in the Store. The Store then sends prevoius state to Reducers and updates its Current state with result from the Reducers (i.e. Reducers "create" the new state). Application can always access current State from the Store via *Get State* function call. The data flow is shown in figure 3.4

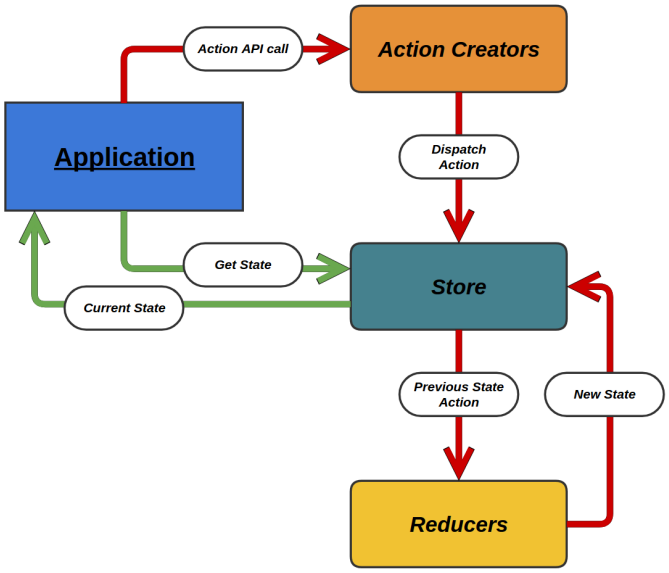


Figure 3.4: Redux data flow: Application uses Action API calls in Action creators to trigger a state update (red arrows). Reducers are used to update previous state. Store provides access to the current state (green arrows).

React

React[9] is a library for building user interfaces. React introduces **Components**, which implement:

1. *render()* method to specify what the component looks like
2. (optional) set of methods, which are called when the component status is changed (pre-defined method names, e.g. *componentWillMount*)
3. (optional) set of methods, which are called when the component properties are changed (pre-defined names, e.g. *componentDidReceiveProps*)

[9]https://facebook.github.io/react/

React components look even better when used together with **JSX**[10] - a preprocessor, allowing to use XML tags in the code instead of generating XML trees through method calls.

An example Component (with use of JSX) is shown in figure 3.5

Reselect

Reselect[11] is a library that simplifies extracting important data from the application state. Instead of working directly with the application state, we create **Selectors**, which provide only the information we need. This provides

1. Direct access to a state variable, even if it is stored deep within the state structure
2. No access to other state variables - thus making the application less error-prone
3. With React Components monitoring it's props, events like *componentWillReceiveProps* will be triggered only on relevant state changes

The library is usually used together with React-Redux. In figure 3.6 below a diagram describes structure of such application. In the actual implementation, components are usually separated in two files: A **Container** file contains definition of a React Component, while a **Duck** file contains definitions of Actions, Reducers and Selectors.

D3.js

D3.js[12] is a library that simplifies creating HTML, SVG and CSS from data. The library consists of many useful features including **Scaling, Transformations, Axes Creation** and many others.

React Google Maps

react-google-maps[13] is a simple wrapper over the Google Maps APIs[14], allowing to create visualisations on the Google Maps[15].

[10] http://buildwithreact.com/tutorial/jsx
[11] https://github.com/reactjs/reselect
[12] https://d3js.org/
[13] https://github.com/tomchentw/react-google-maps
[14] https://developers.google.com/maps/
[15] https://www.google.cz/maps

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}
```

Figure 3.5: Example of a React Component: Note render() and componentDidMount() methods - those are called as the Component state is changing. React with JSX allows us to use xml tags directly (like the HTML tag) instead of generating them in multiple method calls. Source of the example is the official React guide at https://facebook.github.io/react/

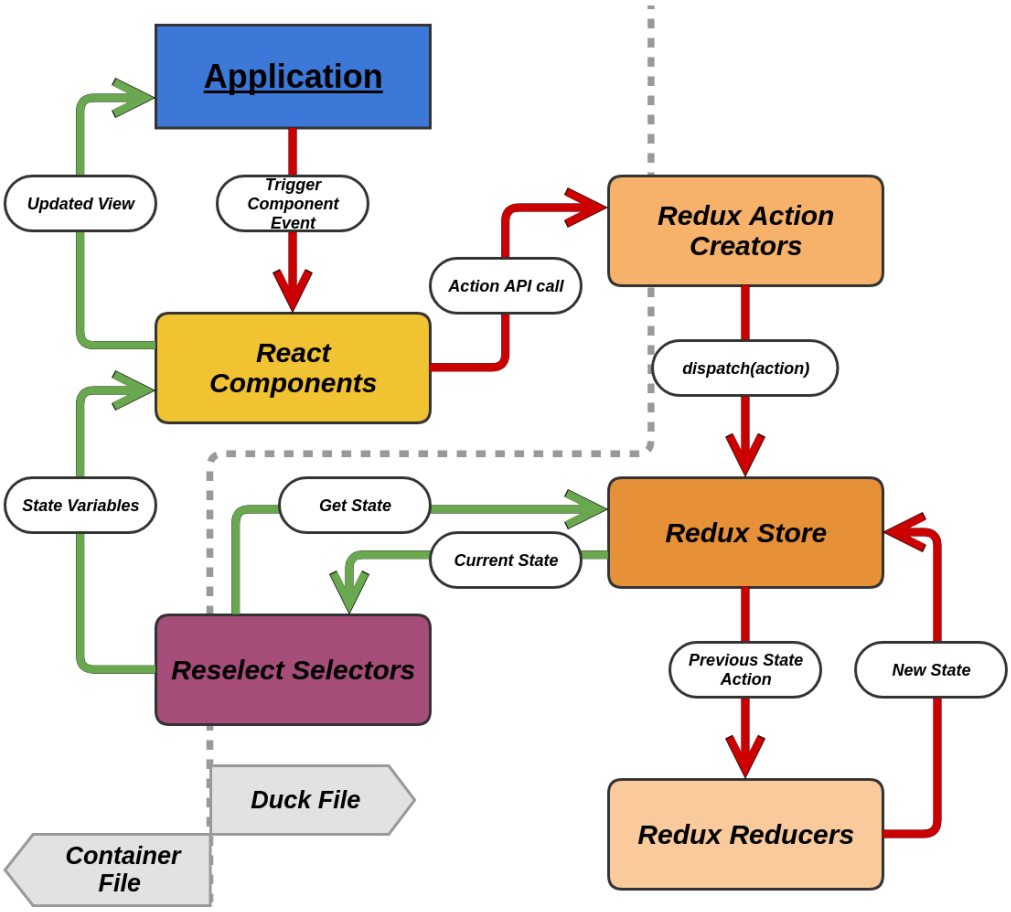


Figure 3.6: React-Redux with Reselect: User (or Application) triggers React Component event. The diagram shows how an updated view is delivered. Note the division into two files - a Container file contains a Component implementation, which uses Selectors to access state variables and Action API calls to trigger state updates. Ducks file contains Actions + Reducers + Selectors.

3.4 Visualizers

In this chapter, we will take a closer look at the visualizers. We will start with a visualizer architecture, we will look at currently implemented visualizers, and we will finish by determining step for integrating new visualizers.

3.4.1 Visualizer Architecture

In the Assistant architecture overview, we have already shown, that the Frontend is a set of Single-Page Applications. Furthermore, as we can see in the Assistant use cases, each visualizer works in 2 phases: Configuration phase and Published phase.

As this feature suggests, each visualizer has to be in two separate SPAs - one with a Configurator interface and one with an Application interface.

1. **Configurator** interface is integrated into a common Platform SPA. This SPA contains Main configuration components (like the top toolbar) and all visualizer configurator interfaces.

2. **Application** interface is integrated into a separate SPA used to display published views of the corresponding visualizer.

Architecture of a common visualizer can be separated into

Components - React Components that do not request state updates or data loading

Containers - React Components that contain business logic 3.6

Ducks described in 3.6, contain definitions of Redux actions, reducers and Reselect selectors

Misc contains various utilities, styles etc.

Pages define Configurator and Aplication user interfaces

Newly implemented visualizers are strongly recommended to follow the described standard of used modules. As the structure covers all requirements for the new visualizers, this is the structure of all new visualizers as well.

3.4.2 Available Visualizers

Currently there are two visualizers available:

D3.js Chord shown in examples above (figure 5). The visualizer shows relationships between entities and their strengths. To get more information about this visualizer, look at the Assistant thesis [7, Section 6.1]

Google Maps Visualizer works with map-oriented data and visualizes them using Google Maps Markers. To get more information about this visualizer, look at the Assistant thesis [7, Section 6.2]

3.4.3 Adding new Visualzers

As we have already covered important used technologies and how the application works from the architectural point of view, we will now focus on how to add new visualizers.

The Assistant thesis [7, Section 5.5] contains a guide to integrating new visualizers. The guide shows, where to add new components and how to integrate those components into the platform.

To add a new visualizer, we will have to Implement:

1. A **visualizer definition** and add it as a LDVMi component (shown in 3.1).
2. Backend **model** with SPARQL queries and Data extractors (i.e. modules for parsing SPARQL query results)
3. Backend **controller** for handling API calls from the client
4. Frontend visualizer **Configurator** and **Application** Single-Page Application Components.

Configurator and Application interface can be further divided into three parts:

Data Access - Asynchronous API calls to the Backend

Configurations - Allow users to restrict visualised data

Visualisation - Core of the visualizer, e.g. a Chart or a Diagram

Routing - Registrating visualizer routes into the Assistant frontend (so that the Assistant can redirect users to our visualizer).

When implementing those steps, developers are expected to keep the standards described in the common Visualizer architecture.

4. Implementation

This chapter provides the implementation overview of the required features. We will start with Labels and Comments and then move onto the Visualizers. We will cover the overview of TimeLine and GoogleMaps visualizers. At the end of this chapter, we will focus on components for filtering values and full-text search; We will finish by proposing ways to handle large data.

4.1 Labels & Comments

The Linked Pipes Visualization Assistant already supports Labels described in requirements. This leaves just Comments, where the architecture is very similar. First, we will have to solve multiple possibilities for data location. Secondly, we will have to handle dereferencing, and finally, we cannot forget about the language support.

4.1.1 Extracting values

As we have already mentioned in the requirements, we have to try and extract data using multiple properties. For that, we have to solve missing values - SPARQL *SELECT*, by default, has to match all triplets in its *WHERE* section. Luckily, this behaviour can be omitted via the *OPTIONAL* keyword. A triplet marked as optional will be matched if a value is available. If no value matches the pattern, variables stay unchanged (i.e. undefined, by default) - instead of returning no results for the whole query.

To ilustrate the SPARQL OPTIONAL pattern, the following examle shows the Comments query, where the (resource URI) is replaced by resource URIs before executing the query:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX dct: <http://purl.org/dc/terms/>

SELECT DISTINCT ?rc ?dd ?sd ?sn
WHERE {
   OPTIONAL { <(resource URI)> rdfs:comment ?rc . }
   OPTIONAL { <(resource URI)> dct:description ?dd . }
   OPTIONAL { <(resource URI)> skos:definition ?sd . }
   OPTIONAL { <(resource URI)> skos:note ?sn . }
}
```

4.1.2 Prioritising

The requirements defined a concept of predicate priorities - as the user can be shown only one value of a Comment, we have to be consistent in choosing which value to display. In our visualizers, we view the value, which is linked to the resource using a predicate with the highest priority.

This feature is implemented in the Comment data extractor in the Back end *Models*. As we get results of the SPARQL queries, we try to extract values in the specified order and return first valid value found.

4.1.3 Dereferencing

Dereferencing - i.e. looking up resource URI using HTTP requests - is also a part of getting the values of Comments. The pattern is straightforward (and already described in requirements) - Try to find values in the input data source, if none were found, try dereferencing.

The platform has already supported dereferencing URIs. The algorithm requires the resource URI and an SPARQL query for extracting the value from an RDF graph as its input. The steps are:

1. Send HTTP GET request to the resource URI
2. Specify return type to Turtle ("Accept:text/turtle" header)
3. Create an RDF Graph from the result
4. Run the SPARQL query against the created graph.
5. Return results of the SPARQL query.

As we already have the SPARQL queries for Labels and Comments defined from when we were trying to get values from the input data sources, we use the same queries in dereferencing.

4.1.4 Language Support

To implement the language support, Data Extractors have one more feature. In addition to simply extracting values, we monitor available languages, and instead of returning one value, we return one value for each language found. This way, the application loads all available languages into the front end, and the user can select his preferred one.

Figure 4.1 shows us the algorithm and separation of the logic into queries and data extractors.

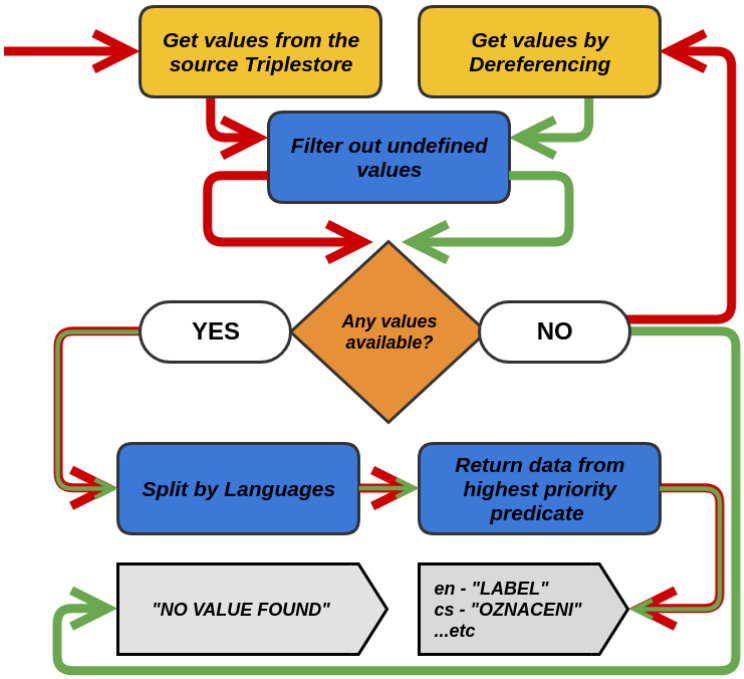


Figure 4.1: Algorithm for getting values of Labels and Comments. Handles multiple possibilities for data source locations (Triplestore or Dereferencing), multiple possibilities for value locations in Graph (multiple predicates) and Language support. Blue parts are implemented in data extractors, while yellow parts are implemented by SPARQL queries and by querying various data sources.

4.2 Visualizers

This section covers implementation details of the visualizers. We will start by providing an overview of the component for filtering resources. We will continue by the colour support. The next subsection will show, how the application handles large input data and finally, we will get to the implemented visualizers - where we will cover the charts, loading of data and provide a quick overview of visualizer's components.

4.2.1 Filtering resources

To allow filtering of resources (which is needed in most of the visualizer configurations), a *RecordSelector* component was implemented and can be seen in 4.2. For a given set of RDF resources, the component supports:

Showing resources - by showing *Labels* for resource URIs.

Searching resources - the search is case-insensitive and searches for the input string in available labels.

Selecting resources using checkboxes. The component requires functions *onResourceSelcted* and *onResourceUnselected* to be specified and uses them to notify the rest of the application.

To make the component smaller, a concept of **Pages** was introduced, where the component shows a maximum of ten resources per page. Buttons *next page* and *previous page* support navigation between pages.

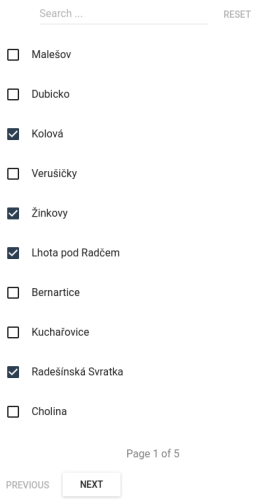


Figure 4.2: RecordSelector component for filtering resources. Shows labels for each of the resources. Allows searching amongst the resources and selecting them. When a resource is selected, the component notifies the application via supported functions. Resources are split into pages to limit maximum component size.

4.2.2 Colours

The visualizers also require a concept of colours for different visualised resources. The algorithm for getting a colour to display is:

1. Try to find colour corresponding to given resource (represented by its URI) in the colour map

2. If the URI *is not* in the Colour map, generate a new colour and add it to the map, together with the URI

3. If the URI *is* in the Colour map, return the corresponding colour.

Note, that this logic is implemented using react-redux, with the map stored in the application state and updated via Redux actions. The colour map is a simple Map, with keys representing URIs (i.e. String) and values being colours (i.e. Colour value).

4.2.3 Handling large data

As the input data sets could be potentially large, we will have to introduce a few features/concepts to limit the amount of loaded data. Otherwise, some of the API calls could hit timeouts or cause other errors due to their size.

Limits

Limits eventually get transferred to LIMIT clauses in SPARQL queries. The main idea is that user specifies, how much data does he want to load. More data means slower application, while less data means less precise visualisations. On the other side, by limiting visualised data, the visualisation chart can become clearer or less overstuffed. To show, how much data could be visualised (compared to how much is actually visualised), we will show *"Loaded (visualised) of (total available) records"*.

Although this provides significant value to the user and makes the visualisations faster, the implementation is a bit tricky.

First of all, getting the total count of available data requires a different query - hence each query used by visualizers has two versions (one SELECT (data) and one SELECT (count)). This feature also requires two versions of API calls from the client to the server.

Secondly, getting that number can potentially be slow, while the value it provides is not that important compared to the visualisation part. Therefore the count data is loaded asynchronously and *"Loading count"* is shown while the value is not available.

Data Levels

Consider an RDF graph of a following structure:

```
<http://example.org/#ObjectA>
        somePrefix:Property1 <http://example.org/#ObjectB>.
<http://example.org/#ObjectB> ;
        somePrefix:Property2 <http://example.org/#ObjectC>.
```

When loading data from this structure, the obvious option is to load everything together in one query. However, when there is a need to load B-C and A-B relations separately, we can split the loading into two requests. We will call the A-B, and B-C relations *levels*, where the B-C relation represents the first level and A-B represents the second. Figure 4.3 shows this structure.

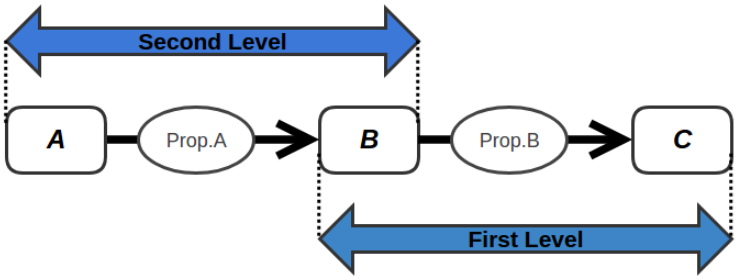


Figure 4.3: Splitting a data structure into levels.

The first reason to load the data separately is *reusability*. If various parts of a program need to access different levels of data, we can simply load the first level without having to load the second one.

The second reason to load the data in levels is a *need to restrict* the loaded data. Imagine, that we want to specify, which of the possible predicates are accepted on the first level. Instead of reloading all records, levels allow us to reload only the first level and simply drop the second level records without corresponding first level records.

To keep the loaded records equivalent to records loaded by a single request, levels have to support:

- Levels from 2 upwards have to make sure, that there are data available in the lower levels. To achieve this, we use the SPARQL *FILTER EXISTS* construct.
- All levels except the top one have to support restricting loaded records. Using our diagram, this means that the second level *loads A-B* relations; the first level then *restricts loaded data to the B* records from the second

level and then *loads B-C* records. To achieve this, we use the SPARQL *VALUES* construct.

An example of an SPARQL query loading such levelled data is shown in figure 4.4

```
SELECT ?a ?b WHERE {
  # load data in this level
  ?a somePrefix:some_property ?b.

  # make sure lower levels contain data
  FILTER EXISTS {?b somePrefix:other_property ?c}

  # restrict loaded resources using data from an upper level
  VALUES ?a {<(resource URI 1)> <(resource URI 2)> ...}
}
```

Figure 4.4: SPARQL query for loading leveled data. Levels make sure, that data in lower levels exists (FILTER) and support restricting loaded data (VALUES) using resources loaded by upper levels.

When we take a closer look at the data descriptors for our visualizers, we can see several patterns for levels. TimeLine will be split into Things ⇒ Things ⇒ Instants/Intervals, while GoogleMaps will be divided into Quantified things ⇒ Places ⇒ Coordinates and Quantified places ⇒ Coordinates.

Restricting loaded records in levels comes with an another problem - API request (client → server) can be potentially large, as we have to supply URIs of loaded resources. We will handle this by introducing *batches*.

Batches

After observing usual scenarios in the application, most of the large API requests are usually generated using a collection of objects. To reduce the size of such API calls even more, the application supports batching requests, achieved by the following steps:

1. Instead of a direct API call, create a request template - a request without data from the collection

2. Generate several API calls using parts of the collection - *"batches"*

3. Collect server responses and compose them into one response

As we have already mentioned, this is useful when implementing requests from data levels (introduced above). To load records from a lower level, we will have to support resource URIs loaded in the upper level, leading to potentially large queries. Splitting the requested URIs into multiple batches can solve the problem quite efficiently.

With Limits and Batches, the application can handle virtually any size of input data. However, since web applications are not among the best-performing ones (as they run in the browser), users are expected to be smart (or be patient while waiting for the data to load/for the chart to update).

4.2.4 Application layout

All visualizers in the application have to support two layouts - the Configurator and the Application, where the latter one does not have the main configurations (like filtering values, setting limits etc.). Both our visualizers share the same User Interface layouts, shown in figure 4.5 below.

The Configurator layout has the main configurations available on the left side of the view. To enable full-screen visualisation, the window is retractable. To hide it, use a cross in the upper left corner, to view it again, use the *CONFIGURE* button. This functionality is implemented in *ConfigurationToolbar* component.

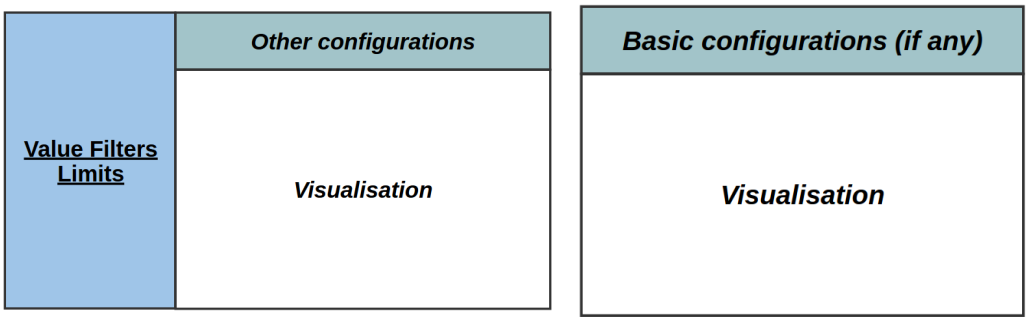


Figure 4.5: Configurator (left) and Application (right) UI layout. Used by both TimeLine and Google Maps.

4.2.5 Time Line

This section covers the implementation of TimeLine visualizers. We will start with the chart, go through data loading and finish with components overview; The end of the chapter covers use cases.

Chart implementation

As the current Assistant visualizers do not contain such visualisations, the Time Line is a newly implemented chart, where D3.js libraries are heavily used in the implementation. Mockups of the chart have already been shown in chapter 2, figure 2.1. The real implementation of the charts can be seen below, in the figure 4.7.

Input data have to either contain a start-end combination for intervals visualised as rectangles, or a date-time for instants visualised as circles.

The chart supports zooming the visualisation and moving the zoomed view through the whole chart - both described in the requirements. A mouse wheel controls zooming, while dragging the chart moves the zoomed window.

The chart itself is two-dimensional, where the **x-axis** represents *time*. The range is determined by the configuration. To prevent overlapping of visualised objects, the chart will try to distribute visualised intervals on up to ten *levels* on the **y-axis**.

Data Supply

Instead of loading all data in one request, the visualizers use the concept of levels from the previous section. The following diagram shows such levels in timeline-oriented data (figure 4.6)

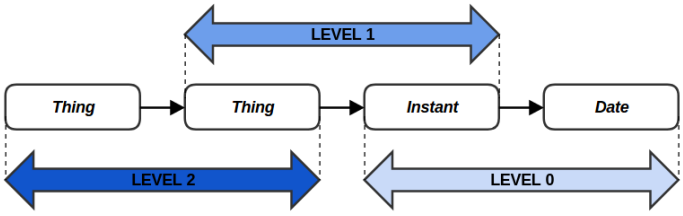


Figure 4.6: Data Levels in Time Line Visualizer. Instead of loading all data in single request, we will load them in up to three levels. The levels work together to maintain the functionality of loading data in a single request.

After a closer look at input data descriptors we will notice that the visualizers have the following structure:

Instants/Intervals visualise level 0 records

Things with Instants/Intervals visualise levels 0 and 1

Things with Things with Instants/Intervals visualise levels 0,1,2

We will be loading levelled data in **level loaders**. As explained in the section above, to maintain the illusion of loading data in a single request, the loaders work together. Each level loader has to support restricting of loaded resources and every loader has to make sure, that there are data in the levels below.

We need 4 level loaders for timeline visualizers - Instants (level 0), Intervals (level 0), First Level and Second level loaders are merged (i.e. not separated for instants and intervals).

Code structure

With a working chart, we now have to look at the rest of visualizer's implementation. We need components for loading data, configurations and for additional features like viewing count of visualised resources (described in Limits section above). In addition to that, visualizers have to be properly registered into the application, so we have to provide application routes (i.e. URL where the visualizers live from the user's perspective).

The visualizers' implementation consists of following modules:

Data Loaders take care of loading data, using levels introduced in the previous section. They are implemented as application's Containers (i.e. React Components that load data using React-Redux library). They also contain most of the configuration logic - filtering values and selecting the loaded time range.

TimeLine Chart containing the visualisation. It is implemented in Misc and used via two separate application's Components (i.e. React Components that do not load data) - one for Instants and One for Intervals. The chart supports zooming and moving the view.

Count viewers and the *Info window* complete the list of visualizer components, i.e. React components without requests for additional data.

Pages contain definitions of visualizers' main user interfaces - one for each of the six visualizers. The components are used as both Configurator (with a configuration toolbar) and Application (without the configurations) user interface - where the components are rendered according to the provided properties.

Bundles contain modules necessary for integrating visualizers into the platform, mainly for routing and compilation purposes.

Configurations

As we have mentioned before, Data Loaders take care of required configurations. Here is a list of configurations supported by the implemented data loaders.

- *Level 0 loaders* cover limiting start and end date of loaded resources, i.e. the configurations required by Interval/Instant visualizers
- *Level 1 loader* covers 1. Filtering types of visualised resources, 2. Filtering available predicates (used to connect instants), i.e., the configurations required by Things with Interval/Instant visualizers
- *Level 2 loader* covers 1. Filtering visualised resources, 2. Filtering available predicates (used to connect level 1 things), i.e., the configurations required by Things with Things with Interval/Instant

Use cases

The visualisation usage is quite straightforward. Using the configuration toolbar, filter the allowed properties, types, time range and limits. Confirm the choice by *LOAD*-ing the selection. *RESET* button can be used to clear the selection and load all data. The chart itself can be zoomed by scrolling the mouse wheel. A zoomed view can be moved by dragging the visualisation sideways. Clicking on the objects shows date for instants, start and end for intervals. Information about linked resources (up to level 2) is shown as well. For each visualised resource the view contains its Label and Comment. Screenshots of the Configurator user interface is shown in figure 4.7

4.2.6 Google Maps

This section covers the implementation of Google Maps visualizers. We will start with the Chart, cover data loading and finish with components overview. We will finish with the use cases.

Chart

A basic chart was already a part of the Assistant and is used by the Google Maps visualizer. We use this chart in our visualizers for viewing **markers**. For the visualisations of quantified records, we have added a new chart, where **circles** are drawn instead of markers.

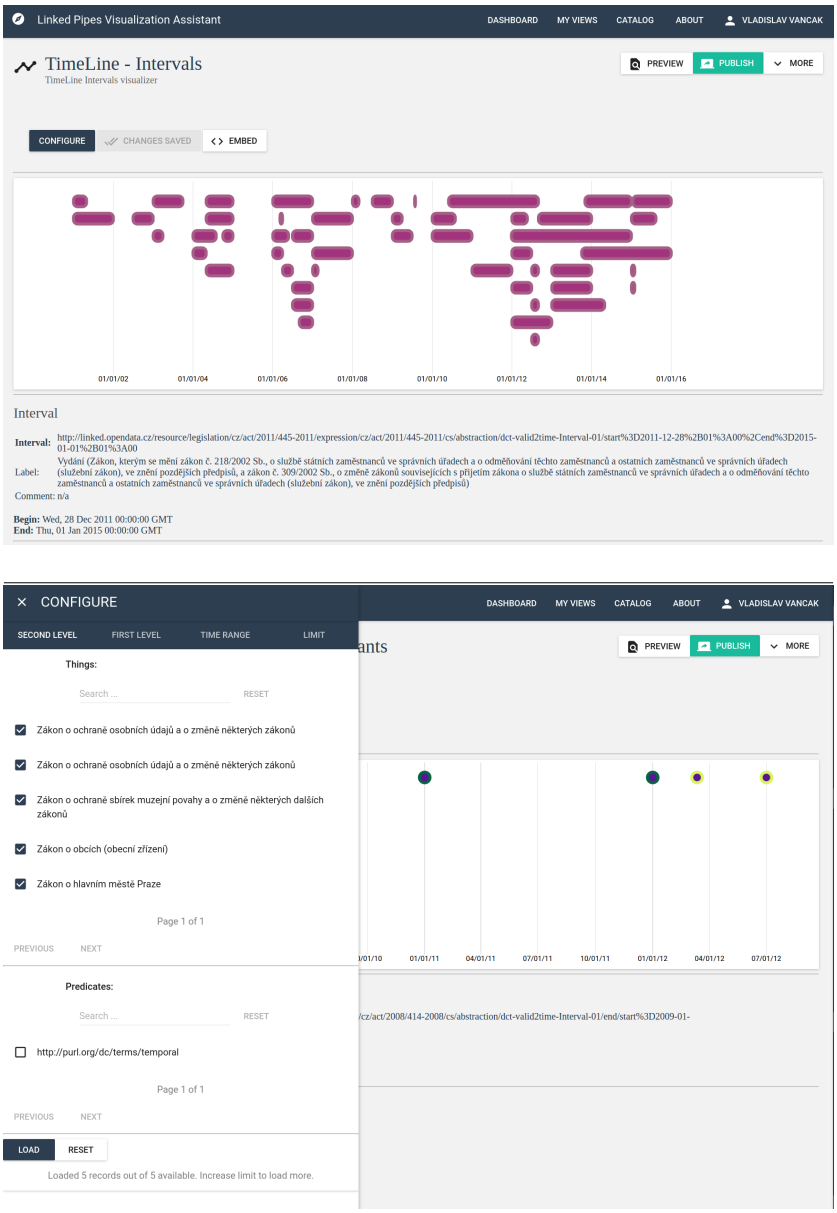


Figure 4.7: Time Line visualizers. On the top view, the configuration toolbar is hidden. We can see the buttons to show the configurator toolbar, save the configuration, and embed the view (if the view is published). Publishing the view is available via a toolbar in the upper right corner. In the top view, we can see the information window (below the timeline chart) containing all available resources linked to the selected interval. The bottom view shows the configuration toolbar and visualised instants.

Input data contains geo coordinages, represented by a *latitude* and a *longitude*. For quantified visualizers, the circle *radius* corresponds to a quantifier value.

The map can be moved using a mouse scroll wheel (or buttons in the bottom right corner). We can also move the visualised area by dragging the map.

Clicking on the visualised records shows all information about all resources linked to them in an info window. Multiple info windows can be open at the same time.

Data Supply

Data supply and architecture of the data-loading components is similar to the timeline: we are splitting our data into multiple levels to allow components to be reusable. The diagram in figure 4.8 shows the Google Maps data levels.

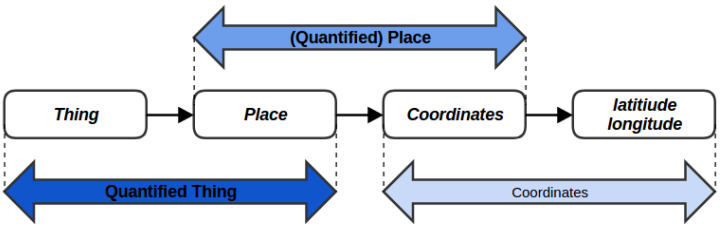


Figure 4.8: Data Levels in Google Maps Visualizer. Instead of loading all data in single requests, we load them in up to three stages. Note, that Place can have a quantifier (in quantified places visualizer). Place without coordinates is loaded in places visualizer.

Like in the TimeLine, each of the levels is be loaded by a separate **data loader**. We need four such loaders - Coordinates, Places, Quantified Places and Quantified Things.

Code structure

The code structure is similar to the TimeLine:

Data Loaders take care of loading data, using levels introduced in the previous section. They are implemented as application's Containers (i.e. React Components that load data using React-Redux library). They also contain configurations that require filtering resources.

Google Maps Chart is a wrapper over *react-google-maps* libraries. Two separate charts exist, one for markers and one for circles. Both are implemented as application's Components.

Count viewers and the *Info window* complete the list of visualizer components (i.e. React components that do not load data).

Pages contain definitions of visualizers' main user interfaces - one for each of the six visualizers. Like in the Time Line, the components can be used as both Configurator and Application user interface, according to the provided properties.

Bundles contain modules necessary for integrating visualizers into the platform, mainly for routing and compilation purposes.

Configurations

As we have mentioned before, Data Loaders also take care of required configurations. Here is a list of configurations supported by the available data loaders.

- *Coordinates loader* - all configurations required by the Coordinates visualizer are supported directly by the map chart. Therefore the loader does not support any configurations.

- *Places loader* covers filtering types of visualised places, i.e., the configuration required by the Places visualizer.

- *Quantified Places loader* covers 1. Filtering types of visualised places, 2. Filtering available predicates (used to connect quantifier), i.e., the configurations required by the Quantified Places visualizer.

- *Quantified Things loader* covers 1. Filtering visualised resources, 2. Filtering predicates used to link places, 3. Filtering predicates used to connect quantifiers, i.e. the final configurations, required by the Quantified Things with Places visualizer.

All other required configurations (zooming and moving the zoomed view) are supported by the chart. Therefore we have the chart; we have the configured data loaded - the visualizers are ready.

Use cases

The visualisation usage is quite straightforward. The *configuration toolbar* contains all *filtering* for visualised data levels ((quantified) places and quantified things) and allows changing the *limit*.

The user can select the allowed properties or resources and then *LOAD* the levelled-data according to the selection. *RESET* can be used to clear the selection and load all data.

The chart itself can be zoomed by scrolling the mouse wheel. A zoomed view can be moved by dragging the map. Visualisers group markers into marker clusters - circles with a number of hidden markers; Clicking on that circle zooms to the location of the clustered markers. Visualised circles change its size during zooming - a circle should never be too small or cover user's whole screen.

Clicking on the objects shows coordinates and information about linked resources (up to level 2) are shown as well. For each visualised resource the view contains its URI, Label and Comment. Information is visualised in text bubbles over the map. We can have multiple information windows viewed at the same time. Screenshots of the Configurator user interface is shown in figure 4.9

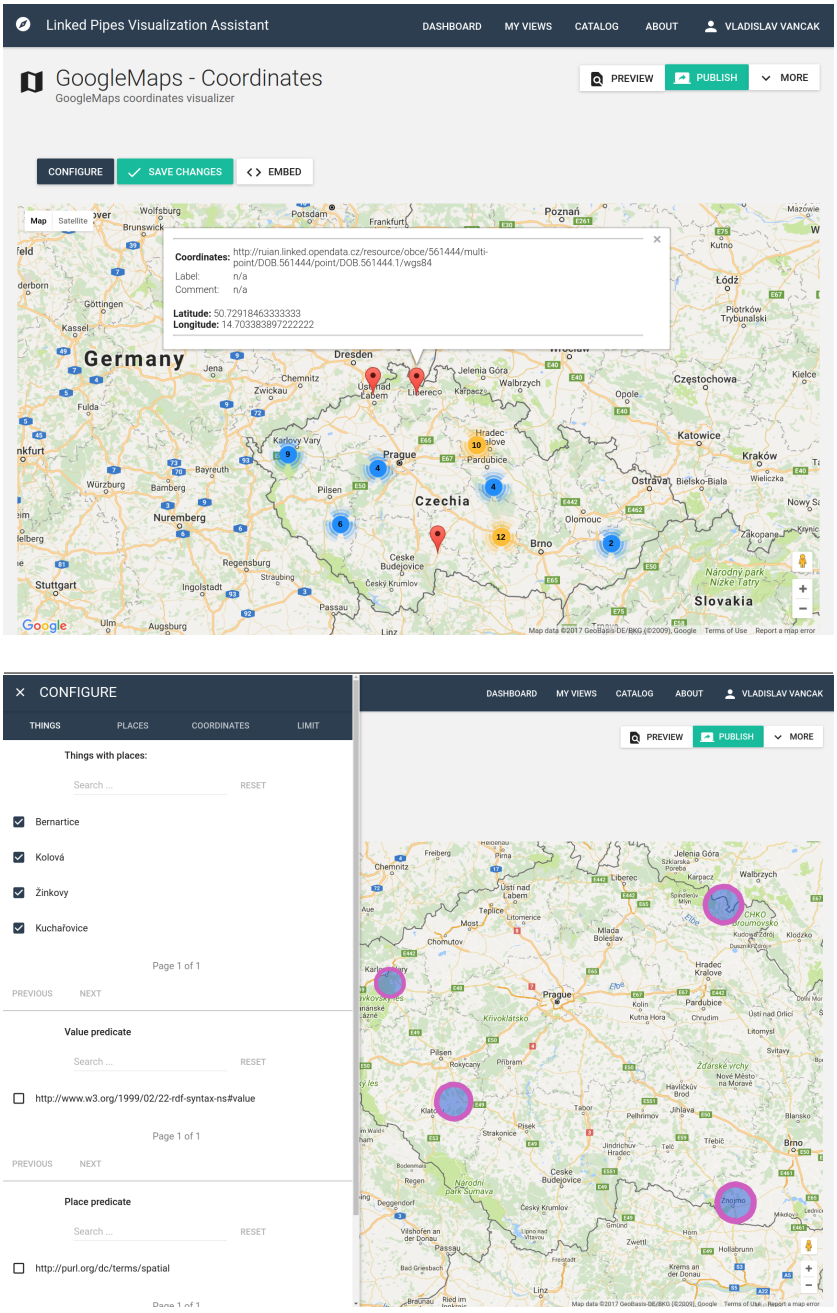


Figure 4.9: Google Maps visualizers. Top visualizer shows markers and example of an info bubble, showing information about the visualised coordinates; The bottom one visualises quantified things, using circles. Like timeline, configuration toolbar, buttons for saving the configuration and publishing the view are available.

5. Testing

With the visualizers implemented, we will test their performance in the following chapter. We will be simulating series of API calls (requests) coming from the front end in the user's browser, to the back end running at a common server (i.e. forms a possible bottleneck). We will monitor the response times and show the impact of various scenarios/configurations on the application performance.

Apache JMeter[1] was used to execute the tests and to create result tables and graphs. Each test was created using following steps:

1. Create, configure and publish a view
2. Observe API calls from client to the server
3. Re-Create the requests in the JMeter tests
4. Run the requests & record response times and throughput

The output graph results are shown in a common chart and can be distinguished by colours:

Data show individual response times (called *Data*)

Average response time

Median (midway value) of the response times

Standard **Deviation** (a measure of the variation)

Throughput – number of processed samples per unit of time

The individual figures at the bottom of the graph charts are values at the end of the test. *Latest Sample* is the last sample response time (shown on the graph as *Data*).

A result table, containing *maximum* and *minimum* response times, together with *error percentage* (i.e. percentage of requests that retuned other HTTP response code than 200-OK), is appended to every measurement as well.

At the beginning of each graph, we can see a sudden increase in response times, as the backend starts to process larger quantities of requests.

[1] http://jmeter.apache.org/

All tests were run with five simultaneous users, by creating 500 sample requests (i.e. 100 per user) with the data limit set to max 50 records loaded.

The test results were executed with the server running at a PC with Intel i7-4700HQ, 4 cores @2.4-3.4GHz CPU and 16 GB RAM available.

5.1 Time Line

First, we will test the TimeLine visualizers. We will start by comparing the highest level requests - Things with Things with Instants vs. Things with Things with Intervals. Secondly, we will show the performance of Instants and Intervals requests, where filtering by date is required and has a significant impact on the request response times.

5.1.1 Level 2 data loading, Instants vs Intervals

We start with the default, highest-level data loading in the TimeLine visualizers. From a single input data set, we created two views, using *Things with Things with Intervals* and *Things with Things with Instants* and left them unconfigured. We have observed the client requests, recreated them in the JMeter and ran them using the common testing scenario (described above).

Figure 5.1 shows the test results. As the charts show, the Intervals requests are a bit slower. That is the result of having slightly more resources to check when ensuring data presence in the lower levels (level 0 has 1x date-time for instants vs. start+end date-times for intervals).

5.1.2 Level 0 data loading, Instants vs Intervals

The second test scenario takes a look at the lowest level data loading. Like in the test above, we created two views of an input data set. We were using Instants and Intervals visualizers this time. We left them unconfigured, i.e. the default time range was used to load resources. In that state, we have copied the request for loading data and ran them using our test configurations.

Figure 5.2 shows the test results. The first observation is that both requests are significantly slower than the requests in the previous test. This is caused by the need of filtering values by date, required by configurations at this level (see implementation).

In comparison between the two level 0 requests, the Instants requests are a bit slower than the Intervals request, de having fewer data to fetch. This is caused by the execution of SPARQL endpoint calls, where the interval queries seem to run faster.

5.1.3 Restricting Level-Loaded data

With the level 0 loading being so slow, we may ask ourselves what is the impact on visualizers, that view more than just this level. As the implementation states, in that case, level 0 loads data only for a restricted set of resources (provided by the level above).

For this scenario, we used the visualisation created in the first test - the Things with Things with Instants. We re-created the request for Instants data (restricted to a set of resources loaded by level 1) and launched the tests.

Figure 5.3 shows a comparison of Level 0 data request from the previous test vs. request for Instants in the Things with Things with Instants visualisation. The average response time of the latter request is significantly slower – thanks to the loaded resources provided by the level above, the SPARQL query execution does not need to search through all available resources.

To answer the question from the beginning of this test scenario, although the standard level 0 responses are slow, they do not have a big influence on higher-level visualisations, thanks to the restrictions on loaded resources introduced by levels.

To conclude the observations from the tests, we can say that:

- When loading data for higher levels, where the levels just make sure that some data in the lower levels exist, loading data for instants is slightly faster.
- When loading data for level 0 (i.e. direct instants or intervals), we have to filter resources by the time range, as the level 0 configurations require. This makes the requests marginally slower. In this scenario, the intervals requests are the faster-responding ones.
- Loading data from level 0 is slow only if loading all values. When viewing data in upper levels, the level 0 data loader is restricted to load data for resources provided by level 1, which makes the requests notably faster.

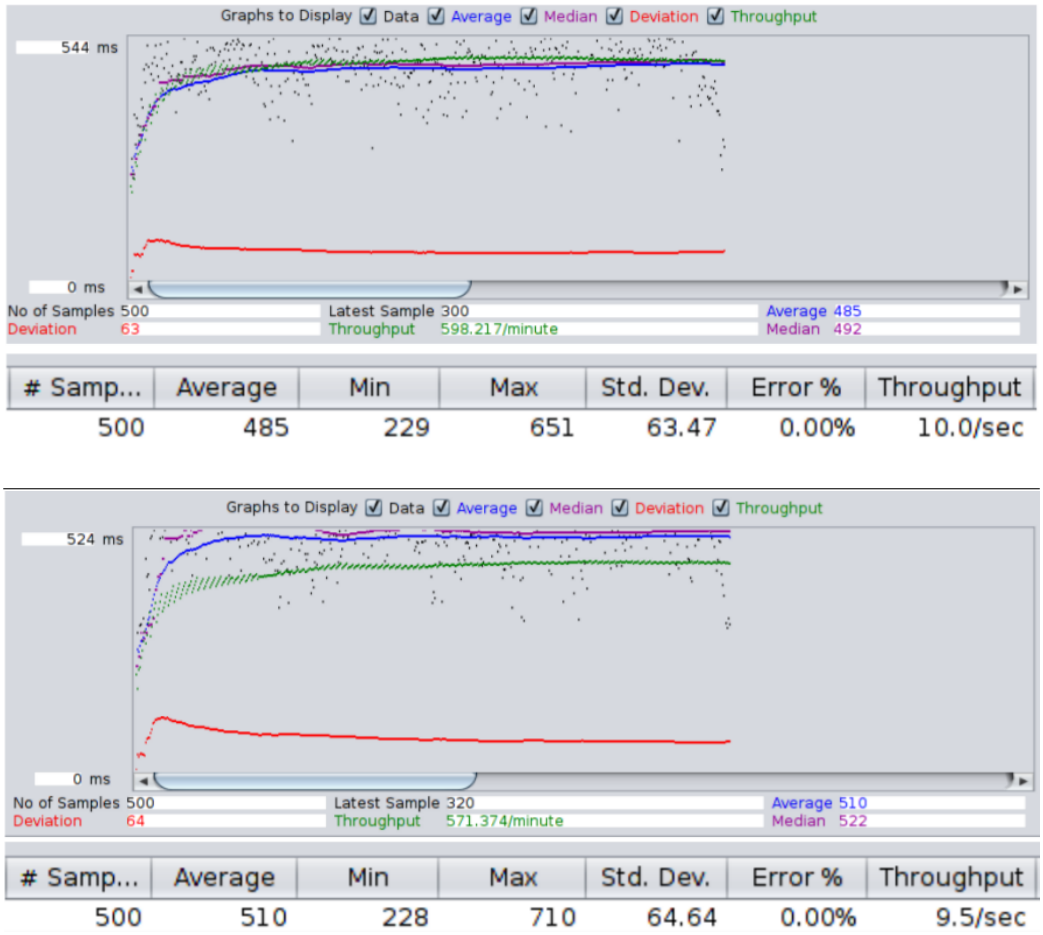


Figure 5.1: Response times of requests for Line Things with Things with Instants(top) vs Things with Things with Intervals(bottom)(i.e. Level 2 data) As we can see, the instants requests are slightly faster, mainly due to less data checks for the lower-level data.

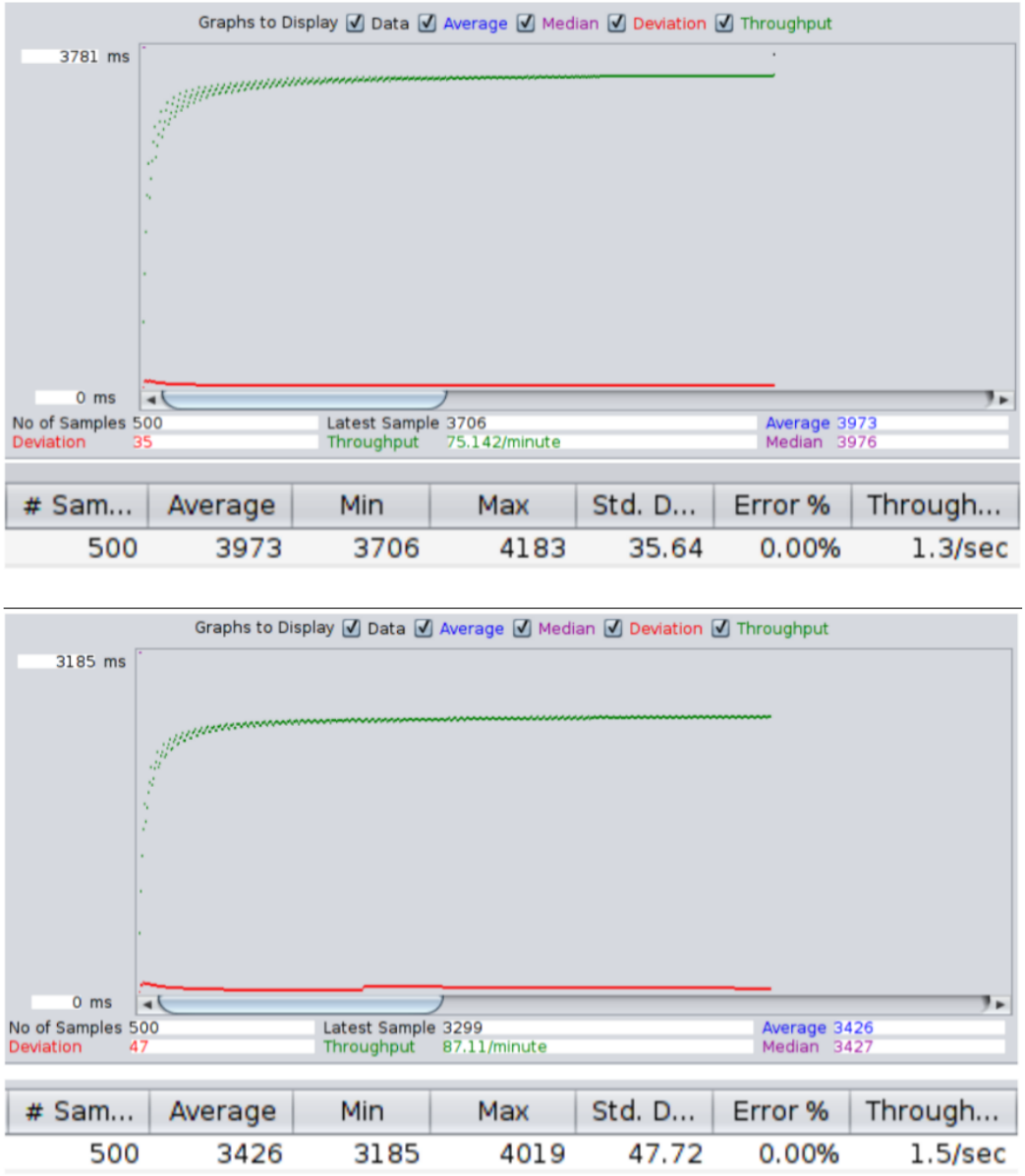


Figure 5.2: Response times of requests for Instants(top) vs. Intervals(bottom)(i.e. level 0 data). Note that in this level Intervals request is the faster one. The overall increase in response times, compared to the previous test is caused by a need to filter dates. Data, Average and Median graphs are not visible, as they are above the available range on the y-axis.

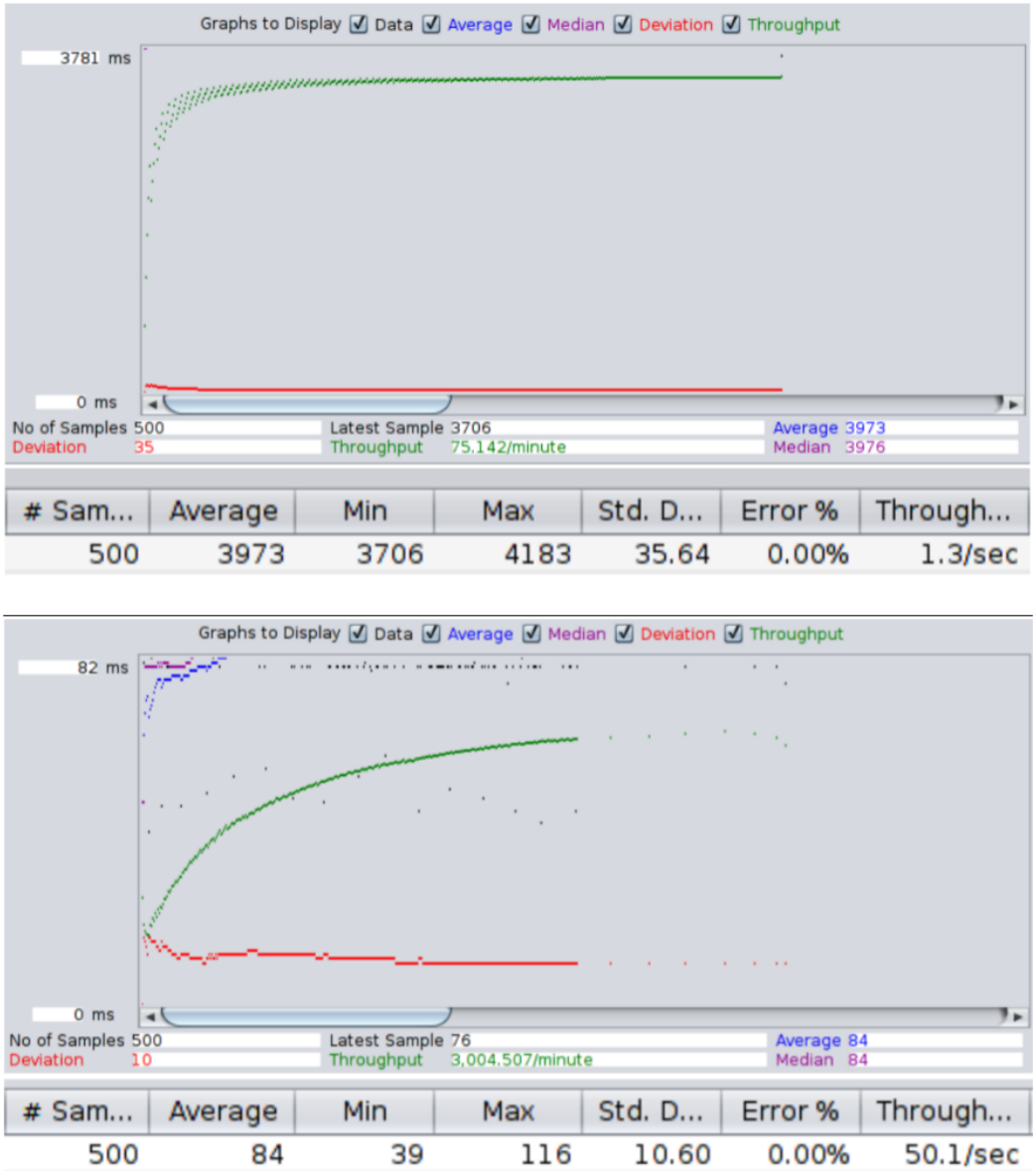


Figure 5.3: Response times of unrestricted(top) vs restricted(bottom) requests for Instants. The restricted request is significantly faster, thanks to the loaded resources being restricted by data from the upper level. Data, Average and Median graphs are visible only partially, as they are above the available range on the y-axis once again.

5.2 Google Maps

In the following section, we will test some of the GoogleMaps visualizers. We will start by comparing data loading in Places and Quantified Places, as it offers a unique opportunity to test the impact of adding one value to the requested data. As a second test, we will show the influence of visualizer configurations using the Quantified Things visualizer.

5.2.1 Places vs Quantified Places

The first test scenario shows the impact of adding a quantifier to the loaded data. We created Places and Quantified Places visualisations from the same data set, recreated the requests and ran the standard test scenario (5 users, 500 requests, limit 50 records loaded).

Figure 5.4 shows the test results. As we might have guessed, the Quantified Places responses are much slower, caused by a need to look for quantifiers in all linked resources.

5.2.2 Impact of a view configuration

After the last test, the immediate question is how to make the requests for resources with quantifiers faster. In this test, we created two Quantified Things visualisations and tried to specify where to look for the quantifier by configuring the visualisation. Specifically, we "restricted" properties used to find quantifiers - we selected all possible options. The visualised data stayed the same, let's now look at the performance.

Figure 5.5 shows the test results. As predicted, thanks to a more specific knowledge of the data location, response times from the configured visualisation represent less than half of the response times recorded by the unconfigured one.

To conclude the observations from the tests, we can say that:

- Loading data from an unconfigured visualisation is slow, as the underlying SPARQL endpoint has to look for data everywhere
- We can improve that by configuring the visualisation; specifying where to look for resources helps the application performance.

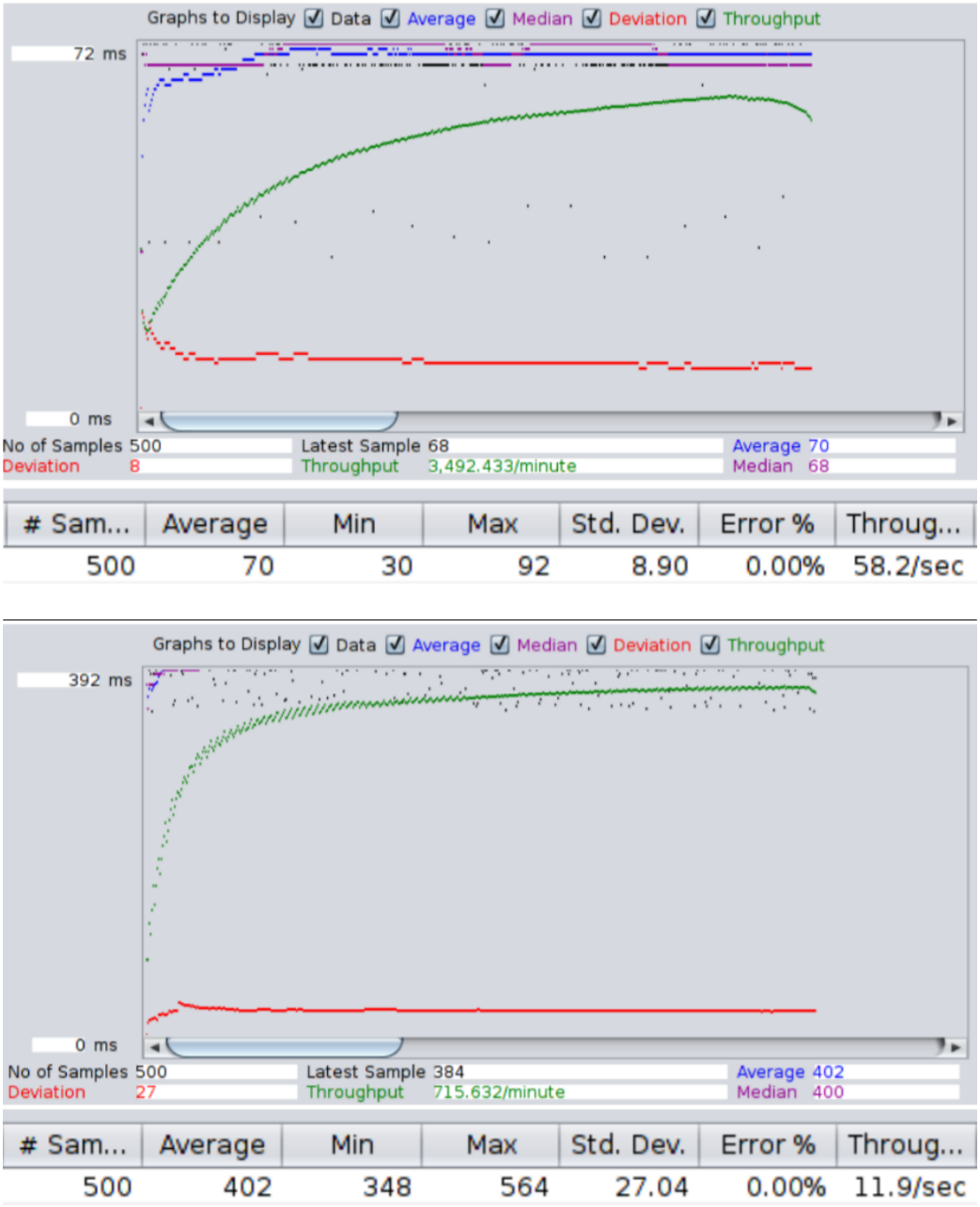


Figure 5.4: Google Maps test results: Places (top) vs Quantified Places (bottom) requests. Notice the difference in response times, caused by fetching additional data (quantifiers).

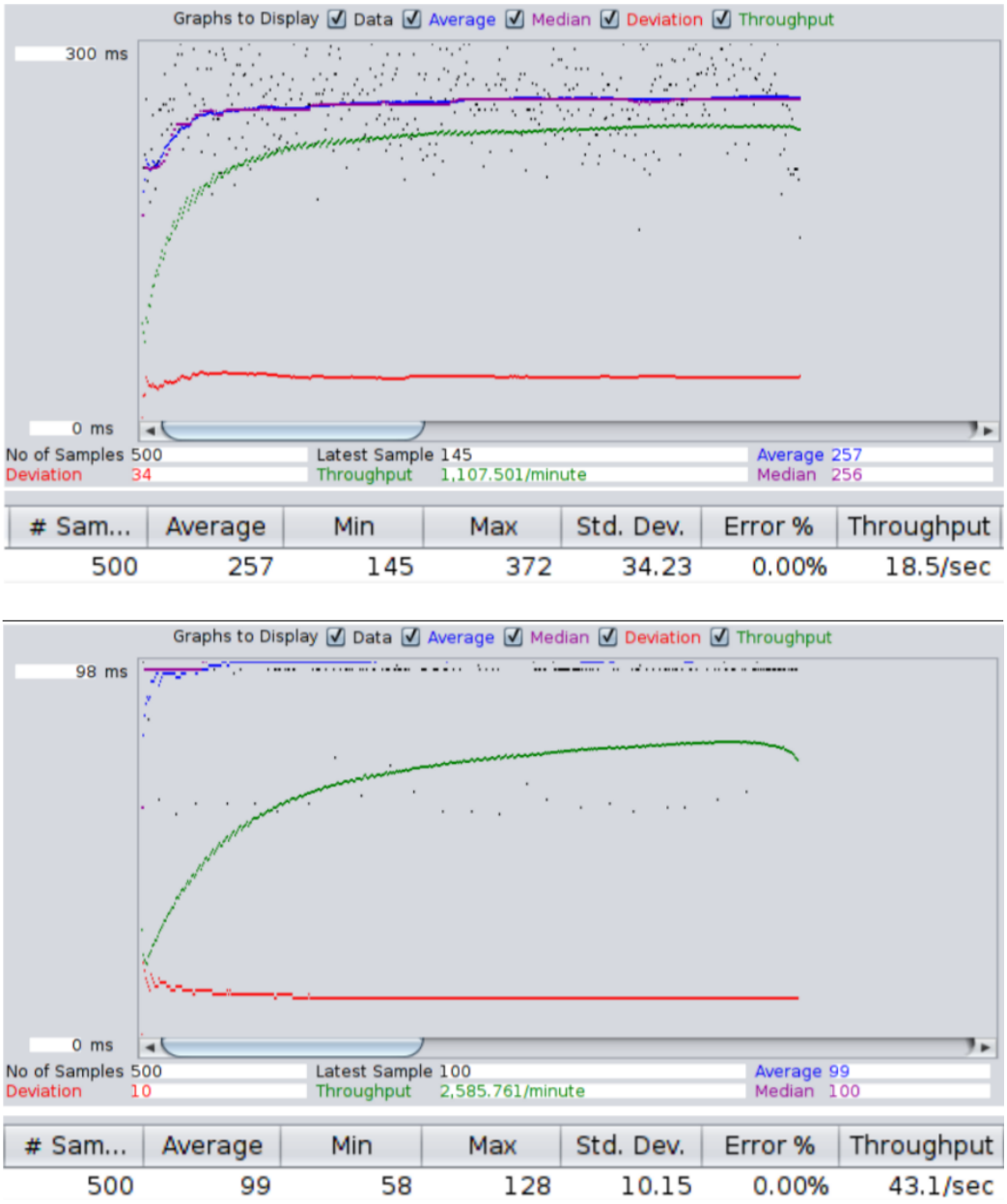


Figure 5.5: Influence of Configuration on Quantified Things requests. The top chart shows an unconfigured view, which allows searching for quantifiers everywhere. The bottom chart shows the effect of configuring the view by specifying properties used to link those resources (we selected all possible properties to keep the loaded data identical). Thanks to knowledge of the data location, request from the configured view is faster.

Conclusion

We implemented six TimeLine visualizers. *Instants*, *Things with Instants* and *Things with Things with Instants* visualise resources with a date and time on the TimeLine chart as circles. *Intervals*, *Things with Intervals* and *Things with Things with Intervals* visualise resources with a start-end combination of date-times on the TimeLine chart as rectangles. The visualizers support configurations according to the requirements from chapter 2.

We implemented four Google Maps visualizers. *Coordinates* and *Places* visualise resources with latitude and longitude on the map using markers. *Quantified Places* and *Quantified Things* visualise resources with latitude, longitude and a quantifier on the map using circles, with the circle radius corresponding to the quantifier. The visualizers support configurations according to the requirements from chapter 2.

Each visualizer allows the user to configure the visualisation, save the configuration and publish the created view.

The visualizers show *Labels* and *Comments* when visualising resources. Labels are also used in configurations, with configurable components using them as the main viewed value.

All TimeLine visualizers and GoogleMaps visualizers for quantified resources (i.e. those, that view circles) use *colours* to distinguish different types of visualised data.

The visualizers can handle large input data sets using *limits*. Users are allowed to restrict the amount of loaded data, with the visualisation providing information about the whole data set by showing *"Loaded (visualised data) out of (total available) records"*.

The visualizers are integrated into the *Linked Pipes Visualization Assistant*. All visualizers are implemented using the Assistant's conventions. Overall, we have *added* support for visualising time-based data formats and *extended* supported map-based data formats.

Bibliography

[1] Tim Berners-Lee. Linked Data: Design Issues, 2006. URL https://www.w3.org/DesignIssues/LinkedData.html.

[2] Tim Berners-Lee. The next web, 2009. URL https://www.ted.com/talks/tim_berners_lee_on_the_next_web.

[3] Josep Maria Brunetti, Sren Auer, Roberto García, Jakub Klímek, and Martin Nečaský. Formal Linked Data Visualization Model. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, page 309. ACM, 2013.

[4] Network Working Group. HyperTextTransferProtocol/1.1; RFC 2616, 1999. URL https://www.w3.org/Protocols/rfc2616/rfc2616.html.

[5] Ecma International. Ecmascript 2015 language specification, 2015. URL http://www.ecma-international.org/ecma-262/6.0/.

[6] Martin Nečaský Jirka Helmich, Jakub Klímek. Linked Data Visualization Model implementation, 2014. URL https://github.com/ldvm/LDVMi.

[7] Tobiáš Potoček. Linked Pipes Visualization Assistant, 2015. URL https://github.com/ldvm/LDVMi/tree/master/doc/assistant.

[8] W3C. SPARQL Query Language for RDF, 2008. URL https://www.w3.org/TR/rdf-sparql-query/.

[9] W3C. RDF - Semantic Web Standards, 2014. URL https://www.w3.org/RDF/.

[10] W3C. RDF 1.1 XML Syntax, 2014. URL https://www.w3.org/TR/rdf-syntax-grammar/.

[11] W3C. Terse RDF Triple Language, 2014. URL https://www.w3.org/TR/turtle/.

[12] W3C. Support for RDFa in HTML4 and HTML5, 2015. URL https://www.w3.org/TR/html-rdfa/.

A. Install guide

Requirements

1. *Java Development Kit (JDK)*[1], recommended version 8 or higher
2. *H2 Database*[2], recommended version 1.4 or higher
3. *Scala (sbt)*[3], recommended version 0.13 or higher
4. *Node.JS*[4], recommended version 4.2 or higher
5. *npm*[5], recommended version 5.0 or higher
6. *Virtuoso-Opensource*[6], recommended version 07.20 or higher

Launch the application

1. *H2 Database*

   ```
   (h2-root)/bin/h2.sh
   ```

2. *Virtuoso-Opensource*

   ```
   cd (virtuoso-root)/db && sudo virtuoso-t -f
   ```

3. *BackEnd - Scala*

   ```
   cd LDVMi/src && sbt run
   ```

4. *FrontEnd - JavaScript*

   ```
   cd LDVMi/src && npm run assistant-dev
   ```

5. *Launch the Assistant.* By default, it is available at *http://localhost:9000/assistant/*

[1] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[2] http://www.h2database.com/html/main.html
[3] http://www.scala-sbt.org/0.13/docs/Setup.html
[4] https://nodejs.org/en/
[5] https://www.npmjs.com/
[6] https://virtuoso.openlinksw.com/

First-Time run & setup

Before the first run, we will need *npm* to install all dependencies. To do that, run *npm install* in the project *src/* folder.

At the first run, a warning *Database 'default' needs evolution!* should appear. We will proceed by clicking on the button *Apply this Script Now*.

Now, we will need to create an account. Click on *Sign up* and complete the registration. Then, we should be able to sign in.

We are now on the home page. The next step is *Install and import LDVMi components and data sources* - available via a button located in the left section of the webpage.

The application is now ready. We can start creating views directly from the home screen (*Create View* button), or from the *Dashboard*, available via the top menu.

Select (or Add New) a Data Source ⇒ Select one of available visualizers ⇒ Configure the view ⇒ Publish the result. The whole process (with screenshots) is described in the LDVMi Assistant overview.

B. Testing datasets

Data structures required for our visualizers are not that common in well-known SPARQL endpoints like the DBpedia (http://dbpedia.org/sparql) or the Linked Data (https://linked.opendata.cz/sparql/), we will provide a list of example data sets for testing the visualizers.

Laws introduction dates

Endpoint: http://student.opendata.cz:8990/sparql
Graph: http://linked.opendata.cz/resource/application-pipeline-output/cz-legislation-acts-time-01
Visualizers: timeline Instants, timeline Things with Instants.

Laws with versions and intervals of validity

Endpoint: http://student.opendata.cz:8990/sparql
Graph: http://linked.opendata.cz/resource/application-pipeline-output/cz-legislation-acts-time-02
Visualizers: timeline Instants, timeline Intervals, timeline Things with Interval, timeline Things with Things with Intervals (as law-verson-validity interval).

Cities in the Czech Republic as geo-coordinates

Endpoint: http://student.opendata.cz:8990/sparql
Graph: http://linked.opendata.cz/resource/application-pipeline-output/cz-towns-ruian-geo-01
Visualizers: googlemaps - Coordinates, googlemaps - Places

Cities in the Czech Republic as coordinates, with quantifier representing the population count

Endpoint: http://student.opendata.cz:8990/sparql
Graph: http://linked.opendata.cz/resource/application-pipeline-output/cz-towns-ruian-geo-03
Visualizers: googlemaps - Coordinates, googlemaps - Places, googlemaps - Quantified Places

Cities in the Czech Republic with quantifiers representing the population count as things on the map.

Endpoint: http://student.opendata.cz:8990/sparql
Graph: http://linked.opendata.cz/resource/application-pipeline-output/cz-towns-ruian-geo-02
Visualizers: googlemaps - Coordinates, googlemaps - Places, googlemaps - Quantified Things with Places