Insert here your thesis' task.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Framework for mobile applications using Linked Data and the RÚIAN registry

Bc. Martin Melka

Department of Software Engineering
Supervisor: RNDr. Jakub Klímek, Ph.D.

April 28, 2018

Acknowledgements

I would like to thank my supervisor, Jakub Klímek, for the insight and guidance he offered me whilst I was working on this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on April 28, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Melka, Martin. *Framework for mobile applications using Linked Data and the RÚIAN registry.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018. Also available from: ⟨`https://github.com/andruian`⟩.

Abstrakt

Tato diplomová práce se zabývá návrhem a tvorbou frameworku, jenž umožní poskytovatelům dat propojit jejich data s existující sadou lokačních dat za využití principů propojených dat (Linked Data). Pro ověření a demonstraci funkcionality frameworku budou použita data z českého Registru územní identifikace, adres a nemovitostí (RÚIAN). Tato data jsou veřejně dostupná a dostatečně rozsáhlá pro zátěžové testování. Navrhovaný framework definuje flexibilní způsob propojování entit bez lokačních údajů s městy, domy a jinými objekty z RÚIAN nebo jakéhokoliv jiného vyhovujícího zdroje propojených dat. Práce nejprve definuje RDF slovník, kterým lze popsat metainformace o datech, jenž jsou propojována. Následně je vytvořena knihovna pro zpracování dat strukturovaných podle onoho slovníku do Java objektů. Dále je navržen a implementován indexovací server, který urychlí vyhledávání propojených objektů na základě jejich lokace. Nakonec je vytvořena klientská Android aplikace, využívající tento framework.

Klíčová slova Linked Data, RÚIAN, Framework, RDF, Android, mapa

Abstract

The aim of this thesis is to design and implement a framework, which will allow data publishers to link their data to an existing geospatial dataset using the Linked Data principles. The functionality of the framework will be validated and demonstrated using geospatial data provided by the Czech registry of territorial identification, addresses and real estate (RÚIAN). The data is publicly accessible and large enough to assess the performance of the framework. The proposed framework defines a flexible mechanism of linking entities with no spatial information to cities, houses and other objects located in RÚIAN or any other Linked Data source. Firstly, an RDF vocabulary used to provide metadata about the data to be linked is defined in the thesis. Then, a parsing library is created that converts data structured according to the vocabulary into Java objects. After that, an indexing server is designed and implemented to speed up spatial queries. Finally, an Android client application leveraging this framework is created.

Keywords Linked Data, RÚIAN, Framework, RDF, Android, map

Contents

Introduction **1**

1 State-of-the-art and available technology **3**
1.1 Linked Data and RDF 3
1.2 Existing applications for visualizing Linked Data 10
1.3 Existing libraries for RDF and Linked Data on Android 13
1.4 Techniques for spatial querying using SPARQL 13
1.5 Basic usability standards for Android OS and API for geolocation 16

2 Analysis **19**
2.1 RÚIAN registry 19
2.2 Framework functionality 23
2.3 Naive solution 24
2.4 Solution using an index server 25
2.5 Data definition vocabulary 32
2.6 Android app 40

3 Design **43**
3.1 Data definition parser library 43
3.2 Index server 44
3.3 Android app 57

4 Implementation **67**
4.1 Data definition parser library 67
4.2 Index server 68
4.3 Android app 72

5 Testing **79**
5.1 Automated testing 79
5.2 User scenarios 80

5.3 Stress test 88

6 Deployment **93**
6.1 Integrating linked data to the Andruian framework 93
6.2 Index server software stack 94
6.3 Android app 96

Conclusions and future work **97**
Future work 99

Bibliography **101**

A Acronyms **107**

B Resources **109**
B.1 Index SPARQL query template 109
B.2 Data definition vocabulary 111
B.3 Example data definition 116
B.4 Property path source for RÚIAN objects 118
B.5 SPARQL query for creating incremental testing data 121
B.6 Data definition created during a user scenario 122

C Diagrams **125**

D Contents of enclosed CD **129**

List of Figures

1.1 Linked Open Vocabularies graph [1] 8
1.2 DBpedia Mobile map view [2] 11
1.3 DBpedia Places screen [3] 11
1.4 LinkedPipes Visualization screen [4] 12

2.1 RÚIAN VFR data model [5] 20
2.2 Model of source data for Andruian framework 24
2.3 Architecture of solutions with and without an index server 27
2.4 Actors of the index server 29
2.5 Index server use case diagram 30
2.6 Index server domain model 31
2.7 Schema of including a remote RDF file 36
2.8 Data definition parser library domain model 39

3.1 Data definition parser library class model 44
3.2 Components of Jena Spatial 47
3.3 Indexing using Solr and MongoDB 48
3.4 Component diagram of Andruian index server 50
3.5 Class diagram of Andruian index server 51
3.6 The color scheme of the ViewLink app. 59
3.7 The home screen of ViewLink app 60
3.8 The home screen of ViewLink app with drawer menu open 60
3.9 The place details screen of ViewLink app 60
3.10 The data sources configuration screen 61
3.11 The data source color choosing dialog 61
3.12 The add data source screen 61
3.13 The Model-View-Presenter pattern [6] 62
3.14 ViewLink app Component model 62
3.15 ViewLink app package model 64
3.16 Subpackages of the `ui` package 65

4.1 Query visualization screen 69
4.2 Administration screen 70
4.3 ASyncTask sequence diagram 73
4.4 The main screen of the ViewLink app. 75
4.5 The drawer menu of the ViewLink app. 75
4.6 The place detail screen of the ViewLink app. 75
4.7 The data definition manager screen of the ViewLink app. 76
4.8 The color picker dialog of the ViewLink app. 76
4.9 The new data definition screen of the ViewLink app. 76

5.1 Schema of the testing dataset 85
5.2 The structure of testing data 87
5.3 Determining a SPARQL endpoint in Fuseki UI 87

C.1 Andruian Framework Data Definition schema 126
C.2 Class diagram of the DataDef class of the ViewLink app 127
C.3 UI flow of the ViewLink app 128

List of Tables

5.1 Indexing times with dual-core 2.8 GHz CPU and 12GB of RAM . 89
5.2 Android app data loading times without using index server 90

Introduction

The Internet is a vast collection of information and a large portion of it is unstructured. That makes extracting knowledge from it difficult. One approach to make this task easier is the concept of Linked Data. The "regular" web which is widely known and used is a web of documents. The documents are linked together through hyperlinks which have no metadata associated with them. Their meaning must be therefore inferred by the reader. The web of data, facilitated by the Linked Data paradigm, consists of ontological entities and clearly defined relations among them. One source of such data is RÚIAN, the Czech Registry of Territorial Identification, Addresses and Real Estate. The data describes all spatial entities in the Czech Republic and provides meaningful links among them.

CHANGED: The aim of this thesis is to create a framework which allows users to visualize entities linked to RÚIAN. The entities may be offices, points of sale, businesses and others which have been previously linked to the RÚIAN registry by data publishers. The framework also allows for effective geospatial querying over data linked this way.

The thesis first describes the RÚIAN registry and its data model. Then it defines an RDF vocabulary used to provide metadata about the data linked to the registry. Next, a parser library for converting the metadata into Java objects is created. Then, an index server is discussed and implemented to speed up spatial queries of the data. Lastly, an Android application prototype is designed and implemented to demonstrate how the framework may be used to provide information to users about entities around them.

Chapter 1

State-of-the-art and available technology

The first section of this chapter gives an introduction into the technology of Linked Data, the RDF model, its serialization format Turtle, the querying language SPARQL and the concept of RDF vocabularies. The second section provides an overview of already existing solutions from the domain of this thesis. That is, applications visualizing Linked Data with a geospatial component. Thirdly, the chapter introduces some available libraries for RDF and Linked Data manipulation for the Java platform and Android OS. Fourthly, techniques for spatial querying using SPARQL are discussed. The fifth section highlights some basic usability standards of the Android platform and explores the available geolocation API.

1.1 Linked Data and RDF

This section is an introduction to the technology of Linked Data. It examines the basic principles of Linked Data, the reasons for it and the language used by it.

1.1.1 Linked Data

Linked Data is a set of best practices, a paradigm for publishing data in a flexible and inter-operable way. It was coined by Sir Tim Berners-Lee and provides a solution to the *silo problem*, where organizations and departments within those organizations all store data in their own separate way, creating isolated data islands [7]. Even when data is open and intended for public use, its utility may be degraded by the format it is published in.

Linked Data uses the well-established *Hypertext Transfer Protocol (HTTP)* and *Uniform Resource Identifiers (URIs)* and so does not try to reinvent the wheel. It reuses already existing technologies to create a web of data, readable

by machines and algorithms, as opposed to the web of documents, which is mostly readable by humans. There are four basic principles of Linked Data [8]:

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.

Following those principles results in a unifying data model, where data can be shared globally across different schemata. Because entities are identified by HTTP URIs, data from different data sources may be linked and automatically discovered. However, unlike the web of documents, Linked Data uses the concept of vocabularies. Vocabularies are sets of metadata which define the ontological meaning of entities and relationship links among them.

1.1.2 RDF

Resource Description Framework (RDF) is a model designed for data exchange on the web [9]. Its core principle is evolvability of data and its schema. In contrast to common data sharing formats, such as JSON and XML, and standard data storage technologies, such as relation database systems, the schema of data published using RDF is flexible and may change without necessitating data consumers to act upon this change.

RDF model is formed by a set of triples. Each triple represents a relationship between two entities. The simplest RDF serialization format, N-Triples, consists of entries such as:

```
<subject> <predicate> <object>
```

This triple specifies that a `subject` is in a relationship of type `predicate` with an `object.` Each of the first two terms are URIs. The third term, the *object*, may either be an URI or a literal. Literals may be typed and they might even have a language tag.

According to Linked Data principles, every modeled entity has its own unique identifier, and that includes predicates. Therefore the meaning of a link between two entities can be found by dereferencing its URI and obtaining more information about it.

RDF model can be thought of as a directed multigraph, with subjects and objects as its nodes and predicates as its named edges.

1.1.2.1 Turtle

There are several serializations which can be used to represent RDF. These include `RDF/XML`, `N-Triples`, `Turtle`, `JSON-LD` and others[10].

All examples of RDF in this thesis will be using Turtle format because of its density and good human readability. This section covers the basic syntax, for more details refer to the RDF 1.1 Turtle W3C Recommendation document [11].

Each subject and predicate must be an IRI. IRI is a superset of URI with the difference that Unicode characters may be used instead of plain ASCII. That is only a subtle difference so I will be using URI and IRI interchangeably throughout the thesis. An IRIs can either be fully qualified and enclosed in angle brackets, e.g.

```
<http://example.org/resource>
```

or it can be written in a compact form, CURIE [12], without angle brackets. In that case the IRI will consist of a prefix and a reference. Prefixes in Turtle are defined using the keyword `@prefix`. The two triples are identical:

```
@prefix ex: <http://example.org/> .

ex:subject ex:predicate ex:object .

<http://example.org/subject>
    <http://example.org/predicate>
    <http://example.org/object> .
```

To be concise, in examples below where the prefix is irrelevant, an empty prefix (a colon followed by a reference) will be used.

In Turtle a triple is represented as

```
:subject :predicate :object .
```

Literals are values in place of the *object* that are not resources and do not have IRIs. They may be integers, floats, strings and other types, as shown below:

```
:s :o "string" .  # A plain string literal
:s :o 42 .        # An integer literal
:s :o 3.14 .      # A floating-point literal
:s :o "2018-03-28T19:20:45"^^xsd:dateTime . # A datetime-typed literal
:s :o "řetězec"@cs . # A language-tagged literal
```

Often a subject will be present in several triples with only the predicate and object changing. In such cases, a shortened form may be used, where different predicates and objects are separated by a semicolon (`;`). This is a representation of two triples with the same subject:

```
:subject
  :predicate        :object;
  :anotherPredicate :anotherObject .
```

Sometimes both subject and predicate should be the same and only object be different. In that case, a colon (`,`) is used to separate objects. This is a representation of two triples with the same subject and predicate:

```
:subject :predicate :object,
                    :anotherObject .
```

It may be helpful to model a relationship, where the URI of an object node (in the graph representation) is not needed. In that case a *blank node* may be used in place of an object. For example, a subject having a relation to a date, which consists of a day, month and a year, may be modeled this way:

```
:subject :date [ :month 3;
                 :day   14;
                 :year  1592
               ] .
```

Furthermore, the `rdf:type` predicate is used so often that it warrants a shortcut - the keyword `a`:

```
:subject a :object .
```

Sometimes we might want to traverse multiple properties at once and get the object at the end of the property chain. Turtle offers a shortcut which is called a *property path* and does exactly that. Two or more properties may be separated using a slash (`/`). In the following example, the `:subject` contains a property of type `:a` to an anonymous object. That object contains a property of type `:b` to another anonymous object, which contains a property of type `:c`, that leads to the final `:object`.

```
:subject :a/:b/:c :object .
```

1.1.3 RDF vocabularies

Plain RDF does not concern itself with ontological meaning of entities and relationships among them. It does not define any domain-specific concepts and instead makes its users provide the meaning through RDF vocabularies. A vocabulary is a mechanism for ontologically describing RDF data. It allows for creation of new RDF classes and properties, giving them human-readable names and descriptions and impose certain constraints on them.

A basic and widely used language for describing vocabularies is RDF Schema [13]. It defines several elementary classes and property types upon

which ontologies may be built. They belong to the `rdf` or `rdfs` namespace[1]. Below is an incomplete list of the most important entities [13].

rdfs:Resource is *anything* described by RDF. Everything is a resource.

rdfs:Literal is a node containing a literal value. It may be typed to represent a number, a string, a date and more.

rdfs:Class represents the concept of a *type* of a resource. For example, a resource identifying a book could belong to the category (have a type) `dcterms:BibliographicResource`, which is a class defined by the Dublin Core Metadata Initiative Terms vocabulary located under the `dcterms` prefix.

rdf:Property represents resources that are RDF properties.

rdf:type property indicates that a resource is a member of a class, or, in other words, that it is an instance of a class.

rdf:range is a property that may be defined for `rdf:Property` and specifies instances of which classes may be objects of the property.

rdf:domain is similar to `rdf:range`. It is a property that may be defined for `rdf:Property` and specifies instances of which classes may be subjects of the property.

rdfs:label is a property used to provide resources with a human-readable name.

rdfs:comment is a property used to provide resources with a human-readable description.

This is an example definition of a `rdfs:Class` describing a book, an article, or any other documentary resource:

```
dcterms:BibliographicResource
    dcterms:hasVersion <http://dublincore.org/usage/terms
                        /history/#BibliographicResource-001> ;
    dcterms:issued "2008-01-14"^^<http://www.w3.org/2001
                                          /XMLSchema#date> ;
    a rdfs:Class ;
    rdfs:comment
        "A book, article, or other documentary resource."@en ;
    rdfs:isDefinedBy <http://purl.org/dc/terms/> ;
    rdfs:label "Bibliographic Resource"@en .
```

[1] A useful tool for looking up the fully qualified IRIs based on prefixes is available at `https://prefix.cc`

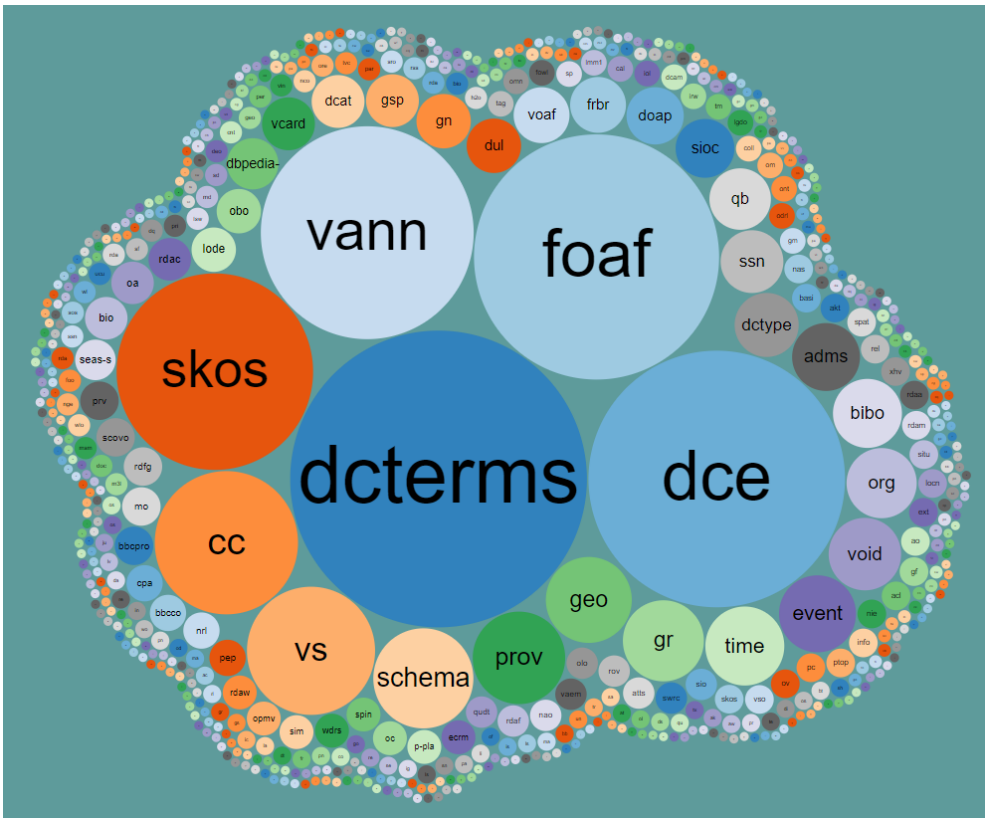

Figure 1.1: Linked Open Vocabularies graph [1]

Hundreds of vocabularies have already been created. It is a cumulative effort and vocabularies often reuse one another. The Linked Open Vocabularies (LOV) is a service which gathers information about the most popular vocabularies and attempts to help users find a vocabulary that fits their needs. The Figure 1.1 shows the vocabularies contained in LOV. The size of each circle reflects the amount of other vocabularies using the given vocabulary.

The vocabularies used further in this thesis include:

dcterms[2] is a vocabulary maintained by the Dublin Core Metadata Initiative. It defines classes and properties mainly related to books and publications and their properties.

schema[3] is a vocabulary maintained by Google, Microsoft and others and aims to provide a way for webmasters to markup their data. Correctly formatted data may then be easily understood by search engines and offer users more relevant information. The vocabulary covers the concepts of creative works, e-shops, events, organizations and more.

ruian[4] is a vocabulary used in the Linked Data representation of RÚIAN. It defines territorial concepts of regions, counties, cities, address places, relationships among them and more.

1.1.4 SPARQL

This section describes the basic of the SPARQL query language. It only covers features used later in the thesis.

SPARQL[14] is a RDF query language for the retrieval and manipulation of RDF data. It can be loosely thought of as a counterpart of SQL for data stored in RDF format. It defines several query types, including inserting and deleting data, but we will only focus on the type used in this thesis - the `SELECT` query. Much like SQL, the `SELECT` query returns data based on a given set of conditions.

Each query of this type consists of a `SELECT` clause, where variables to be returned are listed, and a `WHERE` clause, where search conditions are specified. Below is an example of a SPARQL query. This query will match all triples in the database that have the given `title` as their predicate. For each matched query the subject IRI and the object will be returned. As shown in the example, variables in SPARQL are identified by a question mark (?).

```
SELECT ?subject ?title
WHERE
{
  ?subject <http://purl.org/dc/elements/1.1/title> ?title .
}
```

In order for SPARQL to return a result, all parts of the query must match. If we want to match some parts of the query optionally, and return a null value if they are not provided, we can use the `OPTIONAL` clause. The following query will match patterns where a resource contains the property `:a` *and* the property `:b`. In cases where the object of property `:b` contains the property `:c`, its object is returned as variable `?optVal`. Otherwise, nothing is returned for this variable (but the pattern still matches, because it is marked as optional).

```
SELECT ?subject ?val ?optVal
WHERE
{
  ?subject :a ?val;
           :b ?anotherObject.

  OPTIONAL{
      ?anotherObject :c ?optVal.
  }
}
```

The result patterns may be filtered based on a custom set of conditions, using the `FILTER` function. It accepts a boolean expression as a parameter, which must evaluate to True else the result pattern is discarded from the result. The following query will return IRIs of all resources that have the property `:a` defined and its value compares as greater than 10.

```
SELECT ?subject
WHERE
{
  ?subject :a ?val.
  FILTER(?val > 10)
}
```

It may be the case that a given RDF store does not contain all the data we need and a part of it is located elsewhere. As long as the remote location also exposes the data via SPARQL, it can be fetched using a *federated query*. The following query demonstrates this feature. The "main" SPARQL endpoint contains triples with properties of type `:a`. The objects of those triples further contain properties of type `:b` whose objects we want to retrieve, but they are accessible at a remote SPARQL endpoint, `http://example.org/sparql`:

```
SELECT ?subject ?val
WHERE
{
  ?subject :a ?anotherObject.

  SERVICE <http://example.org/sparql> {
      ?anotherObject :b ?val.
  }
}
```

1.2 Existing applications for visualizing Linked Data

This section highlights a few applications that offer Linked Data visualization functionality. However, none of the applications solves the problem that is addressed by this thesis.

1.2.1 DBpedia Mobile

DBpedia is a project started by the Free University of Berlin, Leipzig University and the University of Mannheim in collaboration with OpenLink Software[15]. It is the pioneer project leveraging the linked data paradigm and is one of the

Figure 1.2: DBpedia Mobile map view [2]

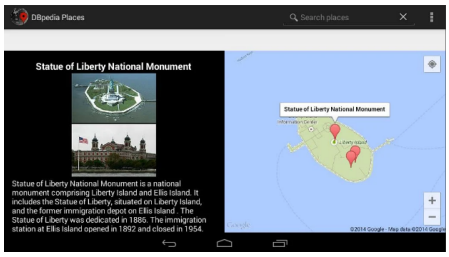

Figure 1.3: DBpedia Places screen [3]

main sources of data in the *linked open data cloud*[16]. Its aim is the extraction of structured content from Wikipedia and related Wikimedia projects. The content includes over 4 million consistently classified entities in the English version. Various amount of data is also available in over 100 other languages[17]. All data is published in accordance to the linked data principles, discussed in subsection 1.1.1.

DBpedia Mobile is a DBpedia client application for mobile devices. It allows users to view and browse data sourced from DBpedia based on their real world location. The data is presented on a map and upon inspecting a data element, detailed information about it is displayed based on the well-known RDF links defined for it. This data may include related entities, reviews, images and more. The application utilizes the location of a mobile device provided by an embedded receiver of GPS or other geospatial system to show places in proximity to the user[2]. An example screen of the application is shown in Figure 1.2.

The application is rather outdated and aimed at devices using Windows Mobile 6. It is not clear what its performance is, nor how efficiently it can query geospatial data. Its data source is also only limited to DBpedia, no custom sources may be added.

1.2.2 DBpedia Places

DBpedia Places is another mobile location-aware browser of DBpedia data. It is aimed at Android devices and allows users to display data sourced from DBpedia on a map and view their details. It also supports searching through entities. A picture of the screen is shown in Figure 1.3.

It was last updated in 2015 and like DBpedia Mobile, it does not support serving data from a source other than DBpedia[3].

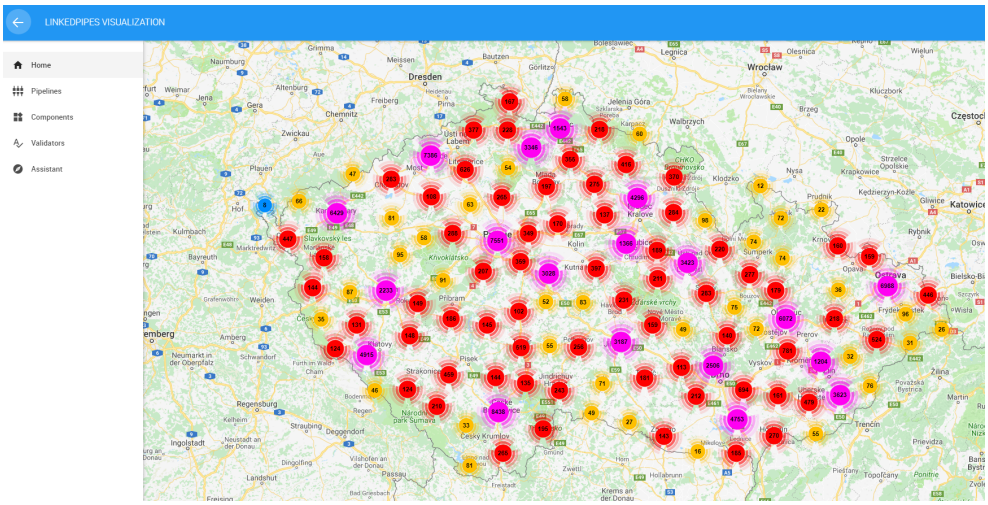

Figure 1.4: LinkedPipes Visualization screen [4]

1.2.3 LinkedPipes Visualization

LinkedPipes Visualization[4] is a project of the LinkedPipes team that aims to provide a graphic representation of Linked Data. The tool does not focus solely on map-based visualization. It understands geospatial data described through the `schema.org` vocabulary, and also other data types described by appropriate well-established RDF vocabularies. Only the map-based visualization is relevant for this thesis, so the other operation modes will not be discussed further.

The visualization tool can consume data from a local or remote RDF file or through a SPARQL endpoint. It does not require any further configuration and is able to visualize data based solely on its structure and vocabularies used to annotate it. Given geospatially-annotated data, the tool shows a map with markers placed at the locations of the data points. An example of the visualization screen is shown in Figure 1.4.

The tool supports arbitrary data sources, which makes it much more flexible than the mobile applications discussed previously. However, it presupposes a particular structure of the data. It requires that every data point has its location coordinates attached to it. A data point may not be linked to another data point which would supply the location information. Furthermore, no optimization in terms of geospatial indexing is implemented. The dataset is processed in its entirety and it is not possible to query data only around a certain region.

1.3 Existing libraries for RDF and Linked Data on Android

This section provides an overview of existing libraries that may be used to work with RDF and Linked Data. It describes the libraries without going into much detail and does not attempt to name the most suitable one. That question is expanded upon and answered later in the thesis, in section subsection 3.1.1.

1.3.1 Semargl

Semargl is a modular Java framework for crawling linked data. It is designed to be lightweight and requires no external dependencies. That makes it a good candidate for use in mobile applications which are sensitive to library sizes. However, it only supports a small subset of RDF serialization formats.

The library is publicly available from GitHub[18].

1.3.2 Jena

Apache Jena is a Java framework for RDF manipulation and building application leveraging Linked Data[19]. Along with its RDF API it also provides SPARQL support, a triple store for RDF data and ontology APIs. It is not primarily intended to be used in mobile applications and is quite large.

Jena does not work on Android out-of-the-box due to missing packages and namespace conflicts. However, ports of the original library for Android exist, such as `androjena`[20] and `jena-android`[21], which attempt to work around this issue. Both ports seem not to be maintained anymore, however.

1.3.3 RDF4J

RDF4J is a Java framework for RDF manipulation which was initially developed by the Aduna company under the name OpenRDF Sesame. After the company dropped its support in 2016, it was forked by Eclipse and named RDF4J [22]. The feature set of the framework is similar to what Jena offers. Namely, it provides an RDF model, API for its manipulation, a RDF parser and serializer called *Rio (RDF I/O)*, a triplestore and a HTTP SPARQL server.

Like Jena, RDF4J does not work on Android out-of-the-box due to missing packages for XML manipulation.

1.4 Techniques for spatial querying using SPARQL

This section provides an overview of possible approaches for querying over spatial data using SPARQL. The approaches include a “naive” solution as well as more efficient techniques supported by various RDF database engines.

1.4.1 Naive spatial SPARQL querying

This approach for spatial querying works with any database supporting SPARQL. It presumes that the spatial element of the data is represented by a latitude and longitude value. It is constituted by a `FILTER` function inside a SPARQL query which filters out all resources whose position does not fall into a certain rectangular area. The filtering is done by a expression consisting of a set of comparisons. In the following example `?lat` and `?lng` are variables containing latitude and longitude coordinates of a resource, respectively. The search area is defined by a bounding rectangle with coordinates `minLat`, `minLng`, `maxLat` and `maxLng`.

```
FILTER(
    ?lat > ?minLat && ?lat < ?maxLat
    ?lng > ?minLng && ?lng < ?maxLng
)
```

This formula needs to be adjusted for the corner case of the latitude or longitude of the area "overflowing" from positive values to negative ones. That is due to the fact that the positional system WGS84 which is used in this thesis allows latitude to have values ranging from -90 to +90 and longitude to have values ranging from -180 to +180. For example, a rectangle's leftmost longitude may be 170 and its rightmost -175. This may be solved by detecting the case of an overflow and adding an "offset" of 180 or 360 to the latitude or longitude, respectively.

The simplicity of this approach results in poor performance. The database engine still has to effectively walk through all triples in the dataset and filter each of them individually.

1.4.2 Jena Spatial

Jena Spatial[23] is an extension to Apache Jena ARQ, which is the Jena SPARQL processor. The extension requires an external spatial index for efficient spatial queries. The index may be either Apache Lucene[24] or Apache Solr[25].

In order to introduce the spatial element into SPARQL queries, Jena Spatial defines an RDF vocabulary with properties specific for spatial searching. That includes the `spatial:nearby` property, which returns all resources around a certain position. This is an example query taken from the Jena Spatial documentation [23] which makes a query for all places within 10 kilometers of latitude/longitude 51.46, 2.6.

```
PREFIX spatial: <http://jena.apache.org/spatial#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

```
SELECT ?placeName
{
    ?place spatial:nearby (51.46 2.6 10 'km') .
    ?place rdfs:label ?placeName
}
```

1.4.3 GeoSPARQL

GeoSPARQL[26] is a geographic query language for RDF data. It defines a vocabulary for representing geospatial data in RDF and it also defines an extension to SPARQL for processing such data.

Like Jena Spatial, GeoSPARQL defines functions for SPARQL for posing spatial queries. For example, the following query finds all resources of type `ex:Park` that are less than 3000 meters away from the resource `ex:WashingtonMonument`.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?p
WHERE {
    ?p a ex:Park ;
       geo:hasGeometry ?pgeo .

    ?pgeo geo:asWKT ?pwkt .
    ex:WashingtonMonument geo:hasGeometry ?wgeo .
    ?wgeo geo:asWKT ?wwkt .

    FILTER(
        geof:distance(?pwkt, ?wwkt, units:m) < 3000
    )
}
```

Unlike Jena Spatial, which was specific to Apache Jena, GeoSPARQL is a standard and is not bound to a particular SPARQL processor and triple store. One of the databases that support GeoSPARQL is GraphDB. It supports spatial querying both with and without using an index [27].

1.4.4 Virtuoso Geo Spatial Enhancements

OpenLink Virtuoso also offers spatial query functionality [28]. The support includes SPARQL functions for working with geometric shapes, determining their relative location, intersections, transformations and more.

The documentation states that Virtuoso is also increasingly compliant with the aforementioned GeoSPARQL standard.

1.5 Basic usability standards for Android OS and API for geolocation

This section is concerned with two topics - the usability standards and the geolocation API for the Android platform. The usability standards are thoroughly covered by the Material Design visual language, which is further discussed below. The geolocation API offered by the Android platform is shortly described afterwards.

1.5.1 Material Design

In 2014, With Android 5.0 Lollipop, Google introduced Material Design[29]. It is a set of principles that aim to help create better user interfaces (UI) not only for mobile devices that are easy to grasp by the users. It has become widely used and is the de-facto standard for most apps targeting the Android platform.

The basic principle of the Material Design visual language is the metaphor of a material. Each element of the user interface is a piece of material, which is grounded in tactile reality [29]. The material has surfaces and edges, user interface elements are in relation to one another, they overlap and cast shadows. The concept of light, surface and motion conveys how objects exist in space and relate to each other.

Another principle of the visual language is the use of grids, white space and contrasting colors. That creates hierarchy among the visual elements and helps guide the user's focus. Each application is defined by its primary color used, among others, for toolbar headings and an accent color, which is usually used to indicate the main action the user may perform.

These are the very basic concepts of the Material Design. It is a comprehensive set of guidelines and its analysis is out of the scope of this thesis.

1.5.2 Geolocation API

The Android platform provides a straightforward API for determining the device location. Each application may access the system `LocationManager` and query it for device location updates [30]. The location may be requested at two precision levels. The first is a *coarse location*, which is precise to the size of a city block and is determined through GSM towers, WiFi networks in range and others. The second one is a *fine location*, which is as precise as the GPS chip allows. The difference among the two modes of location is in accuracy, speed of obtaining the location and battery efficiency.

Both levels of location precision require a user permission declared in the AndroidManifest.xml file in order for the Android OS to deliver the location data to the application. The requirement names are:

android.permission.ACCESS_FINE_LOCATION for fine location,

android.permission.ACCESS_COARSE_LOCATION for coarse location.

Android versions prior to 6.0 Marshmallow will prompt the user to agree to the permissions required by the app upon installation, whereas with later versions the developer must prompt the user when the resource protected by a permission is first requested.

Subscribing an application to location updates is illustrated in the following code snippet [31]:

```
// Acquire a reference to the system Location Manager
LocationManager locationManager = (LocationManager)
          this.getSystemService(Context.LOCATION_SERVICE);

// Define a listener that responds to location updates
LocationListener locationListener = new LocationListener() {
    public void onLocationChanged(Location location) {
      // Called when a new location is found by the
      // network location provider.
      makeUseOfNewLocation(location);
    }

    public void onStatusChanged(String provider,
                                int status,
                                Bundle extras) {}
    public void onProviderEnabled(String provider) {}
    public void onProviderDisabled(String provider) {}
  };

// Register the listener with the Location Manager
// to receive location updates
locationManager.requestLocationUpdates(
     LocationManager.NETWORK_PROVIDER, 0, 0, locationListener
);
```

Chapter 2

Analysis

This chapter is concerned with the analysis of topics relevant to the thesis. Firstly, it describes the RÚIAN registry in regards to the proposed framework. The RÚIAN data model and the RDF vocabulary for that data are also covered. Secondly, the functionality of the framework is outlined. Thirdly, a naive approach to the problem of the framework implementation is discussed. Fourthly, a better, scalable solution to the problem, which uses an intermediate index server, is presented. Lastly, the requirements on the proposed Android applications are defined.

2.1 RÚIAN registry

RÚIAN stands for the Registry of Territorial Identification, Addresses and Real Estate[32]. It is one of four primary registries which were launched by the Czech government in 2012. RÚIAN contains descriptive and location data about territorial elements, addresses and other units relevant to the Czech State Administration of Land Surveying and Cadastre. It is the source of all addresses in the country - any new address place must be registered and created here [33]. The registry exposes stored data in XML-based format VFR (Výměnný Formát RÚIAN) [34].

XML is a format that is easily readable by computers, but is not ideal for linked data manipulation. The OpenData.cz initiative, consisting of students and staff of Prague universities, works to provide the public with government-published data in an open data fashion. One of their projects is the conversion and publication of RÚIAN data from the VFR format into the RDF, linked-data format. The resulting dataset is published on the OpenData website[5] and can be queried via a SPARQL endpoint[6].

[5] https://linked.opendata.cz/dataset/cz-ruian
[6] https://ruian.linked.opendata.cz/sparql

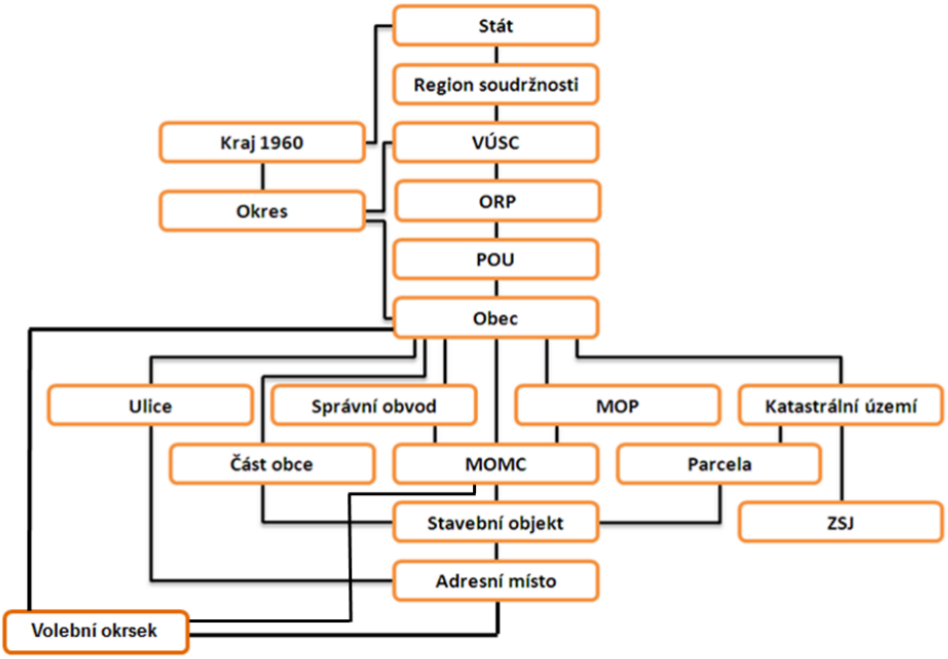

Figure 2.1: RÚIAN VFR data model [5]

The VFR format defines the underlying data model shown in Figure 2.1. The model starts with the broadest entity on top - the country - and goes on to describe every region, county, town, street, cadastre unit, parcel and more. The converted RDF schema reuses the same entities used in VFR. However, according to Linked Data principles, the RDF schema has to define unique URIs for all objects, including the links between them. This is not covered in the VFR format specification and had to be done by the OpenData initiative.

2.1.1 RÚIAN linked data model

This section analyses the RÚIAN linked data model, as provided by the OpenData organization. It is based on the VFR data format, but enriched with unique names, links and properties. The analysis is concerned with the HTTPS version of the RÚIAN SPARQL endpoint[7]. A HTTP version also exists, but uses a slightly different naming schema and is considered deprecated by the maintainer.

The analysis and diagram do not cover all properties, but only focus on items relevant to the framework. That primarily means determining how is each entity type linked to information about its latitude and longitude

[7]https://ruian.linked.opendata.cz/sparql

coordinates. This information will be used to make linking custom source classes to RÚIAN easier. Data publishers will be able to simply link their data to RÚIAN objects and the framework will automatically know how to retrieve the location data for any object.

The linked data model mirrors the structure of the underlying RÚIAN data shown in Figure 2.1. It adds names to classes and relationships among them, per linked data standards. A brief description of each entity type is provided below. Each entity type also contains information about its coordinates.

The naming of the entity links is consistent. The links fall under the `ruian` prefix and always lead from a more specific entity to the more general one. In terms of the diagram in Figure 2.1 it means that the link always leads upwards. For example, from `Adresní místo` (an address place) to `Stavební objekt` (a building) and to `Ulice` (a street). The name of the link is identical to the name of the class it leads to, only camel-cased. For instance, a link from `ruian:AdresníMísto` to `ruian:StavebníObjekt` would be `ruian:stavebníObjekt`.

The `ruian` prefix corresponds to the RÚIAN ontology URL[8]. When used in a SPARQL query, it must be first defined:

```
PREFIX ruian: <https://ruian.linked.opendata.cz/slovník/>
```

2.1.1.1 ruian:Stát

A country. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.2 ruian:Kraj1960

A region of the country, based on the region division system which was in place before the year 1960. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.3 ruian:Okres

A county in a pre-1960 region. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.4 ruian:RegionSoudrznosti

A region of the CZ-NUTS classification (Nomenclature of Units for Territorial Statistics). These regions are mostly formed by several pre-1960 regions merged together. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

[8] https://ruian.linked.opendata.cz/slovník/

2.1.1.5 ruian:Vusc

A subdivision of a `ruian:RegionSoudrznosti`. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.6 ruian:Orp

A subdivision of a `ruian:Vusc` which is administered by a single city. This entity does not have a location linked to it. Custom data must not be linked to instances of this class under the Andruian framework.

2.1.1.7 ruian:Pou

A subdivision of a `ruian:Orp`, a territorial unit administered by a single municipal office. This entity does not have a location linked to it. Custom data must not be linked to instances of this class under the Andruian framework.

2.1.1.8 ruian:Obec

A city. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.9 ruian:Ulice

A street of a city. This entity does not have a location linked to it. Custom data must not be linked to instances of this class under the Andruian framework.

2.1.1.10 ruian:ČástObce

A disctrict of a city. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.11 ruian:KatastrálníÚzemí

A cadastral territory belonging to a city. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.12 ruian:Zsj

An inhabitable unit belonging to a cadastral territory. The location is linked to instances of `ogcgml:Point` via the property path `ruian:definičníBod/ogcgml:pointMember`.

2.1.1.13 ruian:Parcela

A parcel located in a cadastral territory. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.14 ruian:Momc

A territory of a city district. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.15 ruian:Mop

A territory of a district in the city of Prague. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.16 ruian:SprávníObvod

A district in the city of Prague. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.17 ruian:StavebníObjekt

A building. The location is linked to instances of `ogcgml:Point` via the property `ruian:definičníBod`.

2.1.1.18 ruian:AdresníMísto

An address place. The location is linked to instances of `ogcgml:Point` via the property `ruian:adresníBod`.

2.1.1.19 ogcgml:Point

A place on Earth with a latitude and longitude coordinates. Instances of this type in RÚIAN have properties `s:geo/s:latitude` and `s:geo/s:longitude` linking to latitude and longitude coordinates of this point, respectively.

2.2 Framework functionality

This section gives a general overview of what the framework's functionality and use cases will be.

Publishing data with a location component may be done simply by publishing numeric coordinates with every data entry. Those coordinates can then be used to display such data on a map. For simple uses, this may be enough. However, this approach leaves out a lot of information about the location the data is pointing at. The latitude and longitude coordinates specify an abstract point on a map, they do not say anything about what the point represents, for example whether it is a building or a city. They do not provide any context.

The Andruian framework aims to provide a simple way for data publishers to link their data about real-world entities to the RÚIAN registry which contains representations of concrete places. Developers may consume and quickly

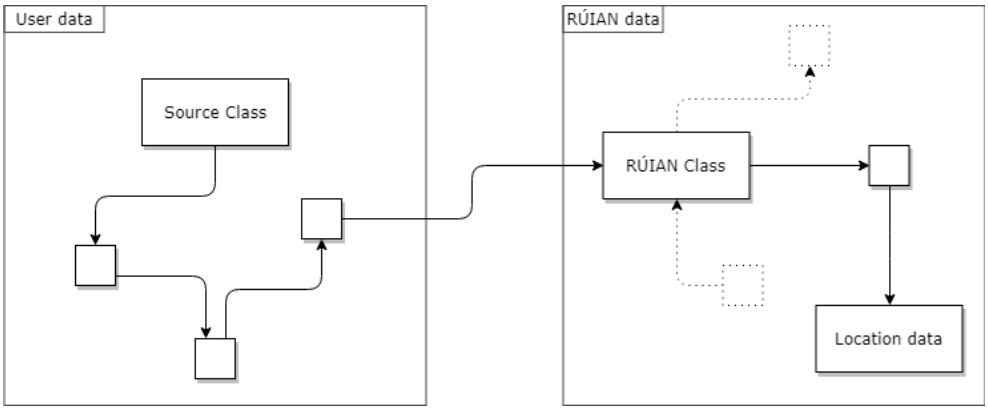

Figure 2.2: Model of source data for Andruian framework

search through this data using location queries and allow end-users to view the linked entities currently around them on their devices.

The diagram depicting the structure of the linked data to be processed by the framework is shown in Figure 2.2. The user of the framework (a data publisher in this case) has an arbitrary data model for their specific domain (left section). They want data consumers to be able to search through and display instances of the `Source Class`, which are linked to instances of the `RÚIAN Class` (right section). The path between the two instances may consist of several properties, forming a property path. The data publisher should only have to publish a basic configuration file for the framework, containing information such as the location of the two data sources and a path from `Source Class` to `RÚIAN Class`.

This configuration file will then also **CONSIDER: be** used by clients of the framework. The **CONSIDER: clients/users - sjednotit nebo vysvetlit, ze client je SW a user je clovek** clients only need to point to the published configuration file to find all the information they need to use the published data. They can query the framework for entities located around a certain position and, for example, view them on a map.

2.3 Naive solution

A straightforward solution to the aforementioned case only uses a single client application. The client, who wants to display all entities around them defined in a data configuration file hosted and accessible via an URL, needs to perform these steps:

1. Download and process the data configuration from the given URL. The configuration will, among other information, define an URL of where

the data source may be polled and a property path leading to a RÚIAN class.

2. Download all relevant entities from the data source.
3. For each of the entities find the linked RÚIAN object and download its location coordinates.
4. Filter all source entities based on their location (which was obtained in the previous step).
5. Display filtered entities.

This solution has the advantage of being fairly simple to set up by the data publisher. They only need to correctly specify the data configuration in addition to publishing their data and clients can take advantage of its connection to the RÚIAN registry.

However, the simplicity has a drawback in performance. Even when the user only wants to display places inside a small area, all source entities must be always downloaded and their location found through RÚIAN objects. Some, potentially most, of those entities will be ultimately filtered out for being outside the required area. This overhead may not be too severe with small datasets, but will increase with the size of the dataset. Furthermore, it strains the client both in terms of computation complexity and data transfer size. The amount of data transferred is especially important, given that the prototype implemented as a part of this thesis (and indeed a typical client as well) will be a mobile application.

For improved performance and smaller data requirements, a more complex solution which preindexes the source data is proposed in the next section.

2.4 Solution using an index server

The approach described in section 2.3 is not satisfactory for large datasets. A solution that is able to effectively handle larger amounts of data uses an index server. The server will consume the same data definitions as the client in the naive approach, which will make it optional - it will only be necessary to create the data definition once, regardless of the infrastructure chosen.

The index server will pre-process data and then serve it to clients without communicating with the data source or RÚIAN. The indexing and querying will include these steps:

1. Download and process data definition.
2. Download all entities specified in the definition from the user data source.
3. For each entity find the linked RÚIAN object and download its location coordinates.

4. Store the entities and their location internally in a way that allows for spatial indexing.

5. When queried by a client, retrieve entities from the internal storage based on a location query and return them to the client.

In order to allow for data updates, the index server will periodically perform reindexing. There are two modes of reindexing. A full reindex of a data definition will drop any existing data associated with the data definition and index the data from scratch. An incremental reindex will query the data source and exclude all the data it already has indexed. This way, only the data that was newly added to the source will be indexed. The incremental reindex will, however, not be able to pick up changes to existing data in the source. If it is likely that the data in a particular data source will be changing, full reindexing will be necessary.

The advantage of this solution is that the client will only receive data that is relevant for it. The query will also be faster as a result of the indexing capability and lack of any network communication between the index server and the data sources.

The disadvantage is the added infrastructure requirements on the data publisher. It is no longer enough to simply publish a linked dataset and a data definition. The data publisher or client application developer must also deploy the server and maintain it. Furthermore, in this setup the index server becomes a bottleneck, so enough resources must be dedicated to it to ensure it can handle the workload.

Figure 2.3 shows the component and communication diagram for both solutions. In both cases the part of the architecture that the data publisher is responsible for is highlighted.

A timely response is a crucial requirement on the framework, as it is intended to be used on mobile devices in real time. Therefore, despite the mentioned disadvantages, this solution will be preferred and an index server implemented.

The rest of this section covers the formal analysis of the proposed server. Firstly, the requirements on the index server are discussed and defined. Secondly, a use case model is created based on the identified requirements. Thirdly, the ontological entities are identified from the first two sections and a domain model of the problem is created.

2.4.1 Requirements

Based on the shortcomings of the naive solution identified above, we can construct formal requirements on the index server solution. The requirements are listed below, each assigned an identifier. The requirements should cover all functionality that is required on an abstract level, as well as specify where are the boundaries of the system and what it will not do.

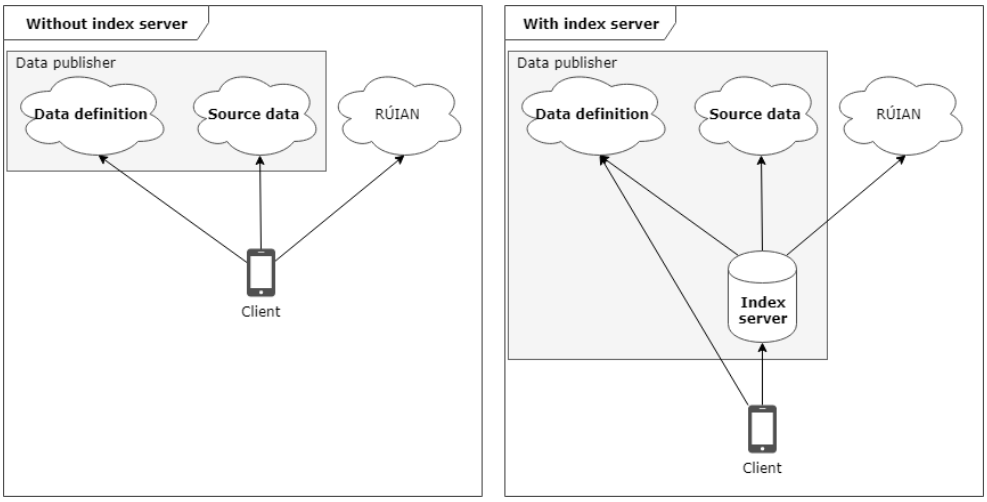

Figure 2.3: Architecture of solutions with and without an index server

F1 **Indexing.**

F1.1 **Spatial indexing.** The server will preprocess the data defined in the data configuration in a way that will make location-based querying over that data effective.

F1.2 **Repeated indexing.** The indexing process can be customized to run repeatedly at a specified time. The indexing should be incremental, unless required otherwise by the administrator.

F2 **Querying.**

F2.1 **Spatial querying.** The server will be able to answer spatial queries - respond with entities of a given class that are located around a given point inside of a specified radius. The server will also be able to list all stored entities or entities of a given class, regardless of the location.

F2.2 **Query response format.** The response of the server will be in a JSON format and will contain the entity IRI, the IRI of the associated location entity, a latitude and longitude coordinates, a human-readable label (if specified) and all selected properties defined in the data definition schema.

F3 **Administrator access.**

F3.1 **Indexing.** The server will provide a way for administrators to trigger indexing, either incremental, or complete.

F3.2 **Data definition management.** The server will allow administrators to add and remove data definitions.

F4 **Query visualization.** The server will provide a simple user interface for visualizing queries. Given a location query and object type, the user will be shown a map with the matching results.

N1 **External data configuration.** The server will be able to consume data sources configuration provided as linked data in a static RDF file using the **CONSIDER: Turtle, serialization. Ujednotit format, language, serialization v cele praci** TURTLE format. The configuration shall adhere to the Andruian data definition vocabulary defined in section 2.5.

N2 **Multiple data configurations.** The server will be able to handle several data configurations at once, in the format according to the requirement N1.

N3 **Data source.** The server will be able to process source data available via the SPARQL protocol.

N4 **Coordinate system.** The coordinates system (e.g. WGS84) used for spatial queries will reflect the coordinates system used in the data source. The indexing server will not do any format conversions.

N5 **Security.** The system will not provide a user-management functionality. A single user with a customizable username and password will be used as an administrator.

2.4.2 Use cases

A use case is a description of a concrete way an actor will be interacting with the system. Use cases are derived from requirements and represent a more detailed view of the requirement.

2.4.2.1 Actors

The actors in the system are shown in Figure 2.4.

1. **Consumer** is a person who wants to use the data made available by the Administrator. He or she will typically be a developer and will use the index server as a part of their application.
2. **Administrator** is a data publisher and the one setting up and running the index server. He or she configures the server with data definitions and other settings. He or she is a specialization of the Consumer and as such all actions available to a Consumer is available to an Administrator as well.

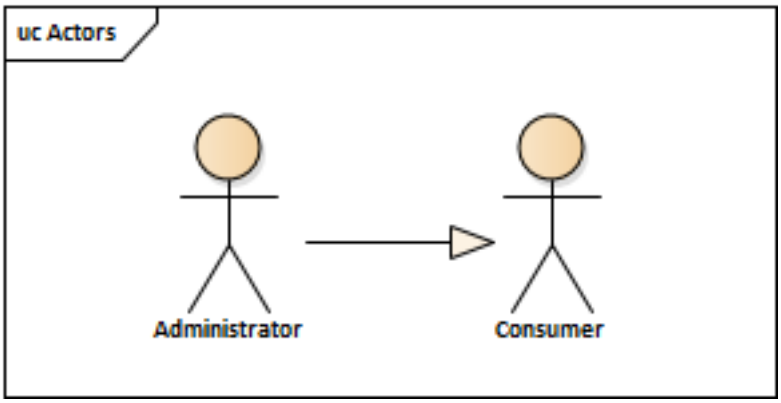

Figure 2.4: Actors of the index server

2.4.2.2 Use cases list

Identified use cases are shown in Figure 2.5. A more detailed description of the use cases follows below.

UC1 **Configure data definitions.** Instructs the server to index data described by a data definition. The definition file is a static TURTLE RDF file adhering to the Andruian data definition vocabulary. The file shall be accessible via HTTP/S. The data definitions may be added and removed while the server is running.

UC2 **Set up a repeated re-indexing task.** Makes the server repeatedly perform an incremental re-index of its configured data sources. A Cron expression [35] may be defined in the server configuration file, which is read on the server startup. This expression specifies when the re-indexing should be triggered.

UC3 **Trigger re-indexing.** Allows the administrator to manually trigger a re-index. This functionality will be available through a web UI and exposed via a HTTP endpoint. The administrator may choose to re-index incrementally, or to drop an existing index and completely recreate it.

UC4 **Add a data definition.** Allows the administrator to add a new data definition by entering a URL pointing to a RDF file. The file must be in the Turtle format and contain one or more data definitions using the schema defined in section 2.5.

UC5 **Remove a data definition.** Allows the administrator to delete any data definition file URL from the system to delete all associated data and stop it from being indexed.

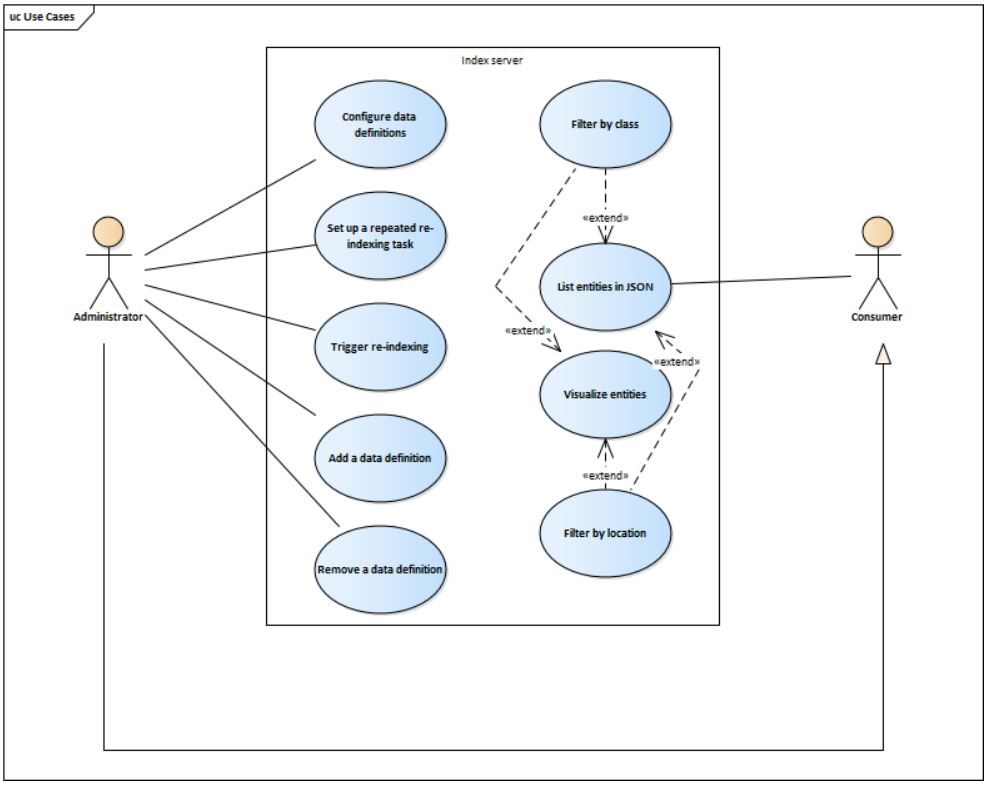

Figure 2.5: Index server use case diagram

UC6 **List entities in JSON.** Lists all entities indexed in the system in a machine-readable JSON format.

UC7 **Visualize entities.** Shows entities indexed in the system on a map.

UC8 **Filter by class.** Extends use cases UC6 and UC7. Filter entities based on their class.

UC9 **Filter by class.** Extends use cases UC6 and UC7. Filter entities based on their location. The user chooses a circular area through latitude and longitude of the center point and the circle radius and only entities located inside this circle will be listed.

2.4.3 Domain model

A domain model provides an abstract view of entities present in the system. It only contains domain-specific entities that can be identified in the previous steps of analysis, it does not contain any implementation details.

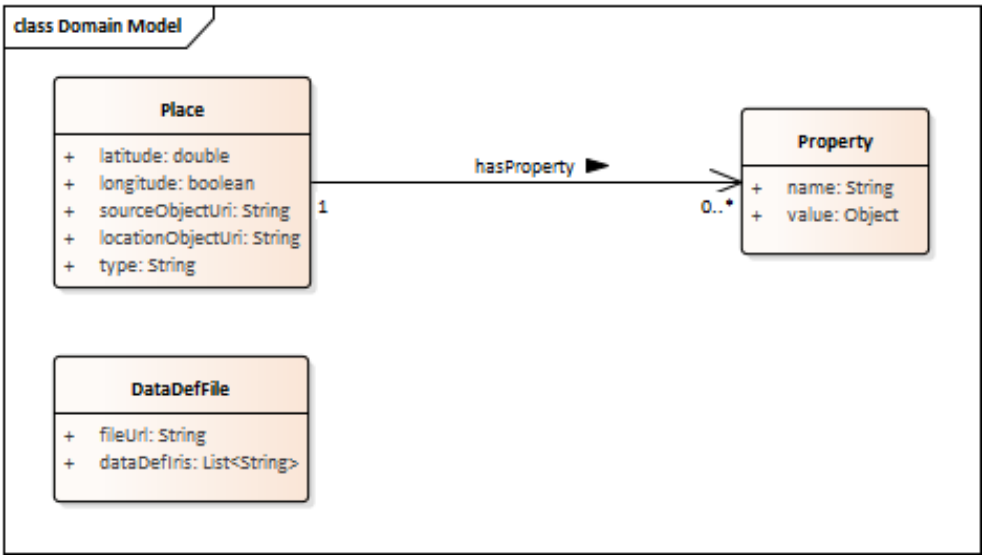

Figure 2.6: Index server domain model

The domain model is shown in Figure 2.6. Below is a textual description of what each entity represents.

Place is the main stored and indexed entity. A Place is created from an instance of the source class and an instance of the location class that is linked to it. It contains information about its location, URIs of the two objects (source and location) it represents, and its type (an URI identifying a subclass of `rdfs:Class`).

Property is an additional piece of information stored with a Place. A property corresponds to the `andr:SelectPropertyDef` class in the framework data definition schema. It may have a value of any type.

DataDefFile is a file hosted on the Internet at a particular URL. The file contains one or more data definitions for the server to index. This entity keeps a reference to the file URL and a cached list of data definition IRIs contained in it to avoid having to look them up.

The index server will have to be able to work with the data definition schema, so the domain also includes a representation of the schema defined in section 2.5. This is, however, provided by the parser library and does not need to be recreated here.

2.5 Data definition vocabulary

This section is concerned with the definition of an RDF vocabulary which describes the data definition schema. The schema is designed so that it conforms to the requirements on the framework informally discussed in previous sections. Firstly, a general structure of the vocabulary is discussed and afterwards a formal definition is presented.

2.5.1 Outline

There are two main components of the framework that need to be configured - the source (user) data and RÚIAN data. This thesis is using RÚIAN as the source of referential entities with location coordinates, but there is no reason this particular data source has to be used. Allowing users to configure their own data source as well as the location data source makes the framework extensible and reusable. For that reason, we will refer to RÚIAN data and its classes as *location data* and *location classes* from now on.

For both the source and the location data, the publisher will need to specify a class. That will define which source class is being mapped to which location class. They will also have to specify where the data can be found. The framework expects that both source and location data are available via SPARQL.

In order for the framework to be able to find the location classes' latitude and longitude, the configuration must provide information about how it is linked to the class. In other words, every class in the location dataset to which the data publisher may link their data must have an entry provided in the configuration. The entry will specify the property path from the given class to its coordinates. I expect that there will be a limited number of location data sources, which will be reused by data publishers. To that end, it should be possible to externalize this configuration and data publishers should be able to reuse it. One such externalized definition is listed in the appendix of this thesis, section B.4. It defines the property paths for all RÚIAN classes discussed in subsection 2.1.1. This definition is ready to be used by data publishers who use RÚIAN as their location source.

For the source data, the data publisher must specify the property path linking the source class to a location class. In addition to that, the publisher may specify a set of properties of the source class that should be downloaded or indexed along with the instances of the class. This will help data consumers find out what information about an entity is important and should be shown to a user.

And optionally, the data publisher may provide information about an index server where the data may be queried more effectively.

2.5.2 Formal definition

This section formalizes the outline of a data definition schema discussed before. The Andruian data definition vocabulary is described using the RDF Schema language [13]. The diagram of the schema is shown in the appendix in Figure C.1. The RDF file with the vocabulary in a Turtle format is available on the medium included with the thesis and on `http://purl.org/net/andruian/datadef`.

RDF Schema does not provide a mechanism for enforcing mandatory objects and properties. Nonetheless, the consumers of data definitions may assume that certain objects and properties are present in the dataset with certain cardinality. This requirement is shown in the schema diagram using UML domain modeling terminology. The same approach has already been used elsewhere, for example in the *DCAT-AP* specification [36]. All classes marked as *mandatory* must be present in the schema. Similarly, all mandatory properties of mandatory classes must be provided. Classes and properties not marked as mandatory are optional. Unless specified otherwise, the cardinality of properties is 1:1, meaning that a property may be defined for a particular subject only once. If the property is also mandatory, it must be provided exactly once for its subject.

The rest of this section describes elements defined by the vocabulary. Resources whose name starts with a capital letter are classes and resources whose name starts with a lowercase letter are properties, as per convention. For each class in the vocabulary, applicable properties are mentioned and their meaning described.

For easier understanding of the schema, it may help to inspect the vocabulary diagram in Figure C.1 and the provided example of a data definition file which uses this vocabulary. It is listed in the appendix, section B.3.

2.5.2.1 andr:DataDef

This is the root element of the data definition schema.

`andr:sourceClassDef` Mandatory property. Links to a definition of the source data.

`andr:locationClassDef` Mandatory property. Links to a definition of the location data.

`skos:prefLabel` Recommended property. A human-readable description of the data. Multiple may be provided, one per language.

`andr:indexServer` Optional property. Links to an index server configuration, if provided by the data publisher.

It must link to a source data and a location data definitions. Optionally it may link to an index server, if set up by the data publisher.

2.5.2.2 andr:SourceClassDef

This class defines the source data.

`andr:sparqlEndpoint` Mandatory property. Links to a SPARQL endpoint URL, where instances of the class described by this data definition may be found.

`andr:class` Mandatory property. Links to a Class.

`andr:pathToLocationClass` Mandatory property. Links to `andr:PropertyPath`. The path will lead from the class referenced by the `andr:class` property to the location class.

`andr:selectProperty` Optional property, may be specified multiple times. Links to a definition of an *important* property.

2.5.2.3 andr:SelectPropertyDef

This class specifies an *important* property of a class. It is a suggestion for the data consumer that it might be useful to show this property to an end-user. It also instructs the index server to cache this property with the index so that it can be served without needing to query the data sources.

`s:name` Mandatory property. Specifies a name that the value of this property should be referenced as by the data consumer.

`andr:pathToLocationClass` Mandatory property. Links to `andr:PropertyPath`. The path will lead to a resource or literal that should be used as *important*.

2.5.2.4 andr:PropertyPath

This class is equivalent to a *predicate* or *sequence* SHACL property path [37]. SHACL does not define a RDFS class for property paths, but only refers to them as to a plain `rdf:Resource` with a textual explanation. `andr:PropertyPath` class is defined in Andruian data definition schema to make the diagram and property domains easily readable.

A predicate or sequence path may be used wherever `andr:PropertyPath` is expected. A predicate path is a single property in place of the object; a sequence path is an RDF list containing two or more properties in place of the object. For example, in Turtle notation:

```
# A predicate path formed by a single property
:subject :predicate example:aProperty .

# A sequence path formed by three properties
:subject :predicate ( example:a example:b example:c ) .
```

2.5.2.5 andr:LocationClassDef

This class defines the location data.

`andr:sparqlEndpoint` Mandatory property. Links to a SPARQL endpoint URL, where instances of the class described by this data definition may be found.

`andr:class` Mandatory property. Links to a Class.

`andr:classToLocPath` Optional property, may be specified multiple times. Links to a specification of property path for a particular location class.

`andr:locationClassPathsSource` Optional property. Links to a source of `andr:ClassToLocPath` objects.

`andr:includeRdf` Optional property. Points to an URL where an RDF file is located and indicates that the contents of the file should be included in the configuration dataset while parsing it.

2.5.2.6 andr:LocationClassPathsSource

A source of `andr:ClassToLocPath` objects. Useful when the data publisher wants to use an external definition of paths from location classes to their latitude and longitude coordinates. This object can be defined in a separate RDF file. Data publishers may link to objects of this class and specify RDF files where they are located via `andr:includeRdf` property. This is shown in Figure 2.7. An instance of this class, a source containing all the paths of RÚIAN objects is provided as a part of this thesis. It is available in the enclosed CD, ready-to-use from a static website[9] and in the appendix, section B.4.

`andr:classToLocPath` Optional property, may be specified multiple times. Links to a specification of property path for a particular location class. The `andr:LocationClassPathsSource` will have one property per each class for which it defines a location property path.

2.5.2.7 andr:ClassToLocPath

A specification of property paths for objects of a particular class linking the object to its latitude and longitude coordinates. The coordinates pointed at by the property path shall be literals that are decimals or strings convertable to decimal numbers by applying the `xsd:float()` transformation.

`andr:class` Mandatory property. Specifies which class the property path applies to.

[9] `http://purl.org/net/andruian/location-sources/ruian`

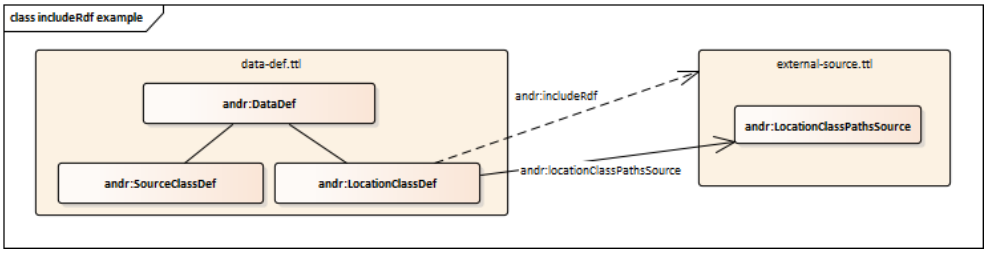

Figure 2.7: Schema of including a remote RDF file

`andr:lat` Mandatory property. Links to `andr:PropertyPath`. The path will lead to the specified class' latitude coordinate.

`andr:long` Mandatory property. Links to `andr:PropertyPath`. The path will lead to the specified class' longitude coordinate.

2.5.2.8 andr:IndexServer

A specification of an index server where the data described by the data definition may be queried effectively.

`andr:uri` Mandatory property. The URL where the index server is listening.

`andr:version` Recommended property. An integer literal representing the version of the index server. Its meaning is specific to clients and index servers.

2.5.2.9 andr:sourceClassDef

A property linking an `andr:DataDef` to `andr:SourceClassDef`.

2.5.2.10 andr:sparqlEndpoint

A property linking an `andr:ClassDef` to a SPARQL endpoint URL. There objects of the class defined in `andr:ClassDef` may be found.

2.5.2.11 andr:class

A property linking an `andr:ClassDef` to any `rdfs:Class`. This property specifies which class is described by the `andr:ClassDef`.

2.5.2.12 andr:pathToLocationClass

A property linking an `andr:SourceClassDef` to a resource that is a SHACL property path [37]. The property path leads from instances of the class specified by the `andr:class` property to instances of the location class specified in `andr:LocationClassDef`.

2.5.2.13 andr:selectProperty

A property linking an `andr:SourceClassDef` to `andr:SelectPropertyDef`.

2.5.2.14 andr:propertyPath

A property linking an `andr:SelectPropertyDef` to `andr:PropertyPath`.

2.5.2.15 andr:locationClassDef

A property linking an `andr:DataDef` to `andr:LocationClassDef`.

2.5.2.16 andr:locationClassPathsSource

A property linking an `andr:LocationClassDef` to an `andr:LocationClassPathsSource`. This property may point to a dataset separate from the rest of the data definition schema is located in. In that case, the external dataset must be an RDF file and it must be pointed to by the `andr:includeRdf` property.

2.5.2.17 andr:classToLocPath

A property linking to an `andr:ClassToLocPath`.

2.5.2.18 andr:includeRdf

A property linking an `andr:LocationClassDef` to an RDF file. When a data consumer is processing the data definition schema, they will import the data contained in the linked RDF file into their internal data model before resolving any other links leading from the `andr:LocationClassDef`.

This property allows for a limited extensibility of the data definition by including external sources. The intended use case is to include a separately published `andr:LocationClassPathsSource` object containing information about location classes the data publisher wants to link to. This case is depicted in Figure 2.7, where an object is located in an external file (*external-source.ttl*) and is being linked to by an object in the data definition RDF file.

Another possible solution to the problem of fetching RDF entities located in remote datasets is using a IRI Resolver. The resolver is a separate service which uses a database of known prefixes and locations to resolve entities located in an external dataset [38]. That is, however, more complicated to set

up as opposed to hosting a plain RDF file at a public server, which is why the schema defines this property.

2.5.2.19 andr:lat

A property linking an `andr:ClassToLocPath` to an `andr:PropertyPath`. The property path leads from instances of the associated class to a latitude coordinate, which is a literal of type `xsd:double`.

2.5.2.20 andr:long

A property linking an `andr:ClassToLocPath` to an `andr:PropertyPath`. The property path leads from instances of the associated class to a longitude coordinate, which is a literal of type `xsd:double`.

2.5.2.21 andr:indexServer

A property linking an `andr:DataDef` to an `andr:IndexServer`.

2.5.2.22 andr:uri

A property linking an entity to its URI. The meaning of this property depends on the entity it links from.

2.5.2.23 andr:version

A property linking to a numeric literal, denoting a version of the resource.

2.5.3 Data definition parser library

All data consumers will need to be able to read the data definition schema to understand the data. This includes the index server and the Android app prototype created as a part of this thesis, as well as any client application created in the future. For that reason, the data definition parsing functionality will be implemented as a standalone library, which will be able to transform raw RDF data to object structure similar to the structure defined by the data definition vocabulary.

This section provides an analysis of the needs on such a library.

2.5.3.1 Requirements

The requirements on the library are straightforward. It should be able to consume data definitions in RDF text format and convert them to native Java objects. The resulting object structure should reflect the data definition schema.

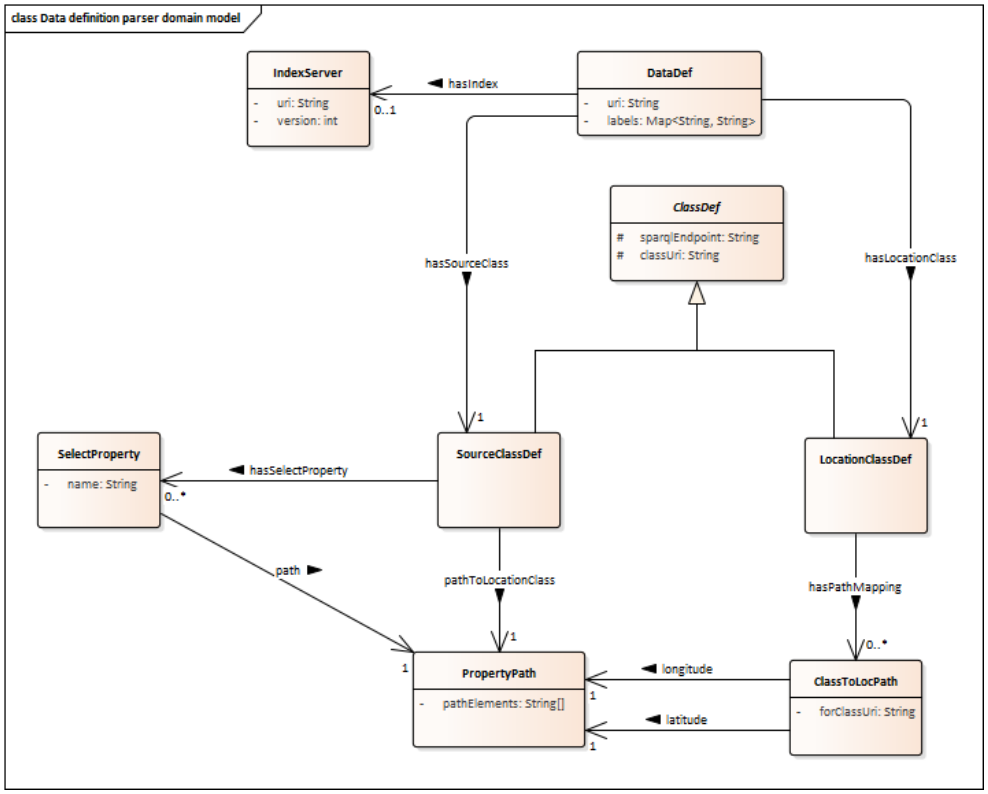

Figure 2.8: Data definition parser library domain model

A single RDF file may contain several data definitions. As long as they conform to the structure defined in section 2.5, the library will convert them all and return a set of disjoint object hierarchies.

2.5.3.2 Domain model

The domain model shown in Figure 2.8 closely resembles the data definition schema. Unlike the schema, it does not contain the `LocationClassPathsSource` class. Its function is to provide several `ClassToLocPath` objects, which, after being resolved, are linked directly to their `LocationClassDef`. The same applies for the `includeRdf` property. It is only useful while parsing and has no function afterwards, which is why it is dropped from the model.

The `DataDef` entity contains a map member which represents the human-readable labels for the data definition in different languages.

2.6 Android app

This section introduces the Android application prototype which will be using the Andruian framework. It is not intended to be feature-complete, but rather a useful starting point for developing customized applications on top of the framework. Firstly, the app requirements are formally defined and analyzed, after which the use cases and domain model are created.

2.6.1 Requirements

This section defines the requirements on the application. The requirements are purposefully vague and describe the overall functionality of the application.

F1 **Data visualization.** The app will allow users to visualize data published in terms of the Andruian framework. The data to be displayed may be restricted to a certain geospatial region.

F2 **Device location.** The app will work with the location of the device and allow displaying data based on it.

F3 **Remote data definition sources.** The user will be able to customize data to be shown by adding data definitions from remote URLs.

F4 **Optional index server.** The app will be able to fetch data from an Andruian index server, if such a server is defined. If it is not, the app will still be able to fetch non-indexed data.

2.6.2 Use cases

This section defines more fine-grained interaction scenarios of a user with the app in the form of use cases. The use cases are based on the requirements listed above.

There is only a single actor in the use case model, the app User. All identified use cases belong to them.

UC1 **Show data on a map.** The data is shown on an interactive map. Each data point has a marker.

UC2 **Show data around the device.** Extends UC1. The data shown may be restricted to the vicinity of the device. The range is customizable.

UC3 **Show data in a custom region.** Extends UC1. The data shown may be restricted to a particular user-defined region on the map.

UC4 **Customize data source map markers.** The color of the map markers can be changed to a user-defined color.

UC5 **Add a new data source.** A new data source may be added by specifying a HTTP URL pointing to an RDF file in Turtle format. If there are multiple data definitions, all of them are added.

UC6 **Remove existing data source.** An existing data source may be removed from the app.

UC7 **Hide and show data from a data source.** Data from any data source may be stopped from being shown on the map, without removing the data source itself.

UC8 **Show detailed information about a data point.** Each data point may be expanded to show detailed information about it, such as location coordinates, source and location object IRIs, a human-readable label (if provided), and all properties defined in the data definition.

UC9 **Navigate to a data point.** Each data point can be navigated to using a third-party navigation app, such as Google Maps.

2.6.3 Domain model

The domain model is very similar to the model of the index server, since both apps work with the same entities. Unlike the index server, the application will not need the `DataDefFile` entity, because once a data definition file is parsed, it will not be needed to keep track of the origin of the data it contains. The application will work with the data definition class hierarchy provided by the `ddfparser` library described in section 3.1. Other domain classes are `Place` and `Model`, as described and shown in a diagram earlier, in section 2.4.3.

Chapter 3

Design

This chapter covers the design of the framework components. Firstly the design of the data definition parser library is created. Secondly, the index server is designed. Its design was created in two phases, after the initial one proved to be insufficient for large data. Thirdly, the design of the Android application is worked out.

3.1 Data definition parser library

This section discusses the design of the data definition parser library, named `ddfparser`. Firstly, the libraries used for implementation are chosen. Then the model of the parser component is created.

3.1.1 Technology

The library will be used both by the index server and the Android app. They will both be built on Java, so the library will be implemented in Java as well. It requires an external library for RDF parsing and because it will be used by the Android app, the library has to be compatible with it.

Available libraries for RDF manipulation were introduced in section 1.3. Out of the three RDF4J will be used, because it supports the most serialization formats (including Turtle, unlike the Semargl library). RDF4J will not work on Android out-of the box due to the Android Runtime (ART) is missing certain packages for working with XML. However, that is fairly straightforward to fix, as is explained later in subsection 4.1.1. That way the latest version is always available and the library does not have to use ports of Jena which are several years old.

For easy distribution and usage, the library will be packaged and available as a Maven artifact.

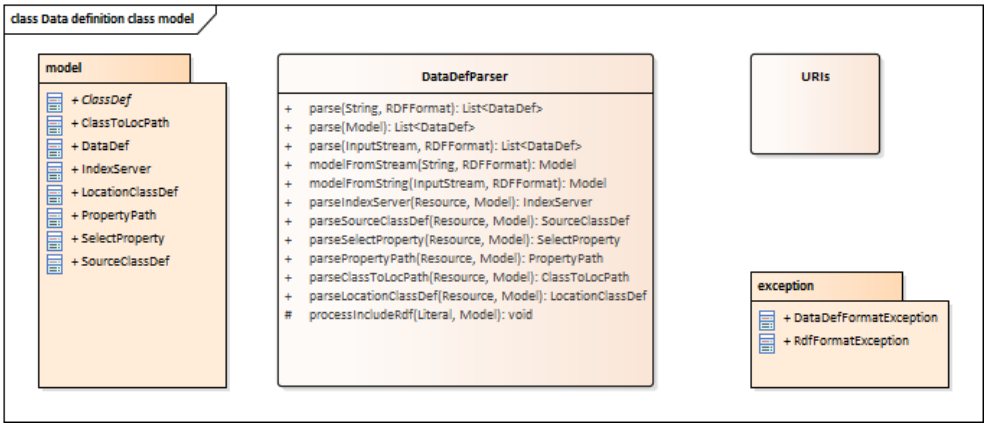

Figure 3.1: Data definition parser library class model

3.1.2 Class model

The class model of the library is depicted in Figure 3.1. All of the user-facing functionality of the library is present in the `DataDefParser` class. The `parse()` methods will take raw RDF text, an input stream or an RDF4J model as a parameter, and return a list of `DataDef` objects, which will be the roots of domain class hierarchies described in the previous section. Those objects are located in the `model` package.

The parser class also provides methods for parsing parts of the data definition schema. Those methods take a RDF4J node and a model as arguments, and return appropriate Java objects constructed from the node.

During the parsing two checked exception types may be thrown:

`RdfFormatException` is thrown when the format of the given text is not valid and cannot be parsed by RDF4J.

`DataDefFormatException` is thrown when the format is correct, but the structure of the dataset does not conform to the Andruian data definition schema.

The library supports parsing any RDF format that RDF4J supports. The `parse()` methods accept a constant defined by the RDF4J API which identifies the RDF serialization format that should be used.

The `URI` class is where all URIs needed for parsing are defined as constants.

3.2 Index server

This section is concerned with the design of the index server discussed previously in section 2.4. Firstly, a language and frameworks used for the imple-

mentation are chosen. Then the process of indexing data is discussed. This includes the choice of a persistence solution which will allow for effective spatial querying. Then the architecture of the solution is outlined and the HTTP API that the clients may use to communicate with the server is defined. Next, the configuration options of the server are briefly described. Lastly, the feature of server-side clustering is introduced, which was found to be necessary after the initial prototype was implemented and tested in subsection 5.3.2.

3.2.1 Language and frameworks

The index server will be written in Java. Java is a mature, widely adopted language with many resources and libraries and can run on any platform where JVM is available.

The server will need to be able to read RDF files and parse the Andruian data definition schemata. This will be handled by the parser library described in subsection 2.5.3.

A framework used by the server will be the Spring Framework [39]. It lets the application use the dependency injection pattern, which simplifies modularization of source code. The framework also includes web support for serving content over HTTP. Finally, it provides data persistence options integrating with many popular data storage systems.

3.2.2 Indexing

A core functionality of the server is the spatial indexing of the configured data. Firstly, the process of fetching data to be indexed is discussed. Secondly, approaches for indexing and storing the data are discussed.

3.2.2.1 Indexing process

The indexing process will be realized through SPARQL queries. First, the server fetches data definition files from the configured locations. The definition files are not large so repeated network requests should not be an issue. Repeated downloading ensures that in the case of any data definition change, all index servers that consume it will pick up the change automatically. The downside is that the file must be reachable at the given URL every time the indexing takes place.

After being downloaded, the data definition file needs to be parsed and understood. Among others, it contains information about where the SPARQL endpoints for *Source* and *Location* classes are.

The server will then build a select query to be sent to the *Source* SPARQL endpoint. The query selects all required properties, i.e. the URI of the object, its type, label, the URI of the associated location object, and all extra properties listed in the data definition. The query will contain a federated query [40]

which will issue a sub-query to the *Location* SPARQL endpoint and retrieve the location information of each object.

When reindexing incrementally, a `FILTER` clause is inserted into the query. It filters out all source objects that already exist in the index server database based on their IRIs.

The selected properties listed in a data definition are considered optional, which means that all objects matching the definition will be indexed, regardless of them having or missing any of the properties. The client consuming data from the index server must handle the cases when expected properties might be missing. The query template is available in the appendix, section B.1.

After the data definition schema is read and the appropriate data is fetched and transformed, it needs to be stored in a way that makes spatial queries over it effective. The possible solutions for indexing, storing and querying spatial data are discussed in the next sections.

3.2.2.2 SPARQL spatial querying

One possible approach to storing and querying spatial data is the family of techniques introduced in section 1.4. Those techniques extend SPARQL with various predicates that allow for geospatial filtering in the queries. Only some triplestore engines implement each of the extensions, however.

The functionality required by the index server is searching for triples around a certain point in a certain distance. All mentioned techniques offer this functionality, and Jena Spatial explicitly shows how to speed queries up by using an external index server. We will focus on it in the rest of this section.

Jena Framework provides support for spatial SPARQL queries via the `jena-spatial` module [23]. The data storage Jena-spatial uses is TDB, a *triple store*, i.e. a database for RDF graphs. That alone would not perform well for spatial queries so Jena also uses Lucene or Solr to index spatial information about the data stored in TDB. The diagram is shown in Figure 3.2.

An advantage of this approach is the fact that the SPARQL server, Jena Fuseki, can use TDB as its data storage. Users then can access this indexed data over HTTP using SPARQL and run custom queries on it. They are not bound by the data format of the responses from the index server.

Unfortunately, at the time of writing, the documentation for Jena Spatial is not particularly thorough and the only Java example code linked to crashes at runtime.

Furthermore, serving data to clients directly from a SPARQL endpoint would not allow the index server to perform any custom query processing. That proved to be necessary after the initial implementation, because the Android client was not able to handle large amounts ($\tilde{1}00.000$) of data points and so the data points in the query response had to be pre-clustered by the index server.

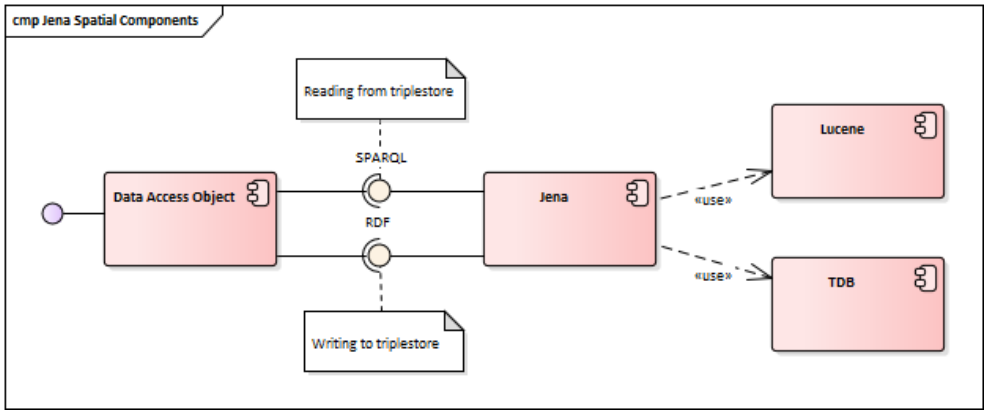

Figure 3.2: Components of Jena Spatial

3.2.2.3 Solr

Another approach to the persistence or, in other terms, indexing layer, does not use Jena to store data. This approach converts the data returned from the index query to Plain Old Java Objects (POJOs) and those are then saved in a storage that supports spatial indexing. Solr is one such storage system that is robust, battle-tested and has a great community and resources.

Solr documentation states that two modes of operation are supported - a schema-based and schema-less. The schema-less mode is more flexible - the data types of unknown fields are guessed based on the content of the first such field encountered. However, this is not recommended for production use and when a field type is guessed incorrectly, subsequent writes may cause errors due to the guessed data type being incompatible with the value to be inserted. The preferred mode of operation is one where a schema is defined. The schema states details of fields of objects stored in the database. A field is identified by a name and has a data type. Each field may optionally be indexed.

Because each indexed place may have any number of additional properties, a dynamic behavior is necessary. Solr schema offers *dynamic fields*, which are similar to regular fields, but instead of having a name, they have a wildcard pattern. Any field whose name matches the pattern will be indexed as defined by the dynamic field settings.

The relevant section of the Solr data schema for the Andruian index server follows:

```
<field name="location" type="location_rpt"
           indexed="true" stored="true"/>
<field name="type" type="string" multiValued="false"
           indexed="true" stored="true"/>
<field name="iri" type="string" multiValued="false"
```

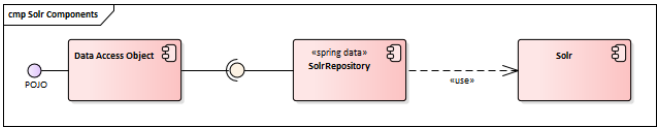

Figure 3.3: Indexing using Solr and MongoDB

```
            indexed="true" stored="true"/>
<field name="locationObjectIri" type="string" multiValued="false"
            indexed="true" stored="true"/>
<field name="label" type="string" multiValued="false"
            indexed="false" stored="true"/>
<dynamicField name="*_dynstr" type="string" multiValued="false"
            indexed="false"  stored="true"/>
```

- Field `location` contains latitude and longitude as plain string in the format `lat,long`. Solr knows how to parse this format and index based on it. Type `location_rpt` is required over the type `location`, because it allows for heatmap faceting. That is a Solr feature used for server-side clustering discussed in subsection 3.2.7.

- Field `type` contains the type of the object - IRI of its RDF class. It is indexed so that queries selecting a particular place type may be executed.

- Field `iri` contains the unique IRI of the place.

- Field `locationObjectIri` contains the IRI of the location object that is linked to the stored object.

- Field `label` contains a human-readable name of the place.

- Field `*_dynstr` matches all fields with the `_dynstr` suffix. It is used to store all additional properties associated with the object.

The whole Solr configuration schema is provided on the enclosed medium and on the indexer GitHub page [41].

Spring Framework, which is used by the index server, offers support for communicating with Solr. The developer only needs to define an interface with method declarations following a certain pattern and the framework does the heavy lifting of implementing those methods. That simplifies the implementation of the persistence layer. The diagram depicting this setup is shown in Figure 3.3.

This setup, leveraging Solr, will be used in the implementation of the index server.

3.2.3 Architecture

The general architecture of the index server is straightforward. It can be thought of as a three-layer architecture. The first layer is the controller layer, which handles interaction with users over HTTP. It processes the HTTP requests and calls appropriate methods on classes in the service layer. The service layer then processes data and stores them using the persistence layer.

The persistence layer leverages Solr for indexing places and uses MongoDB to store server configuration data. The communication with the triple store where source linking data is located is facilitated by SPARQL over HTTP. This allows the triple store to be at a separate location than the index server. The communication should not pose a major overhead, because the majority of time it takes to receive a response is spent by the triple store evaluating the query internally. I have analyzed the option of allowing the index server to use Jena's TDB triple store data storage directly. However, the documentation states that this is advised against:

> *A TDB dataset should only be directly accessed from a single JVM at a time otherwise data corruption may occur. (...) If you wish to share a TDB dataset between multiple applications please use our Fuseki component which provides a SPARQL server that can use TDB for persistent storage and provides the SPARQL protocols for query, update and REST update over HTTP [42].*

The component diagram in Figure 3.4 shows a broad structure of the server application. This view works on a higher level of abstraction; the components do not directly correspond to classes and packages but show a logical structure of the system [43]. The Controller component forms the controller layer, the DAO component forms the persistence layer and the rest of the components form the service layer. Components outside of the bounding box are external to the application.

The *Controllers* component is responsible for handling communication over HTTP. That includes displaying a GUI for browsers and responding to HTTP API queries. It processes the incoming request and delegates business logic to the *Services* component. *Services* use *DAO* for persistence-related functions and *SparqlQueryBuilder* to create queries to be sent to a Sparql endpoint via the *Net* component. The *DataDefFetcher* is responsible for fetching data definition files by getting raw data from a server via the *Net* component and processing the response by using the *ddfparser* library.

3.2.4 Class model

A more fine-grained view of the system is shown in Figure 3.5. A class diagram shows classes and their packages in the system. This section describes the purpose of each package and classes.

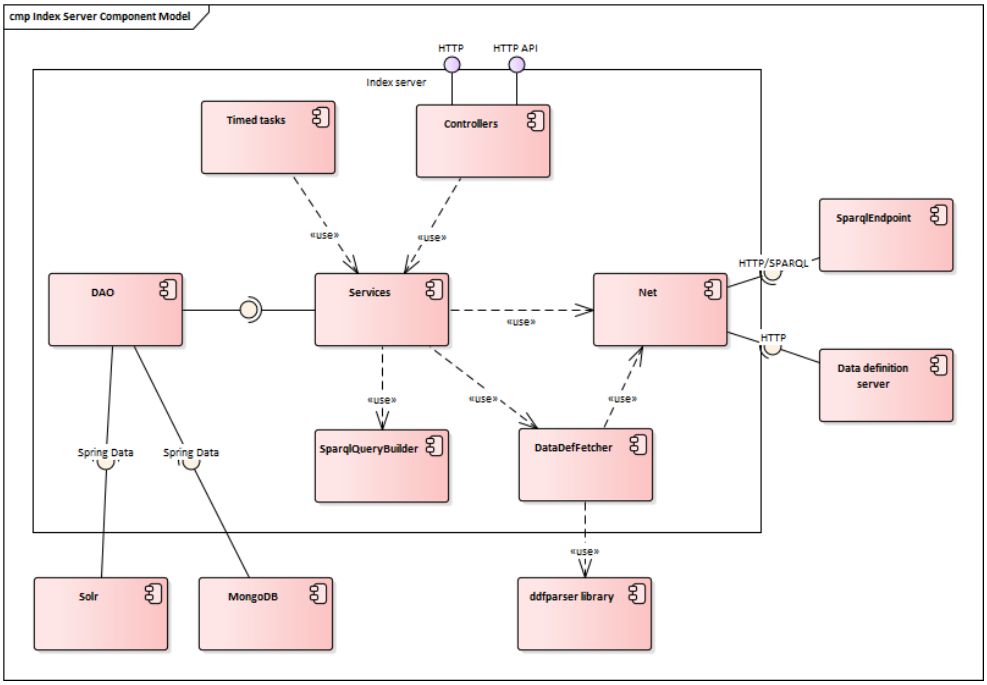

Figure 3.4: Component diagram of Andruian index server

3.2.4.1 `controller` package

This package is responsible for handling HTTP requests. It is divided into two more packages - `rest` and `ui`. The former contains classes which provide API functionality for client applications, described in subsection 3.2.5. The latter deals with the GUI for human users.

These classes are annotated as `@Controller` for the Spring MVC framework.

3.2.4.2 `service` package

This package provides the business logic. `IndexService` handles all index-related operations and `QueryService` performs queries on behalf of the Controller layer. `PostStartupService` is only invoked after the server starts, to perform initialization tasks, such as indexing or reindexing of configured data definitions. `IndexCron` fires repeatedly, as often as defined by the user.

3.2.4.3 `net` package

The `net` package handles tasks related to Internet communication. The `DataDefFetcher` downloads RDF files and uses the `ddfparser` library to convert them to native

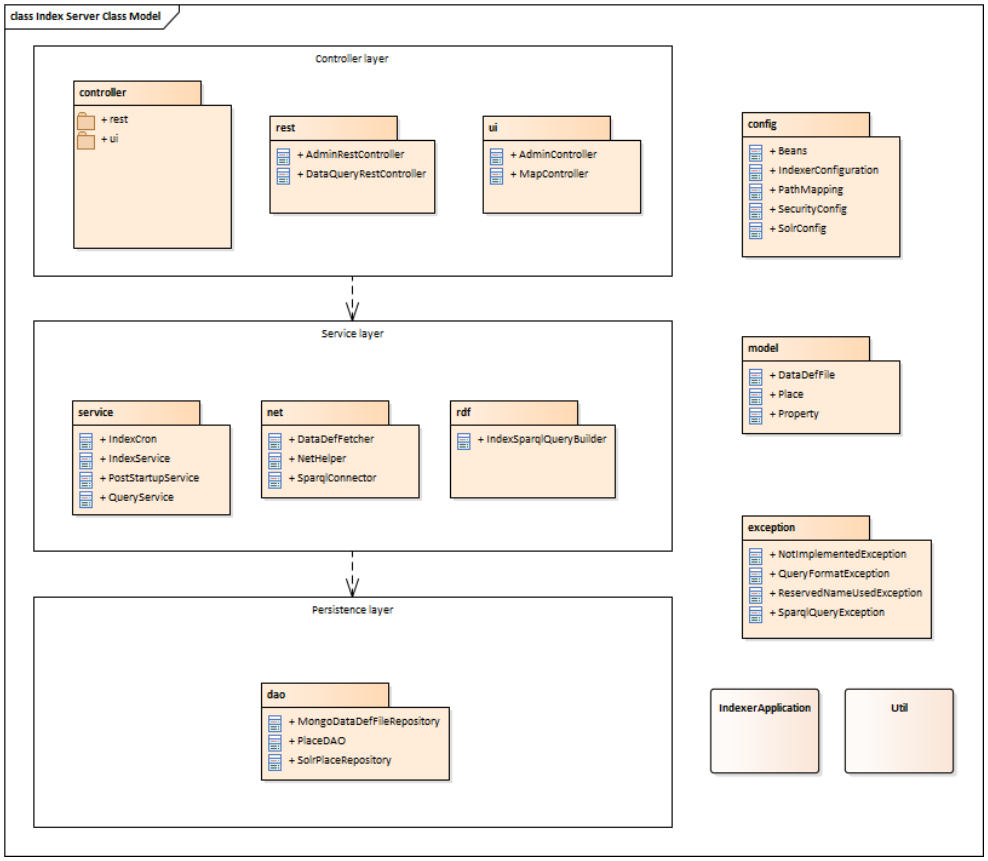

Figure 3.5: Class diagram of Andruian index server

Java objects representing the data definition. `NetHelper` is a utility class providing a simple interface for HTTP requests. `SparqlConnector` is responsible for performing SPARQL queries and processing their results.

3.2.4.4 `rdf` package

A single class belongs to this package, `IndexSparqlQueryBuilder`. It is used to create SPARQL queries based on data definitions. The queries are then sent to endpoints through other components and the returned data is indexed.

3.2.4.5 `dao` package

This package provides data persistence. The entities that need to be persisted are Place and its properties, and configuration-related entities, such as a list of URLs to data definitions. All functionality related to data persistence is accessible via `PlaceDAO`.

3.2.4.6 `config` package

This package contains definitions of Spring Beans, Spring Web security configurations and others.

3.2.4.7 `model` package

The model classes are located in this package.

3.2.4.8 `exception` package

All custom Exception types used by the index server are located here.

3.2.4.9 `root` package

The `IndexerApplication` is the main class of the Spring Context. `Util` contains miscellaneous utility functions.

3.2.5 HTTP API

This section describes HTTP API endpoints exposed by the index server.

3.2.5.1 `GET /api/query`

Perform a data query based on the given parameters. Returns a JSON list of objects, each object describing one indexed place.

Example:

```
http://localhost:8080/api/query
    ?lat=49.74468693637641
    &long=13.37622390978595
    &r=1.011
    &type=http%3A%2F%2Fexample.org%2FAClass
```

Parameters:

lat Floating point number. Latitude of the spatial search center. If used, then all three of `lat`, `long`, `r` must be used.

long Floating point number. Longitude of the spatial search center. If used, then all three of `lat`, `long`, `r` must be used.

r Floating point number. Radius in kilometers. If used, then all three of `lat`, `long`, `r` must be used.

type String. If provided, only show objects of the given RDF class.

count Boolean. Defaults to 0, enable by 1. If true, only return the number of objects that would be returned by the query and do not actually return the objects. (`responseType` 0)

Response formats Each response has a type identified by the `responseType` key and a body identified by the `responseBody` key. The types are following:

0 When only the number of places is being returned.

1 When places matching the query are being returned.

Below are listed JSON examples of the responses.

```
{
  "responseType": 0,
  "responseBody": 42
}
```

```
{
  "responseType": 1,
  "responseBody": [
    {
      "iri": "http://src.com/https%3A%2F%2Fruian.linked.opendata.cz
                                %2Fzdroj%2Fadresní-místa%2F25821318",
      "type": "http://example.org/SourceObjectA",
      "locationObjectiri": "https://ruian.linked.opendata.cz/zdroj/
                                            adresní-místa/25821318",
      "label": "Americká 1",
      "properties": {
        "StreetName": "Americká",
        "PSC": "12000",
        "StreetNum": "1"
      },
      "latPos": 50.070746,
      "longPos": 14.439492
    },
    ...
  ]
}
```

iri is the unique identifier of a particular place.

type is the place's RDF type (i.e. its class).

locationObjectIri is the IRI of the location object that the place is linked to.

label is a human-readable label of the place, taken from the `skos:prefLabel` or `s:name` RDF property, if exists. If no label can be determined, this field will fall back to the place IRI.

properties contains a list of name-values. Names are the names of properties as defined in a data definition and values the corresponding values of those properties.

latPos, longPos are location coordinates of the place.

3.2.5.2 `POST /api/admin/reindex`

Trigger a reindex of all or a particular data definition.
This request requires a basic HTTP authorization of the admin user defined in the application configuration.
Body parameters:

dataDefUri String. If provided, only the data definition with this URI will be reindexed. It must be defined in the application config. If not provided, all data definitions are reindexed.

fullReindex Boolean. Defaults to 0, enable with 1. If enabled, all existing data for the data definition being reindexed is dropped first, and the whole index re-built.

3.2.5.3 `GET /api/admin/datadefs`

List the URLs of all data definition files being indexed by the server and the data definitions they contain.

No parameters.

Response format

```
[
  {
    "fileUrl": "https://example.com/example-datadef.ttl",

    "dataDefIris": [
      "http://foo/dataDefA"
    ]
  }
  ...
]
```

3.2.6 Configuration

Some aspects of the index server are configurable. The configuration is provided by creating a file named `application.properties` in the same folder as the runnable jar file. These are the supported properties:

indexing.cron A quartz cron expression [35] defining when to run incremental reindexing.

indexing.onstart If true, an incremental reindex will be run on application startup.

db.solr.url URL to a Solr server, e.g. http://localhost:8983/solr

db.solr.collection Name of a collection or Solr core to use to store objects.

spring.data.mongodb.uri URI of a MongoDB database.

admin.username A username for the admin account.

admin.password A password for the admin account.

server.port A port for the server to listen to when running in the embedded mode.

logging.file A path to the log file.

The accepted file format is Java Properties Format [44], for example:

```
indexing.cron = 0/30 * * * * *
indexing.onstart = true
admin.username = JohnDoe
admin.password = secret
```

3.2.7 Server-side marker clustering

After the index server and Android client app were implemented based on the initial design, the performance of the app was lacking when it came to showing a large number (tens of thousands) of places. Even though the client application was able to cluster places together to avoid cluttering the map, the place data still had to be kept in memory. Based on the number of places required to show, this made the app sluggish, unresponsive and memory-intensive to the point it would crash due to running out of heap space. This is discussed further in subsection 5.3.2.

To mitigate that, we introduce the option to perform server-side clustering of places in the event that the number of the places in a response should be too high. This functionality is available through a new parameter to the data query endpoint previously defined in subsubsection 3.2.5.1:

clusterLimit Integer. Specifies the upper limit of how many places may be returned without being clustered. If not provided, they are never clustered.

If the number of places is lesser than the limit, a response of type `1`, described earlier, is returned. Otherwise, a new response type identified with number `2` is returned. The body of the response is a list of clusters. Each cluster contains its position and the number of places that belong to it:


```
{
  "responseType": 2,
  "responseBody": [
    {
      "placesCount": 5,
      "latPos": 50.07072687149048,
      "longPos": 14.455454349517822
    },
    {
      "placesCount": 3,
      "latPos": 50.070555210113525,
      "longPos": 14.457128047943115
    },
    {
      "placesCount": 1,
      "latPos": 50.07042646408081,
      "longPos": 14.455368518829346
    },
    {
      "placesCount": 2,
      "latPos": 50.07038354873657,
      "longPos": 14.455196857452393
    }
  ]
}
```


Internally, the clustering functionality of the index server leverages Solr's Heatmap Faceting feature. That allows clients to request a heatmap of spatial data covering a specified geographical area. A heatmap is essentially a grid projected over a portion of a map, with each cell of the heatmap aggregating data from its region. The size of the region is calculated by Solr in a way so that the number of grid cells forming the heatmap is constant, regardless of the size of the area the heatmap covers[45].

The designed architecture of the index server does not need to change much to accommodate for this added functionality. A new class is introduced into the `model`, representing a cluster of places. A new Data Access

Object, `ClusteredPlaceDAO` is added to the architecture to facilitate faceted querying. The *Spring Data for Apache Solr* component used for non-faceted communicating with Solr does not support the full extent of query options required by the faceted queries, nor is it able to deserialize the Solr response into cluster objects. For that reason, the `ClusteredPlaceDAO` communicates with Solr through custom-crafted HTTP queries and deserializes its responses into `PlaceCluster` objects.

The details of this process are discussed further in the thesis, in subsection 4.2.4.

3.3 Android app

This section covers the design of an Android application prototype which was outlined in section 2.6. First the language, libraries and frameworks used are described. Then the UI/UX is designed using lo-fi mockups for each application screen and logical connections among them. The next section talks about the broad architecture of the application, which is then described in detail in the last section.

3.3.1 Language and frameworks

There are two officially supported languages for Android - Java and Kotlin. Both languages are built on JVM and compile into JVM bytecode. They are interoperable, meaning that one can directly call the other. The app will be written in Java, as at this point in time I am more familiar with it.

The app will use several common framework and libraries.

OkHttp [46] is used as a HTTP client. It's stable and has an easy-to-use API.

Butterknife [47] is a small library which helps reduce boilerplate code when inflating Android Views. By using Java annotations, views and constants are automatically bound to their variables and lookups do not have to be implemented by hand.

Room [48] is a persistence library, providing an abstract layer over the native Android storage system, SQLite. The library is similar to ORM tools, such as Hibernate, but better suited for mobile apps. It is able to seamlessly save and load data classes with primitive attributes. It does not, however, automatically load whole object hierarchies. The reason is, according to the library creators, the limited resources of mobile devices and the need for responsiveness. Quite often the app only needs to load and work with a particular object and does not need to load all the objects it references. One solution to this issue is lazy loading, but

the way the Room library works is more low-level. Through Java annotations child entities define some of their attributes to be foreign keys of their parents. This allows the framework to check integrity constraints and delete child relations from the database when their parent is deleted. If a referenced object needs to be retrieved, it must done so manually by calling the appropriate database query method. The library generates code communicating with the database based on user-defined interfaces and SQL query templates.

Mockito [49] is a mocking framework for unit testing in Java. It allows for fine-grained testing thanks to replacing dependencies of objects under test with friendly fake objects with pre-determined, simple behavior.

Espresso [50] is a testing library for automated UI tests. It can simulate a user interacting with the app under test and make assertions about what should be happening on the screen of the device. It is intended to be run on a real device or an emulator and for that reason it is slower than simple unit tests.

3.3.2 User Interface

In this section the UI/UX design of the app is discussed. A mockup of the application flow and its screens is created to visualize the general look and feel of the app.

The first section below introduces the color schema of the application. Each following section below covers one of the screens and describes its purpose. The whole user interface flow is shown in a diagram in the appendix, Figure C.3.

3.3.2.1 Color scheme

A color scheme is an important visual feature of an Android application user interface. With Android 5.0 Lollipop Google introduced the most comprehensive set of guidelines on how to design Android applications. The guidelines define a so-called called Material Design, which describes the user interface as an abstraction of sheets of paper and ink [29]. Applications designed in accordance to the Material Design philosophy are primarily defined by three colors. The primary color is the most ubiquitous one, used for toolbars throughout the application. A primary dark color is a darker version of the primary color, intended to look as if the primary color was covered by a shadow. Finally, an accent color is a color that should complement the primary color, but be different enough to stand out. It is used for important buttons and visual elements that should draw the user's attention.

I created the color scheme using Adobe's online color wheel tool [51]. The wheel helps with finding a palette of complementary colors that are aestheti-

Figure 3.6: The color scheme of the ViewLink app.

From top: primary, dark and accent color.

cally pleasing together. The color scheme used in the Android application is shown in Figure 3.6. The topmost color is the primary color, the middle is its dark version and the bottom color is the accent.

3.3.2.2 Main screen

The home screen is the main screen the user will interact with. It is formed by a fullscreen map which displays markers of the enabled data definitions and the user's location. The map is interactive and can be dragged around. It will center on the user's location and follow them by pressing the GPS indicator button in the bottom right corner.

The places shown will be automatically refreshed when necessary as the map moves. A potential refresh event will be generated on each map movement, which may be automatic, when following the device location, or manual, triggered by the user dragging the map. The event will lead to refreshing the places if the current map viewport is sufficiently different than the last viewport the places were updated for. This will be determined by creating a sub-viewport, a rectangle, every time the map is moved, and calculating whether this whole sub-viewport is visible in the last-updated viewport. If it is, no update is needed. Otherwise, a refresh will be triggered and the last-updated viewport replaced with the new one.

Clicking a map marker will bring up a pop-up label describing the label. Clicking on this label will show a screen with detailed information about the given place, discussed in subsubsection 3.3.2.3.

Clicking the burger menu icon on the top left will expand a drawer menu from the left side of the screen. The menu will contain a list of all data sources configured in the app with a switch which can be used to enable or disable any data source. On the bottom of the list a button is located, which opens the data source configuration screen.

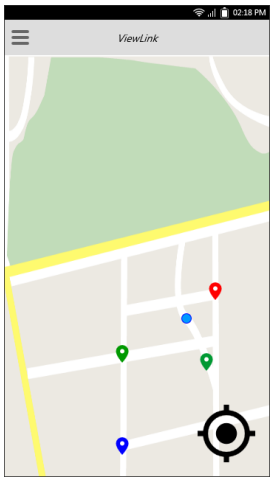

Figure 3.7: The home screen of ViewLink app

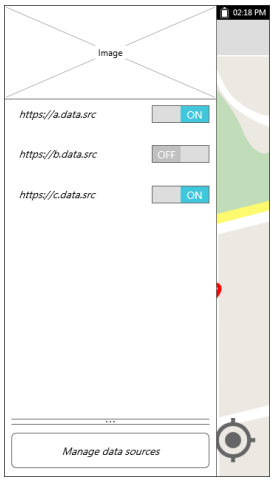

Figure 3.8: The home screen of ViewLink app with drawer menu open

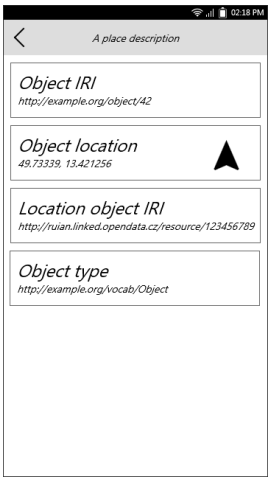

Figure 3.9: The place details screen of ViewLink app

3.3.2.3 Place details screen

This screen, shown in Figure 3.9, shows all details about a place defined by its data source. Some properties are default and defined for all places, and some properties can be defined by the data publishers through the Select Property mechanism in the Andruian data definition schema. This screen shows all that data in text form. When a value contains a link, clicking that link will open a browser with it.

3.3.2.4 Data sources screen

The data sources screen shown in Figure 3.10 lists all data sources registered in the app. Each of them shows a caption or a link to the data source, information about the source and location object class and its assigned color.

Clicking the color circle will bring up a color picker dialog, where the user can change the marker color for this data source. This dialog is shown in Figure 3.11.

Clicking the FAB on the bottom right of the screen brings up screen for adding a new data source, which is discussed in subsubsection 3.3.2.5.

3.3.2.5 Add data source screen

This screen, shown in Figure 3.12, contains a single text field, where the user enters a URL to a data definition file. Tapping the FAB downloads the file pointed to, parses it and adds one or more new data definitions to the app configuration, or reports an error with the communication or parsing.

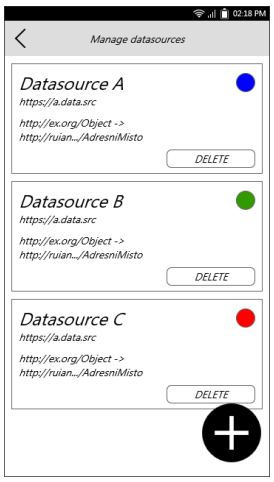

Figure 3.10: The data sources configuration screen

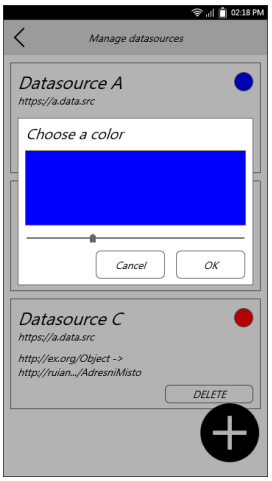

Figure 3.11: The data source color choosing dialog

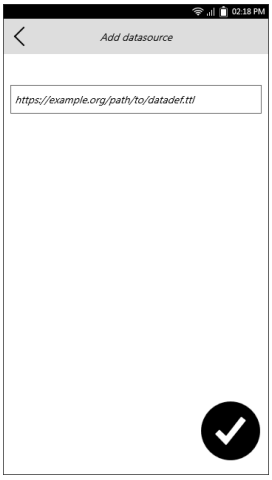

Figure 3.12: The add data source screen

3.3.3 Architecture

A common Android architecture follows the Model-View-Presenter pattern. In this architecture, a View is a plain object from the Android framework, such as an Activity. It implements a View interface and is only responsible for notifying its Presenter with events and displaying data as instructed by the Presenter. No business logic should be located there.

A Presenter is a mediator between a View and the Model. It is notified of events, knows how to react to them, and invokes appropriate methods of the View and Model layers. The Model is responsible for data storage and retrieval. It is often considered to also encompass what would usually be called a service layer - a layer providing functionality such as network communication or device location. A diagram of this architecture is shown in Figure 3.13.

The high-level component diagram of the ViewLink app is shown in Figure 3.14. The components inside the boundary *ViewLink app* belong to the application while the components outside are external - libraries, data storage systems or remote servers.

3.3.3.1 Persistence

The application will need to persist information about the data definitions it is set up with. For that it will leverage the Room library, at a cost of a slight complication. In order to persist objects of a class through the Room library, the class must be annotated using Room annotations and may not contain references to instances of other complex classes. This requirement forces the application to have a separate class hierarchy that mirrors the `DataDef` class

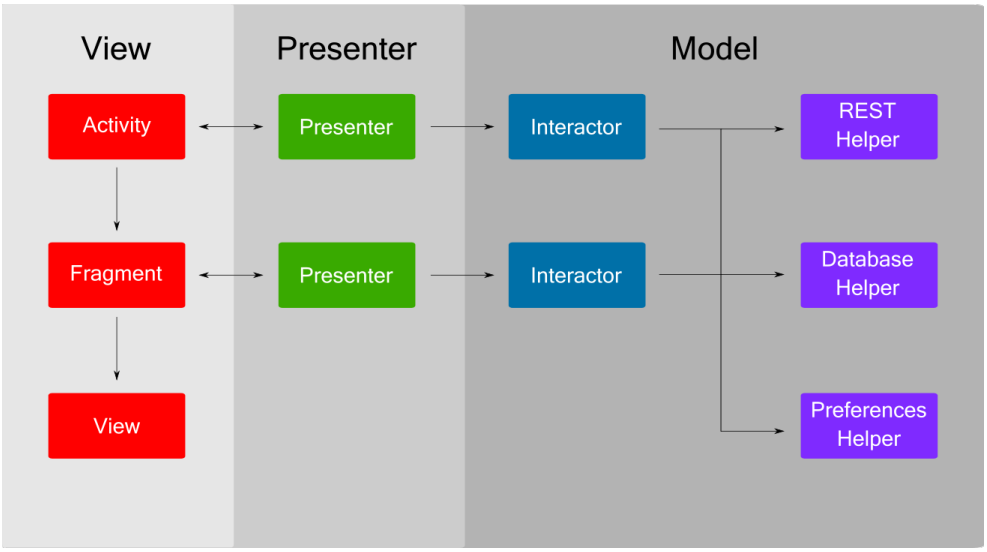

Figure 3.13: The Model-View-Presenter pattern [6]

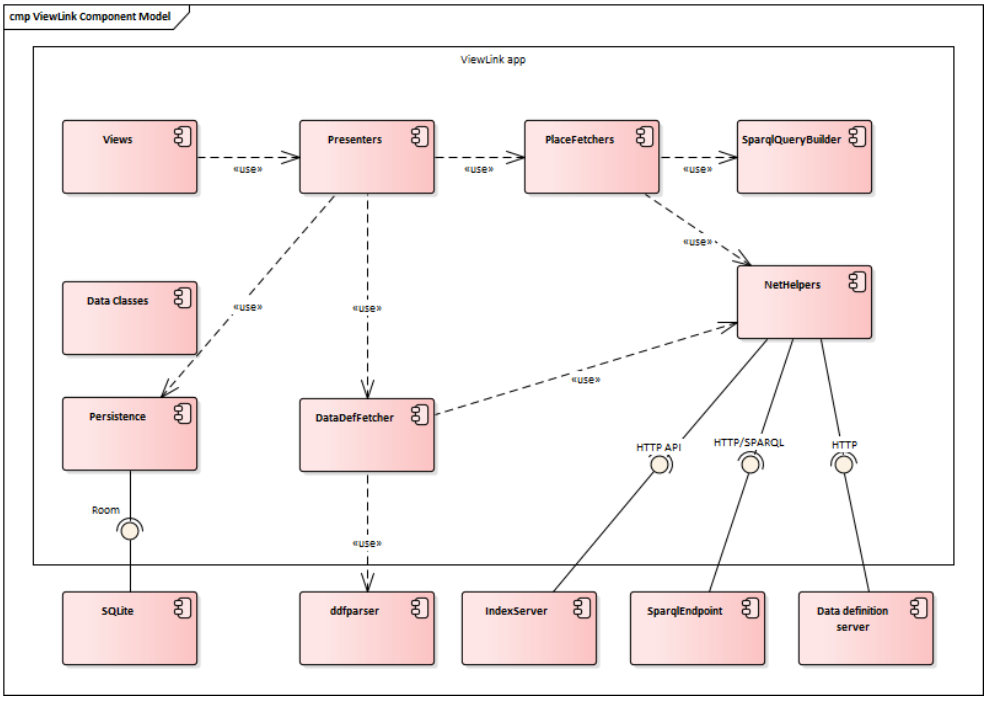

Figure 3.14: ViewLink app Component model

hierarchy of the `ddfparser` library, with slight adjustments made to accommodate for the Room library. The class diagram of this `DataDef` hierarchy is shown in Figure C.2.

3.3.4 Package model

The classes that form the application are divided into several packages based on their concerns. A diagram of the package is shown in Figure 3.15. The responsibility of each package is discussed further in this section.

3.3.4.1 `data` package

The concern of this package is data manipulation. The `NetHelper` interface provides convenience methods for accessing data over the Internet. The `DataDefHelper` interface defines a contract for fetching data definitions converted into internal Java objects. Both of those classes are instantiated via their `Provider` classes, which makes subsequent unit testing easier thanks to being able to provide mock objects instead of real implementations.

The package also contains two subpackages. The first one, `place`, handles fetching data specified in data definitions. The `PlaceFetcher`, given a Data Definition object, fetches Places from a remote source. If an index server is defined, it is used first. If there is no index server defined or fetching from it results in an error, the resolution falls back to the naive solution (section 2.3) and attempts to resolve data from SPARQL an endpoint.

The `persistence` subpackage provides an interface for the rest of the application to the Room persistence library. It defines Room-annotated interfaces that define database queries to be implemented by the library, as well as primitive-to-complex type conversions in the `Converters` class, and the main Room database class `AppDatabase`. The class `ParserDatadefPersistor` is responsible for persisting `DataDef` objects received from the `ddfparser` library. Because the application has its own `DataDef` hierarchy (for reasons discussed in subsubsection 3.3.3.1), this class also acts as an adapter from the library classes to the app classes.

3.3.4.2 `model` package

The `model` package is concerned with the data classes present in the application. The package is divided into three subpackages. The first one, `datadef`, contains classes representing a data definition. There needs to be a separate hierarchy to the one already defined in the `ddfparser` library as discussed in the previous section and in subsubsection 3.3.3.1. The class diagram for classes of this package is shown in the appendix, Figure C.2.

The `place` package defines classes representing the data to be fetched and shown on the map. Each `Place` is represented by one marker on the map and

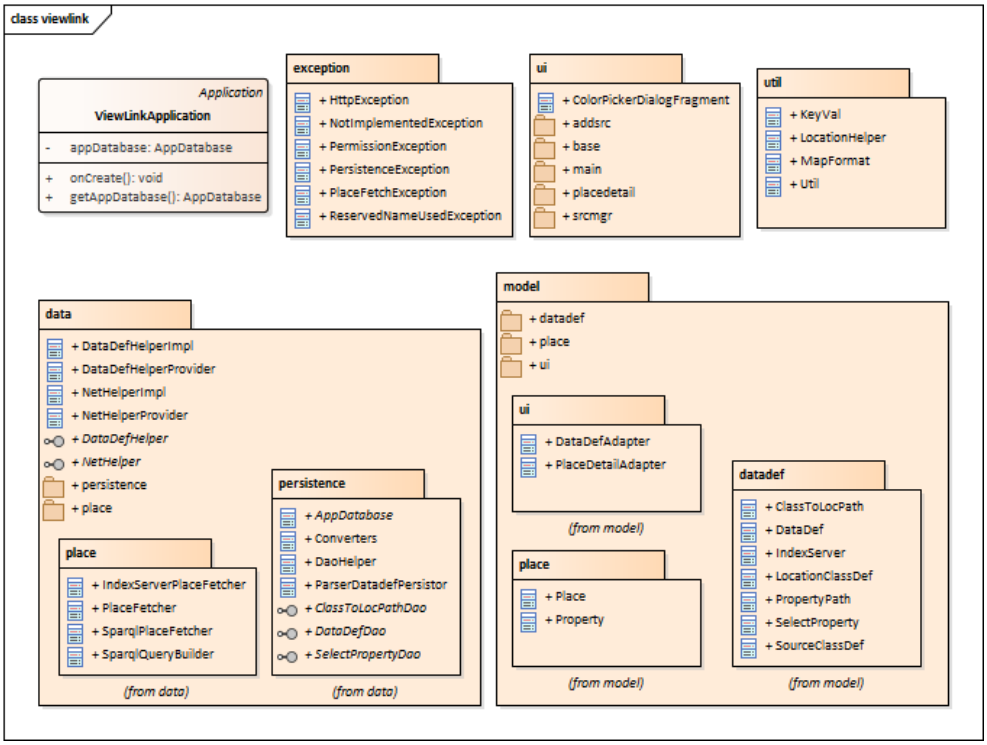

Figure 3.15: ViewLink app package model

refers to a single instance of the Source Class in the Andruian framework. A Place has an optional list of properties, which are simple key-value pairs.

The `ui` package contains classes that tie the model and the UI together. Both classes are subclasses of the `RecyclerView.Adapter` abstract class defined by the Android framework. They are used to show lists of elements on the screen by the *RecyclerView* widget.

3.3.4.3 `util` package

This package contains miscellaneous utility classes, such as a wrapper class for the result of an *AsyncTask* which may be a result or an exception, or a class to simplify the process of asking the user for permission and obtaining their location.

3.3.4.4 `ui` package

This package contains all the classes forming a user interface and handling user interaction. Each of the subpackages corresponds to one screen in the application workflow. Each screen consists of two interfaces - the View and

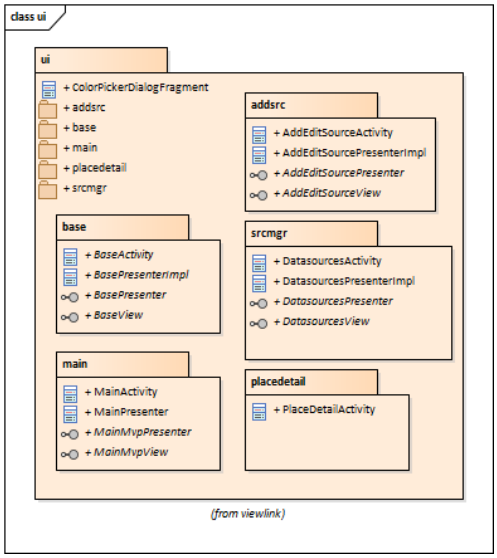

Figure 3.16: Subpackages of the `ui` package

Presenter - and their implementations. A detailed `ui` package diagram is shown in Figure 3.16.

The `base` package defines interfaces and abstract classes implementing those interfaces that are common for all screens. An example of a shared concern is showing messages to the user, keeping and exposing a reference to the Application context and the associated View or Presenter.

Package `main` contains classes forming the main screen of the application showing a map to the user. The classes in `srcmgr` package make up the datasource management screen and classes of `addsrc` package constitute the screen for adding a new data source. The place detail screen is only formed by a single activity, because it does not have any complicated workflow to warrant a definition of View and Presenter interfaces.

3.3.5 Client-side marker clustering

The Andruian framework does not limit the amount of data that can be published and so, theoretically, it may be required to show a marker for every address place in the Czech Republic. The Google Map API does not optimize displaying a large amount of markers in any way and so even though the markers would overlap, they would still all be rendered. This would cause map cluttering, strain the CPU and potentially make the application unusable.

A solution to this problem is marker clustering. When the map is zoomed in in such a way that two or more markers overlap, a cluster marker is created that acts as a placeholder for the place markers in its proximity. The cluster marker is colored the same as the place markers it represents, and differs only

in shape. The cluster marker also shows the order of magnitude of the number of contained markers.

The implementation details of marker clustering as well as relevant screenshots are discussed in subsubsection 4.3.2.1.

3.3.6 Server-side marker clustering

The client-side marker clustering proved to be useful for de-cluttering the map, but in order for the application to handle displaying large quantities of data, a server-side clustering functionality was introduced in subsection 3.2.7. To accommodate for this change in the server, several adjustments must be made in the app design.

On the most abstract level, the architecture does not need to change. The changes are isolated only to some classes. As in the case of the index server, a new `model` class is added to represent a cluster of places. The `IndexServerPlaceFetcher` class responsible for fetching data from an index server must be changed to send queries containing the `clusterLimit` parameter and to accept the new response type - a list of clusters instead of a list of places.

Classes in the `ui.main` package are the last to require a change. The presenter must be able to handle receiving clusters as well as places from the service layer and the view has to be able to show both types of objects on the map.

The visual look of the server-side clusters is identical to the look of client-side clusters. The user should not know whether the application has data about any actual places hidden inside a cluster or not.

Chapter 4

Implementation

This chapter consists of the implementation process of the three components of the system - the parser library, the index server and the Android application. All source code and testing data are provided on the enclosed medium and on the GitHub Andruian page[52].

4.1 Data definition parser library

The source codes of the parser library are publicly available on GitHub[10]. The library can be downloaded via Maven or Gradle from Bintray[11] with the group ID `cz.melkamar.andruian` and the artifact ID `ddfparser`.

The GitHub repository contains detailed instruction about how to use the library. The API is straightforward and contained in a single class. Parsing a text in the form of an `InputStream` is as simple as:

```
InputStream is = null; // Provide your own
List<DataDef> l = new DataDefParser().parse(is, RDFFormat.TURTLE);
```

The library depends on RDF4J, particularly on the artifact `rdf4j-rio-turtle` of group `org.eclipse.rdf4j`. This artifact must be on the Java classpath when using the `ddfparser` library. If using Maven or Gradle, this is done automatically through a transitive dependency.

The `ddfparser` library is capable of parsing RDF formats according to the RDF4J artifacts provided on the classpath. In its basic version, only Turtle is supported. To allow parsing other RDF formats, appropriate artifacts must be added to the project's `build.gradle` file and the library rebuilt.

[10] https://github.com/andruian/datadef-parser
[11] https://bintray.com/andruian/releases

4.1.1 Using parser library on Android

In order to use the `ddfparser` library on Android, the `Xerces2` library must be provided [53]. This can be done either manually or by specifying it as a dependency in the Gradle build system. However, including the Xerces library may pose some stability risks, which are further elaborated in subsection 4.3.4.

4.2 Index server

This section covers the implementation of the index server. The server structure and components have already been discussed in the previous chapter and the implementation is only a realization of it. The design outlined in earlier parts of this thesis proved valid and no major changes had to be made during the implementation. Therefore, this section is brief and only describes topics not covered before.

The source code is commented and publicly available on GitHub[12].

4.2.1 Build system

The Andruian Index Server, or Indexer for short, uses the Gradle build system. It provides a convenient way of automating build-related tasks, such as dependency management, automated testing and application packaging.

The implementation leverages Spring Boot, a project on top of the Spring framework, which takes an opinionated view on Spring configuration. It simplifies development with Spring by offering a set of pre-defined *starter POMs*, which are library and configuration bundles related to a certain implementation aspect, such as web development or security. It also removes the need for tedious XML configuration and provides a sane set of defaults and customization through Java annotations. Furthermore, Spring Boot implements a Gradle task that produces a standalone, production-ready runnable JAR file including all dependencies and an embedded web server. This greatly simplifies both development and deployment of the server. [54]

4.2.2 Web GUI

The web graphical interface is a simple tool for quick visualization of the data indexed on the server. The implementation uses the Spring MVC framework and Thymeleaf templating engine for generating pages.

The GUI constitutes of two screens:

Home screen is a screen with an interactive map and a form allowing users to query and visualize the data. It uses the Google Map JavaScript API to show a map and populate it with markers [55]. The number of markers

[12] https://github.com/andruian/indexer

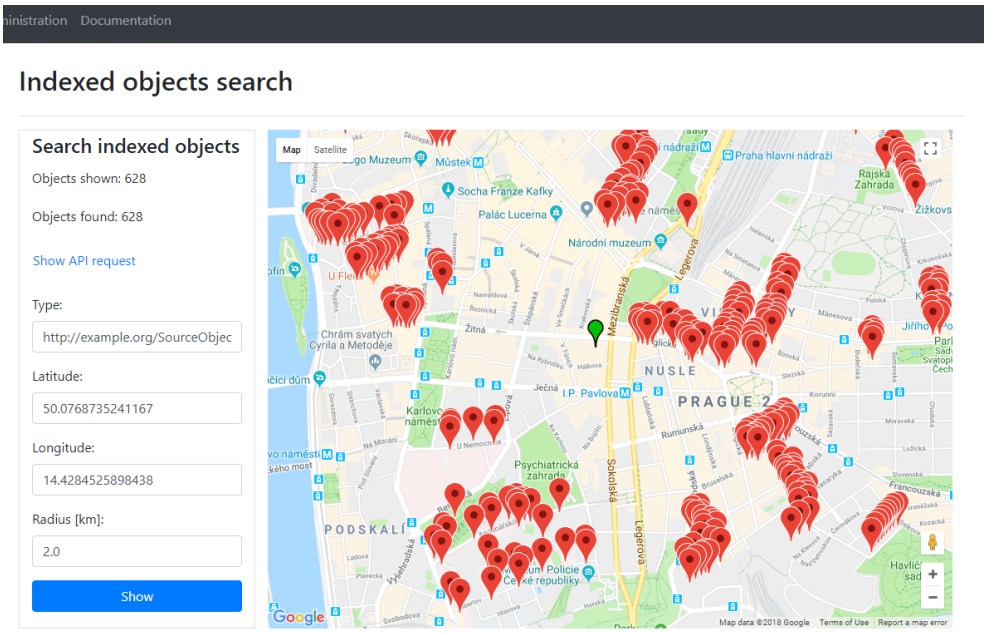

Figure 4.1: Query visualization screen

shown is limited due to performance reasons. The limit is customizable through the `ui.maxPointsShown` configuration key. A picture of this screen is shown in Figure 4.1.

Administration is a screen showing the current configuration of the server. It is only available after logging in as the administrator. It allows the administrator to add a new data definition by providing a URL to a RDF file where one or more data definitions are provided. The administrator can also trigger full or incremental reindexing for a data definition, drop any indexed data or remove a data definition source from the system. Each action is performed on all data definitions located in a specified RDF file. The screen is shown in Figure 4.2.

4.2.3 Reporting of indexing progress

One thing not discussed in the design chapter is the fact that indexing is a task that may take a long time. It must not be implemented as blocking, or the administrator would have to wait for the task to complete before they could interact with the system again.

The server implementation uses asynchronous methods provided by the Spring framework through the `@Async` annotation. When a user triggers reindexing, the asynchronous method is called and it immediately returns a

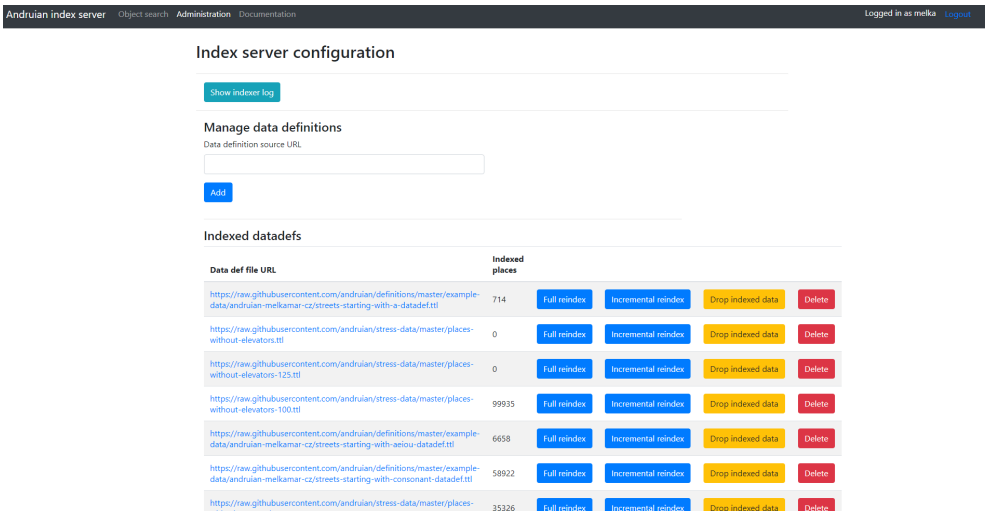

Figure 4.2: Administration screen

`CompletableFuture` object. This object is a wrapper around the result value that may be available in the future. The `Future` is stored in an internal list and the control returned to the user.

Whenever the Administrator page is loaded, the UI controller sends a request to a service to poll all running jobs and retrieve their current status. If the `Future` does not contain any value, the job is still running. If it contains a value, it has already finished and is removed from the running pool. If it contains an exception, something wrong happened during the execution and the indexing failed. The controller then presents this information to the user.

4.2.4 Server-side marker clustering

This section delves deeper into the implementation of processing a faceted heatmap query response from Solr, which is used to cluster markers on the server.

Below is an example of the format of the faceted query response. The heatmap data is identified by the `facet_heatmaps` key. The data is formed by an array of metadata values and a nested array describing the position of clusters and the number of places in them.

The response does not contain the geospatial coordinates of the clusters. Instead, it only provides the bounding rectangle of the heatmap grid region through `minX`, `minY`, `maxX` and `maxY` coordinates and the number of grid cells through `columns` and `rows`. The number of places belonging to each grid cell is represented via a 2D array. The outer index of the array denotes the row and the inner index of the array denotes the column. Whenever the whole

row is empty, a `null` is passed instead to save bandwidth.

In the following example, there are no clusters anywhere except the 4th row. Inside that row, there are 3 places contained in the second cell, 2 places in the fourth cell, 5556 in the 18th cell and 42 in the 27th cell.


```
{
 "responseHeader":{
    ...
 },
 "response":{
   "numFound":107307,
   "start":0,
   "docs":[
      {
        "iri":"http://src.com/adresní-místa%2F21922179",
        "type":"http://example.org/SourceObjectA",
        "location":"50.00653,14.413629",
        "locationObjectIri":"https://example.org/21922179",
        "label":"Amortova 1",
        "srcddf_dynstr":"http://foo/dataDefA",
        "StreetName_prop_dynstr":"Amortova",
        "PSC_prop_dynstr":"14300",
        "StreetNum_prop_dynstr":"1",
        "_version_":1598026549471215616
      },
      ...
   ]
 },
 "facet_counts":{
    "facet_queries":{},
    "facet_fields":{},
    "facet_ranges":{},
    "facet_intervals":{},
    "facet_heatmaps":{
      "location":[
        "gridLevel",2,
        "columns",32,
        "rows",32,
        "minX",-180.0,
        "maxX",180.0,
        "minY",-90.0,
        "maxY",90.0,
        "counts_ints2D",[
          null,
```

```
        null,
        null,
        [
          0,3,0,2,0,0,0,0,0,0,0,0,0,0,0,0,5556,
          0,0,0,0,0,0,0,0,42,0,0,0,0,0
        ],
        ...
        null,
        null]]}}}
```

In order to calculate the position of each cell on a map, the bounding rectangle defined in the response is divided evenly based on the number of rows and columns. That gives us the width and height of each cell. The coordinates of each cluster are then calculated by adding a certain multiple of cell widths or heights to the bounding rectangle's left or top coordinates, respectively. This results in the coordinate of the top left edge of each of the cells. To center, half of the cell's width and height is added to the result.

A special case might occur due to the coordinates "overflowing" when a small correction is required. This has already been discussed in subsection 1.4.1. This way of calculating the position of clusters is not absolutely precise. However, as the user zooms in on the map, the clusters will be re-queried over a smaller area and so the imprecision will not be noticeable.

When processing a query, the server needs to determine how many places would fall into the query result. The number of places is then compared to the `clusterLimit` parameter and the server either responds with a list of places or list of clusters. Processing a clustering query is significantly faster than processing a regular query, especially when covering large ares, such as the whole of Czech Republic. Based on experiments during the implementation, responding to a cluster query is up to 10 times faster than responding to a regular query. For that reason, when a data query is being processed, the server always internally performs a clustering query. That results in a collection of clusters from which the total number of places may be easily calculated. This number is then compared to the `clusterLimit` parameter. If the number of places is smaller than the limit, a regular query is executed and its result returned. Otherwise, the cluster collection is returned without any further processing.

4.3 Android app

The implementation of the Android application, ViewLink, is discussed in this section. Implementation details that were not discussed in the design section are addressed here. Also, a gallery of screenshots is provided and compared to the mockups put forward during the design.

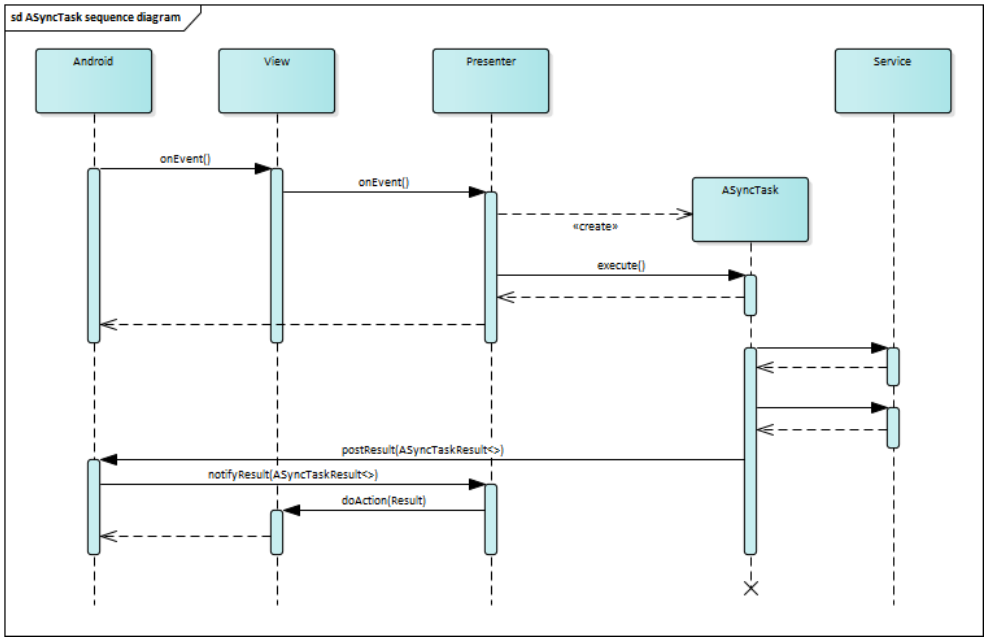

Figure 4.3: ASyncTask sequence diagram

4.3.1 Asynchronous UI

An issue in Android programming that was not raised in the design stage is the inherent asynchronous nature of the system. An Activity is a basic building block of the application and represents a single screen the user interacts with. Navigating to a different screen usually means instructing the Android framework to create a new instance of a different Activity and launch it for the user to see.

When an Activity is being instantiated by the system, various listeners are typically attached to visual elements of the UI. Whenever the user interacts with an element in a particular way, the system calls the listener's predefined method and thus gives control to the application. The application can now react to the event however it sees fit. In the architecture used by the ViewLink app, a method of the Activity's Presenter is usually invoked. The Presenter handles the logic of what that event means and eventually gives control back to the Android system.

The issue is that the scenario above is executed on the application UI thread. The system cannot update the UI or react to events for as long as the thread is being blocked by the application. Blocking the UI thread for a time as short as 16 milliseconds can lead to visual lag that the user may notice. Blocking the thread for more than 5 seconds will cause the system to show an *Application not responding* dialog suggesting to kill the application.

The solution is to make sure that long-lasting tasks are executed asynchronously, off the UI thread. Some functionality is deemed inherently long-lasting and will result in application crashes if run on the UI thread at all - for example, networking. Database access in general may be performed on the UI thread, but Google's Room persistence library, used by the application, requires its methods to be executed off the UI thread as well, else it throws an exception.

This restriction means that most of the application functionality executed outside the View and Presenter needs to be asynchronous. However, the architecture does not differ greatly from the one put forward during design. Any operation that requires network or database access is executed asynchronously from the Presenter layer using an `ASyncTask`. That is a utility class supplied by the Android framework for operations that need to be run in the background, but are not long-lasting (such as a messenger application waiting for events and displaying notifications would be). The Presenter implements callbacks that are invoked on the main thread once the asynchronous execution completes. The control is then returned back to the View and the Android system. An abstract diagram of this process is shown in Figure 4.3. The lifeline objects in the diagram are only abstract entities, not actual objects in the system.

An exception thrown during the asynchronous execution would lead to the app crashing, and catching the exception without giving the user any information would be bad design. To accommodate for that a generic wrapper class was implemented for easier handling of `AsyncTask` results. The class implements two constructors - one for the type of the actual result and one for the `Exception` type. Every consumer of the `ASyncTask` result value is responsible for checking whether this wrapper object contains a correct result, or an exception. If an exception is found, the consumer should display it to the user in a manner depending on the current application context. [56]

4.3.2 Screenshots

This section shows screenshots of the implemented application. Although the screens are based on the mockups realized in section 3.3, there are several differences in the final app.

The drawer menu shown in Figure 4.5 contains one more button than was shown in the mockup - the settings button. Tapping it brings the user to the settings screen where he or she can adjust the application behavior. Currently the only supported option is whether to automatically update markers on the map as the user drags the map view. Enabling it results in a higher data consumption, so some users might prefer to only update markers manually.

The place detail screen is shown in Figure 4.6. Unlike its mockup counterpart, it uses Material elements, such as card views to show a list of items, and a collapsible header toolbar with a main action button.

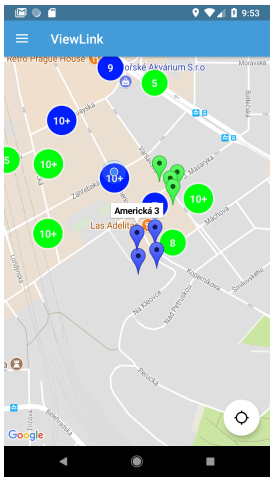

Figure 4.4: The main screen of the ViewLink app.

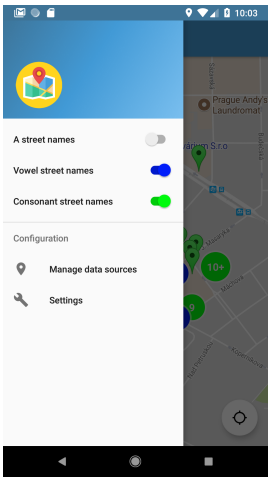

Figure 4.5: The drawer menu of the ViewLink app.

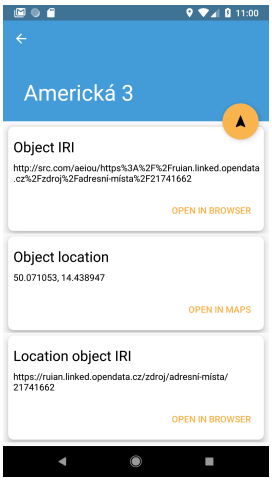

Figure 4.6: The place detail screen of the ViewLink app.

4.3.2.1 Client-side marker clustering

The implementation of client-side map marker clustering uses a utility library provided separately from the Map API, the Marker Clustering Utility [57]. The library provides the `ClusterManager` class, which acts as an adapter between the data source and the map. The application does not populate the map directly, instead it provides data to be shown to a `ClusterManager`, which determines which markers to render directly and which to group together before passing that information to the map. There is one instance of `ClusterManager` for each data source being shown. Multiple `ClusterManager` objects are necessary in order to display the cluster markers in different colors. The marker grouping is shown on the screenshot in Figure 4.4.

4.3.3 Server-side marker clustering

Zooming in on the map requires special handling when displaying server-side clusters. When only client-side clusters are displayed, all the underlying place data is actually available to the application. The marker clustering library is notified when the map is zoomed and performs reclustering, perhaps displaying some places as separate markers. No communication with the data source is needed as long as the user is looking at a portion of a map which has already been queried for.

However, when displaying a server-side markers, it is necessary to perform a query every time the user zooms in. That way, the server may perform clustering over smaller area and provide data with higher detail.

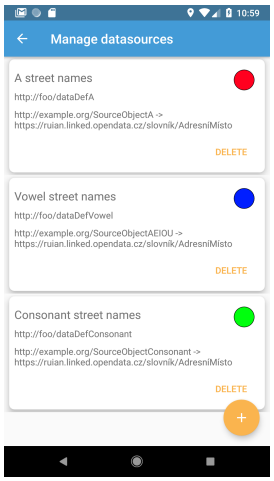

Figure 4.7: The data definition manager screen of the ViewLink app.

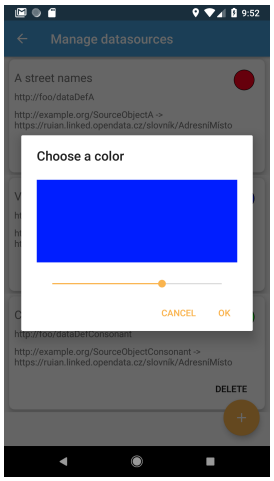

Figure 4.8: The color picker dialog of the ViewLink app.

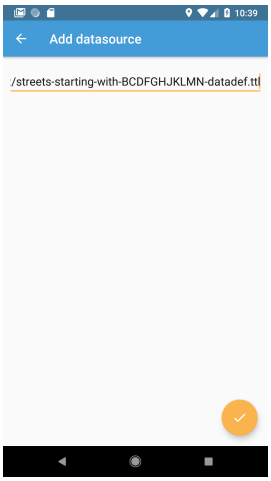

Figure 4.9: The new data definition screen of the ViewLink app.

4.3.4 Using ddfparser library

The RDF4J component used by the `ddfparser` library does not work on Android out-of-the-box. Including and using the library with no further dependencies results in the following exception:

```
Caused by: javax.xml.datatype.DatatypeConfigurationException:
  Provider org.apache.xerces.jaxp.datatype.DatatypeFactoryImpl
                                                    not found
  at javax.xml.datatype.DatatypeFactory.newInstance()
  at org.eclipse.rdf4j.model.impl.AbstractValueFactory.<clinit>
  at org.eclipse.rdf4j.model.impl.SimpleValueFactory.getInstance
  at org.eclipse.rdf4j.rio.RDFFormat.<clinit>
```

RDF4J expects the `org.apache.xerces` library to be present in the classpath. This is the case with JVMs distributed with full-fledged JRE and JDKs[58], but not the case with ART. Besides being embedded with Java releases, Xerces is also distributed as a Maven artifact[53]. Including this artifact in `build.gradle` of an Android project fixes the issue.

However, another problem arises when the application is built in the *release* variant. The build process terminates with an error stating that there is an *"Ill-advised or mistaken usage of a core class (java.* or javax.*) when not building a core library."*. The full error message, very explicit in its warning,

can be found for example on StackOverflow[13]. This is because by default the release build process checks whether a class is being implemented in the core Java package namespace, as it may potentially cause clashes with core libraries in future Android versions.

The suggested workaround is to repackage classes so that they are not located in the core `java` package. That is not a viable solution in this case, however. Repackaged versions of Xerces already exist to allow a smooth usage on Android[59]. But RDF4J uses the `ServiceProvider` mechanism and looks for a provider of a particular name, which is supplied by the Xerces library[60]. Renaming the Xerces classes causes this lookup to fail with the error whose stacktrace is shown above.

A possible solution could be rebuilding the RDF4J library with a changed name of the dependency, but that would mean maintaining a separate branch for Android compatibility. I did not pursue this approach further. Instead I acknowledged the possible risks and instructed the build process to ignore its warning by using the `--core-library` flag.

[13] `https://stackoverflow.com/questions/18266853/attempted-to-fix-androids-ill-advised-or-mistaken-usage-of-a-core-class-went-b`

Chapter 5

Testing

This chapter first covers the automated testing process of all the implemented components. Afterwards, it lays out user scenarios according to which the components have been tested. The scenarios are designed so that they can be followed along by the reader. Finally, the results of a stress test are presented, where the performance of the implemented framework is assessed.

Whenever it is mentioned that sources are available at GitHub, they are also available in the enclosed medium.

5.1 Automated testing

This section briefly explains the automated testing process of the data definition parser library, the Android application and the index server.

5.1.1 Data definition parser library

There are automated unit tests for each parsing function of the library. The tests execute the parsing methods of the library against a set of data definition files. The parsed objects are then run through a series of checks to verify everything was parsed as expected.

With every push into the GitHub repository a build is automatically triggered on Travis CI[61] to verify whether the automated tests succeed or not. The build results are published back to GitHub via a status badge icon.

The tests are available on GitHub in the `ddfparser` repository[62].

5.1.2 Android app

This section discusses the testing of the Android application ViewLink. The same approach has been applied to it as with the Index server - some functionality has been tested automatically, and the complete flow has been tested manually.

The core functionality of the app has been tested automatically using the Espresso UI testing framework. The tests include the manipulation of data definitions and assertions that map markers are being shown as expected.

The tests are available on GitHub in the `viewlink` repository[63].

5.1.3 Index server

Automated tests using JUnit4, Mockito and SprintBootTest were written to check the indexing and HTTP API functionality. Those tests are run with every push to the project GitHub repository.

The tests are available on GitHub in the `indexer` repository[41].

5.2 User scenarios

This section lists user scenarios for testing the Android application and index server. Scenarios for both components begin with the simplest ones and move on to the more complicated ones.

5.2.1 Android app

The application has been tested manually using the following scenarios to ensure all use cases identified during the analysis in section 2.6 are covered. The scenarios are described in the following sections. They start with installing the application and then explore the functionality the application has to offer.

The scenarios in this section are intended to be executed on a real mobile device. In order to make typing links easier, they will be shortened using the Google URL Shortener service[64]. The scenarios are "incremental" in the sense that a latter scenario may require completing a prior one.

5.2.1.1 Installing and opening the application

The first step of testing the application is installing it. The easiest way to install the application is through Google Play. The application may be found either by searching for `viewlink` or on url `https://goo.gl/GsjZdN`.

After the application has finished installing, open it by tapping the `ViewLink` icon in the application drawer.

5.2.1.2 Adding a data source

1. Open the drawer menu by tapping the top left corner or by dragging the left edge of the screen inwards.

2. Tap the *Manage data sources* button to open the *Manage datasources* screen.

3. Tap the plus button on the bottom right side of the screen to open the *Add datasource* screen.
4. Enter URL containing a data definition. To use an example data definition, enter the following URL: `goo.gl/dDxRSd`. It points to several thousands of address places in Prague whose streets begin with a consonant.
5. Confirm your selection and wait for the data definition to be parsed.
6. The *Manage datasources* screen should be shown. Verify that a new entry with the name *Consonant street names* has been added to the data definition list. The entry further contains the IRI of the data definition (`http://foo/dataDefConsonant`) and indicates which source class is being mapped to which location class.
7. Optionally change the marker color of the data definition by clicking the colored circle.

5.2.1.3 Displaying data on the map

1. Make sure that at least one data definition is set up in the application. If not, refer to scenario in subsubsection 5.2.1.2.
2. Open the drawer menu by tapping the top left corner or by dragging the left edge of the screen inwards.
3. Enable any number of data definitions by tapping the switch buttons next to their names. The switch button reflects the color of the markers.
4. Close the drawer menu. Tap the *focus location* button on the bottom right side of the screen to center the map on the device location, or manually drag the map view.
5. A loading indicator should be shown in the top right corner of the map while places are being fetched. When that is finished, markers will be shown on the map.
6. Zoom in the map to view individual markers.
7. Zoom out the map to make the markers cluster together.

5.2.1.4 Changing the auto-refreshing behavior

By default the application will refresh markers shown after the map is moved by the user. This behavior may be disabled.

1. Drag the map around and verify that markers are being automatically refreshed.

2. Open the drawer menu by tapping the top left corner or by dragging the left edge of the screen inwards. Tap the *Settings* button.

3. A Settings screen will open. Disable the *Update map markers automatically* switch.

4. Navigate back to the map view by tapping the back button.

5. Drag the map around and verify that instead of automatically refreshing markers, a button on the top of the screen is shown.

6. Tap the *Show places in this area* button. Verify that markers are refreshed for the current map view.

7. Revert the setting to the original (enabled) state.

5.2.1.5 Displaying a large amount of places on the map

When a large amount of data is to be shown, it is pre-clustered on the server side before being sent to the client application. This scenario uses an existing data definition that contains hundreds of thousands of places to test that case.

1. Add a new data definition to the app by entering this URL: `goo.gl/jSFwfJ`. For detailed steps refer to the scenario in subsubsection 5.2.1.2.

2. After the data definition finishes parsing and is shown in the *Manage datasources* screen, navigate back to the map.

3. Center the map on Prague.

4. Zoom out the map. At a certain level of zoom the clusters shown should form a uniform grid. The grid is calculated by the index server instead of relying on the mobile device.

5.2.1.6 Displaying detailed information about a place

1. Make sure that at least one data definition is set up in the application and some of its markers are shown on the map.

2. Position the map so that a single marker is visible. If only clusters are visible, zoom in on one of them until a single marker appears.

3. Tap the marker to show its label.

4. Tap the marker label to open up the *Place detail* screen.

5. Verify that the *Place detail* screen shows information about the place. The information shown is defined in the associated data definition.

6. Tap the navigation button to start a Google Maps navigation to the place. Afterwards, press the back button to navigate back to the app.

7. Verify that some of the property cards contain buttons. A button is shown when the content of the corresponding card is understood by the application. Currently the application recognizes URLs and GPS coordinates.

5.2.1.7 Displaying data without an index server

So far, all the scenarios used data definitions that relied on data provided by an index server. In this scenario, we will test how the app behaves when no such server is available.

1. Add a new data definition to the app by entering this URL: `goo.gl/fzgPPi`. For detailed steps refer to the scenario in subsubsection 5.2.1.2.

2. Verify that the *Manage datasources* screen contains a data definition named *A street names, no index*.

3. Navigate back to the map.

4. The map should show the spinning bar indicating places are being fetched. This may take over 10 seconds.

5. Verify that after the places have been fetched, the markers behave identically as if they were fetched from an index server. That includes clustering and tapping individual markers.

5.2.2 Index server

The web interface of the index server has been tested manually according to the scenarios listed below. An instance of the index server is set up for demonstration purposes and publicly available here: `http://andruian.melkamar.cz/`. The administrator username is `melka` and the password `1234`.

5.2.2.1 Displaying data on a map

The index server provides a simple interface for displaying the data currently indexed. It is intended to only be used for debugging purposes and so does not have the feature set of the Android application.

1. Open the URL `http://andruian.melkamar.cz` in a web browser.

2. Click on Prague. Verify that a marker has been placed on the map and the form fields *Latitude* and *Longitude* have been populated with the coordinates of the created marker.

3. Click the *Show* button. The page should reload and markers should be shown around the place you selected.
4. Check the *Cluster markers* checkbox and enter "100" to the *Radius* text field.
5. Click the *Show button*. The page should reload and display markers that are evenly spaced, forming an incomplete grid.

5.2.2.2 Removing and adding a data definition to be indexed

1. Open the administration tab of the index server at URL `http://andruian.melkamar.cz/admin`.
2. If a login screen is shown, enter *melka* as the username and *1234* as the password and confirm.
3. If the data definition list contains URL `https://goo.gl/9oxPha` or a long URL ending with *streets-starting-with-a-datadef.ttl*, click the *Drop indexed data* button next to one of them and then delete them from the server using the *Delete* button.
4. Add a data definition source URL `https://goo.gl/9oxPha` to the *Manage data definitions* text input and click the *Add* button.
5. The page should refresh and a message box at the top of the page will inform you that a data definition has been added. Depending on the server configuration, it may automatically start indexing, which would be written in another message. If the indexing did not start, trigger it manually using the *Full reindex* button.
6. The indexing should take about 30 seconds. Refreshing the page continuously will show the indexing progress. More thorough time results are discussed in section section 5.3.
7. After the indexing finishes, verify that the server has indexed 714 places for this data definition.

5.2.2.3 Adding new data to a triple store

This scenario explains how to upload new data to a Fuseki triple store.

An example dataset is already created and available in the `andruian/example-data` GitHub repository[14]. The data was created by running a SPARQL CONSTRUCT query against the RÚIAN SPARQL endpoint. The query is listed in the appendix, section B.5. The schema of the resulting data is shown in Figure 5.1.

[14] `https://github.com/andruian/andruian/tree/master/example-data/datasets/incremental`

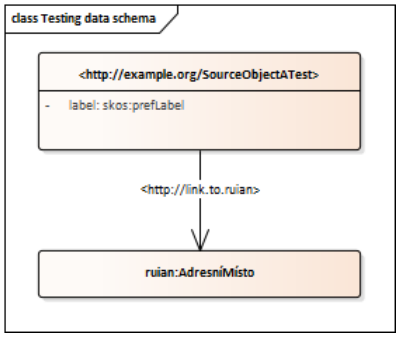

Figure 5.1: Schema of the testing dataset

1. Open the Fuseki web interface at URL `http://fuseki.andruian.melkamar.cz/`. Log in using username `admin` and password `some-secret`.

2. Navigate to *manage datasets* and open the *add new dataset* tab. Choose any name for your new dataset and click *create dataset*.

3. In the list of datasets available on the server, click the *upload data* button next to your newly created dataset. In the next page click the *select files...* button and choose the *streets-starting-with-a-1.ttl* file you can either download from GitHub[15] or find on the enclosed medium. Click the *upload* button.

5.2.2.4 Creating a new data definition

In this scenario, we create a new data definition file and use it to index the data added in the scenario from subsubsection 5.2.2.3. The final data set can be found in the appendix, section B.6.

1. Use an existing data definition as a starting point. A suitable one is provided in the appendix, section B.3, which is used in this scenario.

2. Give a unique IRI to the data definition. Change the *:dataDefVowel* to any IRI. It does not have to actually be dereferencable. It is also recommended to change the *skos:prefLabel* describing the data definition.

3. Change the details of *:sourceClassDef*:

   a) Change the *andr:sparqlEndpoint* to the endpoint you have created during scenario in subsubsection 5.2.2.3. The URL of the endpoint may be determined from the query tab, as shown in Figure 5.3.

[15] `https://github.com/andruian/andruian/blob/master/example-data/datasets/incremental/streets-starting-with-a-1.ttl`

b) Change the *andr:class* to the IRI of the source class, *http://example.org/SourceObjectATest*. The data structure is shown in Figure 5.1.

c) Change the *andr:pathToLocationClass* to reflect the path from the source class to the location class, which is also shown in Figure 5.1.

d) Keep a single *andr:selectProperty* entry, describing the sole property of the source class, *skos:prefLabel*.

4. The location definition will remain the same, because the new data is linking to *AdresníMísto* classes in RÚIAN.

5. Publish your data definition so that it is accessible via HTTP/S. A good tool for testing is Pastebin[65], where you can paste the created data definition. After publishing the definition there click the *RAW* button to obtain a direct link to the plain text.

5.2.2.5 Incremental indexing of new data

This scenario illustrates incremental indexing of the new data which was uploaded to the Fuseki server during scenario 5.2.2.3. The data definition corresponding to this data was created during scenario 5.2.2.4.

The scenario consists of three parts. Firstly, we remove any traces of possible previous testing with the new data. Then we index the data currently loaded in the triple store. Finally, we upload more data to the triple store and perform an incremental reindex.

1. Open the administration tab of the index server at URL `http://andruian.melkamar.cz/admin`.

2. If a login screen is shown, enter *melka* as the username and *1234* as the password and confirm.

3. If the data definition list contains any data definitions with 383 or 714 indexed places, first drop their data and then remove them all from the system.

4. Add the URL of the data definition published in scenario 5.2.2.4 and trigger a full reindex.

5. After the indexing ends, verify that 383 places have been indexed.

6. Following scenario 5.2.2.3, upload file *streets-starting-with-a-2.ttl* into the triplestore into the same dataset as previously.

7. In the indexer web admin interface, trigger an incremental reindex for your dataset.

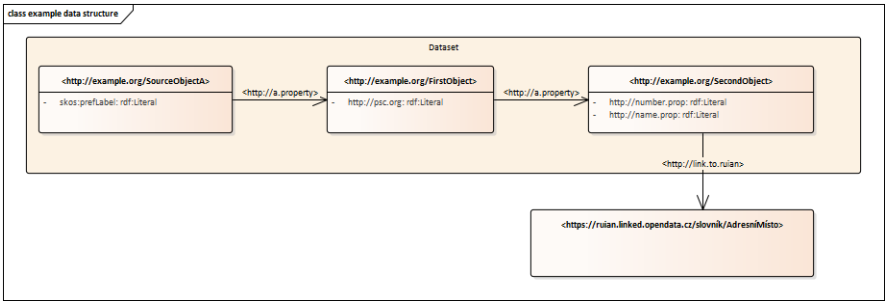

Figure 5.2: The structure of testing data

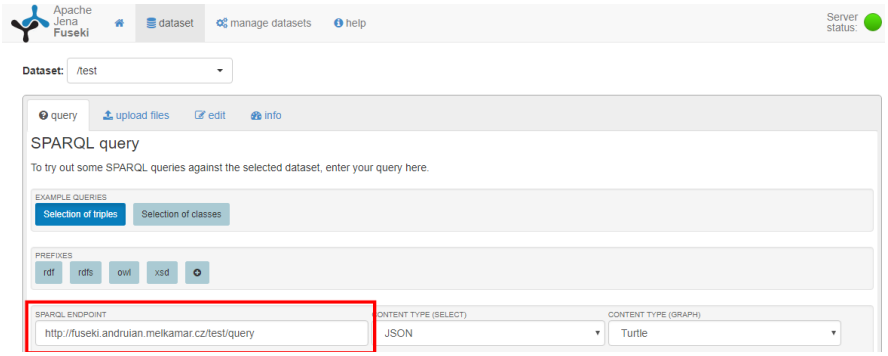

Figure 5.3: Determining a SPARQL endpoint in Fuseki UI

8. After the reindexing has ended, verify that 714 places are now indexed. You may view the incremental query that was executed by clicking the *Show indexer log* button and browsing through the application log.

5.2.3 Further testing data

More testing datasets are available in the *andruian* GitHub repository[16]. They were created using a `SPARQL CONSTRUCT` query based on the template `construct-query.sparql` which is located in the same folder as the data definitions. The structure of the constructed data is shown in Figure 5.2. The structure is similar, but richer than the structure of the data used in scenario 5.2.2.4. The data covers address places in Prague, Czech Republic, and is divided into several datasets based on the first letter of the street the address place is on.

[16]https://github.com/andruian/andruian/tree/master/example-data/datasets

5.3 Stress test

In order to assess the performance and the limits of the index server and framework, a stress test was performed. The data for the test was constructed from RÚIAN using the LinkedPipes ETL tool[66]. The server was a rented VPS with a single-core 1.7GHz CPU and 4GB RAM. Index server had heap space capped at 1.5GB, Fuseki at 1.2GB.

The testing Android device was a Nexus 5X.

The structure of the data was similar to the testing data used and described in subsection 5.2.2. The difference is that it was not limited to Prague, but covered the whole of the Czech Republic. Each building (*ruian:StavebníObjekt*) in RÚIAN has information about whether it features an elevator. A building can either have it, not have it, or be unspecified [67].

Three datasets were created from these three categories and are available on the enclosed medium and GitHub[17]. There are about 360.000 triples in the dataset containing places with an elevator, about 460.000 triples in the dataset containing places where an elevator is undefined and about 3.950.000 triples in the dataset containing places without an elevator. Note that due to the structure of the testing data, 10 triples are generated from a single place in RÚIAN.

5.3.1 Index server

It took the index server about 7 minutes to index the places with an elevator. However, the first 6 minutes were spent waiting for a reply to the indexing query from the source triplestore. The time the index server needed to index the data once it received them was less than a minute. The number of places where an elevator was undefined was similar to the number of places with an elevator, so I did not measure the performance for that dataset and moved straight to the biggest one.

When attempting to index the full 3.950.000-triples dataset, Fuseki ran out of JVM heap space after processing the indexing query for about 85 minutes. Consequently we tried reducing the number of triples to about 2.000.000. Fuseki answered the query after about 63 minutes, but this time the Index server ran out of heap space while processing the response. When the input dataset was reduced to about 1.500.000 triples, the index server was able to successfully process it in 49 minutes. The index consisted of about 150.000 places, each place corresponding to 10 triples in the original dataset.

The stress test shows that the indexing server is sensitive to the amount of memory available. 4GB was not sufficient for MongoDB, Solr, Fuseki, Indexer and Nginx. However, 4GB of RAM is arguably a small amount and most servers will have more of it. One possible way to decrease the memory footprint is to stop using MongoDB for storing the server configuration and

[17] https://github.com/andruian/stress-data

Triples	Places	Indexing time
2.000.000	200.000	62 min
3.000.000	300.000	89 min
3.950.000	395.000	118 min

Table 5.1: Indexing times with dual-core 2.8 GHz CPU and 12GB of RAM

save it locally or in Solr, where an appropriate schema would have to be created. Another improvement is using a scrollable cursor for indexing in chunks instead of requiring all the data to be loaded into memory at once. This is outlined further in 6.3.

In order to verify that having more operating memory at our disposal will allow the index server to process larger datasets, the indexer stack was deployed in a virtual machine on a more powerful computer. It was not accessible from the Internet, however. The computer had 16GB RAM and dual-core Intel Core i7 processor running at 2.8GHz. The virtual machine could utilize 12GB of RAM. The maximum heap space for both the index server and Jena Fuseki was set to 4GB. With this setup, indexing larger datasets was successful. Table 5.1 shows the indexing times for various sizes of the dataset.

5.3.2 Android application

Initially, the server did not support server-side clustering. Without that feature, displaying data in the area of the whole Czech Republic on the Android client was not successful. The index server was able to respond to the query in about 7 seconds, but the Android application was not able to show the data, ran out of memory and crashed. The reason is the fact that all the data contained in the server response is converted into POJOs and passed to the map cluster manager to display. The cluster manager internalizes all the objects given, regardless of how many will be shown. If one was to zoom in from the country level to the street level without re-querying the data, all the places would still be shown, because all of them are kept in memory.

The failure to display large areas prompted a slight change to the design and implementation which allowed the server to generate clusters of places without sending them all to the client. That change is discussed in several sections throughout chapter 3 and chapter 4. After the change the Android application was able to handle all the testing data served from the index server with no issues.

5.3.3 Data loading times

This section summarizes the results of the stress test. It provides an overview of the dataset sizes that may be used without an index server and sizes that the testing server was and was not able to handle.

Triples	Places	Loading time
100	10	0.889 s
1000	100	1.678 s
3000	300	3.647 s
7000	700	7.652 s
15000	1500	15.524 s
50000	5000	Timeout after 30 s

Table 5.2: Android app data loading times without using index server

5.3.3.1 Querying data without an index server

This section presents the experimental results of displaying data of various sizes in the Android application without using an index server. The experiment was performed using the same hardware described earlier in section 5.3.

The data structure is identical to the one described earlier. The data was created by running a SPARQL CONSTRUCT query on the RÚIAN endpoint. For every RÚIAN object matched, 10 triples were created. The queries contained a *LIMIT* clause, limiting the number of results returned to the query. The IRI of the resources in a dataset were unique based on the dataset size. In other words, if a RÚIAN object appeared in two different datasets, the resulting IRI of resources created from it would be different.

Table 5.2 shows the loading times for various sizes of the dataset. Triples from datasets of all sizes were loaded into a single Jena Fuseki endpoint. The results show that the framework is comfortably usable without an index server for small datasets, up to about 100 places. Depending on the user, datasets of sizes 300 or even 700 could still be used without an index server, but the loading time will be very noticeable. Judging from the linear time increase, the application would theoretically be able to handle datasets containing up to 3000 places with the current timeout setting (30 seconds). However, a loading time of 30 seconds is too high for a comfortable usage.

5.3.3.2 Querying data with an index server

Most of the time required to execute a query and display data in the client application is taken by the index server. As mentioned above, a precise lookup of places is only executed after a quick clustering query determines that there are not enough places to warrant being clustered. That means that the worst-case scenario time-wise is displaying non-clustered places, as both clustering and non-clustering query needs to be executed. However, the duration of the non-clustering query is limited by the fact that it is always executed over a predetermined maximum number of elements (otherwise they would be clustered).

A good threshold value for the number of places to show before they should be clustered was experimentally determined to be 1000. Using this value, the loading time was never higher than 2 seconds and was in most cases lower than 1 second. Experiments were also made with an Android emulator querying an index server deployed on a more powerful local machine, as described at the end of subsection 5.3.1. The loading times stayed the same even when using the largest testing dataset containing 395.000 places (3.950.000 triples).

To conclude, the Android application was able to display testing data of all sizes seamlessly.

Chapter 6

Deployment

This section provides a detailed set of instructions about how to generate, deploy or install various elements of the Andruian framework. Firstly, instructions are provided about how to create a data definition describing the structure of data linked to a location data source. Secondly, the deployment process of the index server is described. Lastly, the installation of the Android application is explained.

6.1 Integrating linked data to the Andruian framework

In order to publish a dataset linked to RÚIAN or any other location source using the Andruian framework, two steps are mandatory:

- Publish your linked data so that it is accessible via SPARQL over HTTP.
- Create a data definition file using the RDF vocabulary discussed in section 2.5. An example of such a file is shown in the appendix of this thesis, section B.3, in the enclosed medium and on the project's GitHub[18]. Publish this file somewhere publicly accessible. For a more detailed guide, refer to subsubsection 5.2.2.4.
- Optionally set up the index server and set it up to index the data definition created above.
- Set up client applications to use the URL of the published data definition file.

[18] https://github.com/andruian/definitions/tree/master/example-data

6.2 Index server software stack

This section describes how to deploy the index server and the services it depends on - Apache Solr, MongoDB and a triple store - Jena Fuseki in particular. First each component is described separately, and then a guide for *Docker Compose* is provided, simplifying the deployment of the whole stack to a simple command.

6.2.1 Solr

The index server requires a Solr instance which has a core set up using a particular configuration set. The configuration is provided on the medium supplied with the thesis and is available from the Indexer GitHub repository[41].

First, download the Solr distribution from the project download page[19]. Unpack the archive and run the following commands in the unpacked folder, supplying your own path to the Solr configuration:

```
$ bin/solr start
$ bin/solr create_core -c andruian -d "/path/to/andruian_configset"
```

Then, in the `application.properties`, supply the index server with settings *db.solr.url* and *db.solr.collection* pointing to the url of the server and the name of the core created (the `-c name` argument).

6.2.2 MongoDB

MongoDB does not need any configuration. Simply download and run the database[20], or use the cloud solution, and provide the configuration key *spring.data.mongodb.uri* in `application.properties`.

To start MongoDB, first create a folder to store the database in and then start the database with the `--dbpath` parameter:

```
$ mkdir /data
$ bin/mongod --dbpath /data
```

6.2.3 Apache Fuseki

Apache Fuseki is used to store and server RDF data and its setup is extremely simple. Download the Fuseki archive from the Jena download page[21], unpack it, and start the server using the following command:

```
$ ./fuseki-server
```

[19] http://lucene.apache.org/solr/downloads.html
[20] https://www.mongodb.com/download-center
[21] https://jena.apache.org/download/

6.2.4 Index server

The index server may be built from source or obtained as a release JAR. The sources are located in the Indexer GitHub repository[41] and on the enclosed medium. The release JARs are also provided on the medium and published as Releases in the same GitHub repository as the.

The deployment of the index server itself may be done in two ways - using an external servlet container to deploy a WAR file, such as Tomcat or Jetty, or by running the self-contained JAR file. Both deployments are functionally equivalent, but unless the reader already has a servlet container deployed, it is easier to run the JAR file. WAR files are not provided and must be built from source.

To run the JAR file, invoke this command, substituting the JAR filename for whichever version you are running. The `java` executable must be in `PATH`.

```
$ java -jar indexer-1.0.0.jar
```

The default configuration of the server expects Solr and Mongo to be accessible on `localhost` on their default ports, `8983` and `27017`, respectively. To change the server configuration, place the `application.properties` file in the same folder as the JAR. The configuration is described in detail in subsection 3.2.6.

The index server was tested with Java `1.8.0_131`.

6.2.5 Docker and Docker Compose

The deployment can be greatly simplified using Docker and Docker Compose. A `docker-compose.yml` file is available on the enclosed medium and in the Indexer GitHub repository[41] in the `docker` subfolder. The *Docker Compose* setup uses unchanged Fuseki[68] and Mongo[69] images, a slightly customized Solr image [70] and a custom-made image for the Indexer server based on the OpenJDK 8 image[71].

The customized images are automatically rebuilt anytime a new tag with format `x.y.z` is pushed into the Indexer GitHub repository. All of `x, y, z` must be integers. The tag of the image corresponds to the git tag. The `latest` Docker tag is updated with the newest tag automatically. The automatic build is facilitated by the Docker Cloud service[72].

The `docker-compose.yml` file is crafted to be plug-and-play. Simply run `docker-compose -d up` while in a directory with the file and the whole stack comes up. If required, the components may be customized by mounting configuration files for them before starting up. For more information refer to the `docker-compose.yml` and the image homepages.

6.2.5.1 Ports

The ports exposed by the Docker containers where the services are accessible are the following:

3030 : Jena Fuseki

8080 : Andruian Indexer

8983 : Apache Solr

27017 : MongoDB

6.2.6 Testing instances

As mentioned in the Testing chapter, a Andruian Index server stack is running and available for testing. Below is a list of services and URLs they are available at:

`http://fuseki.andruian.melkamar.cz` : Jena Fuseki. Credentials `admin / some-secret`.

`http://indexer.andruian.melkamar.cz` : Andruian Indexer. Administrator credentials `melka / 1234`.

`http://solr.andruian.melkamar.cz` : Apache Solr.

`http://andruian.melkamar.cz:27017` : MongoDB.

6.3 Android app

The ViewLink application can be installed in two ways. One way is manually delivering the signed release APK file to the target device and opening it. The APK is provided on the enclosed medium and on the Releases tab of the ViewLink GitHub repository[63]. When installing this way, it is necessary to allow the installation of applications from untrusted sources.

The second, preferred way of installation is through the Google Play Store [73]. The application will also be automatically updated this way. To avoid having to manually type the link into the device, the app may be found under *andruian viewlink* in the Store search.

Conclusions and future work

The aim of this thesis was the design and creation of a framework allowing the visualization of RDF data linked to a source of geospatial data on Android devices. The framework would define a mechanism to describe the structure of the data and links in it, which would allow clients to efficiently visualize the data based on spatial queries. An index server was implemented to speed up spatial searches of the linked data. The Czech Registry of Territorial Identification, Addresses and Real Estate (RÚIAN) was used as an example source of spatial data throughout the thesis. An Android application capable of visualizing the data on a map was created for the purpose of demonstration the functionality of the framework.

At the beginning of the thesis, in the first chapter, the technology of Linked Data and RDF was introduced, including one of its serialization formats, Turtle. Next, existing solutions that are similar to the functionality of the proposed framework were presented. They were briefly described and it was pointed out why they are not sufficient and why the need for this framework exists. Afterwards, libraries for RDF and Linked Data manipulation for Android were researched, as well as existing techniques for spatial querying using SPARQL. Finally, usability standards for Android applications were introduced and the geolocation API described.

In the second chapter, the RÚIAN registry was analyzed. The data model of the registry was discussed, as well as its linked data publication structure. Afterwards, the requirements on the framework were outlined, including the reasons for the need of such a framework and the analysis of its architecture. Two possible approaches for querying the data linked using the framework were suggested - a naive one where the client communicates directly with location-unaware SPARQL endpoints, and one which utilizes a separate indexing server. The solution using an indexing server was analyzed and its requirements, use cases and domain model were created. Next, the data definition vocabulary was defined. The vocabulary is used to provide metadata about the data being linked and is a core part of the proposed framework. Fi-

nally, the requirements and use cases of a prototype Android application were analyzed. The application demonstrates the functionality of the framework by visualizing data on a map.

The third chapter was concerned with the design of the three components outlined in the previous chapter. The first component was a standalone Java library, `ddfparser`, capable of parsing data definitions provided by data publishers into Java objects. The library was used by each of the remaining components. Next, the design of the index server was discussed. It included the design of the indexing process and the choice of a storage technology supporting spatial querying. Then the component architecture of the server was created, as well as the more detailed package and class architecture and the HTTP API format used to communicate with the server. It was also discussed why it is necessary for the index server to support server-side marker clustering and the appropriate change to the HTTP API was proposed. The Android application design was crafted in the rest of the chapter. The design comprised of choosing the frameworks to work with, defining a color scheme, creating a logical UI flow and producing screen mockups. The chapter concluded with the design of the component and package architecture.

The fourth chapter covered the implementation of the three components designed earlier. It pointed out some of the problems encountered during the framework realization and elaborated on the implementation details of server-side marker clustering. Images of screens of the index server and Android application were shown and compared to the mockups created before.

The content of the fifth chapter was the testing of the implemented server and application. A brief note was made about automated tests. Afterwards, several user scenarios were laid out according to which both the index server and the Android application could be manually tested. The steps of the scenarios were described in detail, illustrating all pieces of the framework's functionality. Finally, a stress test was conducted to assess the performance of the system and find its limits. The limiting factor for the index server was the size of operating memory available to it, but despite not being able to process large datasets on the limited testing machine, it was able to perform well for reasonable amounts of data. Response time was measured for querying data without using an index server and the maximum size of a dataset which is still able to be processed this was discussed. By virtue of the server-side clustering functionality, the client application was able to display any number of data points on the map.

The last chapter talked about the process of using the framework to describe data and deploying the index server to make queries more effective. The deployment process was detailed for each of the components necessary to run the index server and a convenient way of deploying the whole stack, which uses the Docker technology, was suggested.

In conclusion, the Andruian framework created in this thesis can be successfully used for linking data to the RÚIAN registry. Any data source may be

used in place of the RÚIAN registry as long as it exposes a SPARQL endpoint and appropriate metadata for it exists. An index server may be deployed to make spatial queries significantly more effective. The Android application demonstrates the functionality of the framework by displaying places from various datasets across the whole of the Czech Republic.

Future work

There are several areas this work can be expanded upon. The framework core - the data definition schema - could be adjusted to allow publishing data in static files, without having to set up a SPARQL server. In the current state, the index server expects to consume all data from SPARQL servers. However, it is conceivable that a data publisher with only a small set of data, such as a set of cafés, will not be willing to undergo the effort of deploying a server just to link the data to its location. If the dataset was indeed small, it might not even require setting up an index server.

The data definition parser library as well as the index server could be improved to natively understand more well-known RDF links. Currently only *skos:prefLabel* and *schema:name* are processed automatically to assign a name to each indexed element. Other information may perhaps be added, such as a description, an image, publisher, and more.

The index server currently requires more operating memory as datasets get bigger. The reason for that is the fact that the whole response from a SPARQL endpoint is kept in memory at once. One possible improvement is to add support for Virtuoso's scrollable cursor[74], which would reduce the memory requirement by allowing the response to be split and queried in chunks. Unfortunately, at the time of writing, executing the indexing query on a Virtuoso SPARQL endpoint crashes the whole server[75].

Bibliography

[1] Linked Open Vocabularies. Available from: http://lov.okfn.org/dataset/lov/

[2] Becker, C.; Bizer, C. DBpedia Mobile: A Location-Enabled Linked Data Browser. Available from: http://ceur-ws.org/Vol-369/paper13.pdf

[3] DBpedia Places - Android Apps on Google Play. Available from: https://play.google.com/store/apps/details?id=com.lauer.dbpediaplacesandroid

[4] LinkedPipes Visualization. Available from: https://visualization.linkedpipes.com/

[5] Struktura a popis výměnného formátu RÚIAN (VFR). Available from: http://www.cuzk.cz/Uvod/Produkty-a-sluzby/RUIAN/2-Poskytovani-udaju-RUIAN-ISUI-VDP/Vymenny-format-RUIAN/Vymenny-format-RUIAN-(VFR)/Struktura-a-popis-VFR-1_8_0.aspx

[6] Applying MVP in Android. Available from: https://www.grapecity.com/en/blogs/applying-mvp-in-android

[7] What is data silo? Available from: http://searchcloudapplications.techtarget.com/definition/data-silo

[8] Heath, T.; Bizer, C. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, first edition, ISBN 9781608454310.

[9] Resource Description Framework (RDF). Available from: https://www.w3.org/RDF/

[10] RDFSyntax. 2011-01-27. Available from: https://www.w3.org/wiki/RdfSyntax

[11] RDF 1.1 Turtle. 2014-02-25. Available from: https://www.w3.org/TR/turtle/

[12] CURIE Syntax 1.0. Available from: https://www.w3.org/TR/2010/NOTE-curie-20101216/

[13] RDF Vocabulary Description Language 1.0: RDF Schema. Available from: https://www.w3.org/2001/sw/RDFCore/Schema/200203/

[14] SPARQL Query Language for RDF. Available from: https://www.w3.org/TR/rdf-sparql-query/

[15] DBpedia - Contributing Persons and Organizations. Available from: https://web.archive.org/web/20140921021528/http://wiki.dbpedia.org/Team

[16] lod-cloud.net. Available from: http://lod-cloud.net/versions/2017-08-22/lod.svg

[17] About — DBpedia. Available from: http://wiki.dbpedia.org/about

[18] semarglproject/semargl: Highly performant, lightweight framework for linked data processing. Available from: https://github.com/semarglproject/semargl

[19] Apache Jena. Available from: https://jena.apache.org/index.html

[20] lencinhaus/androjena: porting of Jena to Android. Available from: https://github.com/lencinhaus/androjena

[21] sbrunk/jena-android: This project aims to make the Apache Jena Framework usable on Android. Available from: https://github.com/sbrunk/jena-android

[22] Goodbye Sesame, hello RDF4J! Available from: http://rdf4j.org/2016/05/18/goodbye-sesame-hello-rdf4j/

[23] Spatial searches with SPARQL. Available from: https://jena.apache.org/documentation/query/spatial-query.html

[24] Apache Lucene - Apache Lucene Core. Available from: http://lucene.apache.org/core/

[25] Apache Solr. Available from: http://lucene.apache.org/solr/

[26] GeoSPARQL - A Geographic Query Language for RDF Data — OGC. Available from: http://www.opengeospatial.org/standards/geosparql

[27] GeoSPARQL support – GraphDB SE 8.5 documentation. Available from: http://graphdb.ontotext.com/documentation/standard/geosparql-support.html

[28] Virtuoso Geo Spatial Enhancements. Available from: http://vos.openlinksw.com/owiki/wiki/VOS/VirtGeoSPARQLEnhancementDocs

[29] Introduction - Material Design. Available from: https://material.io/guidelines/material-design/introduction.html

[30] LACKO, L. *Mistrovstvi - Android*. Brno: Computer Press, first edition, ISBN 978-80-251-4875-4.

[31] Location Strategies — Android Developers. Available from: https://developer.android.com/guide/topics/location/strategies.html

[32] RÚIAN. 2016. Available from: https://www.cuzk.cz/ruian/RUIAN.aspx

[33] Registr územní identifikace, adres a nemovitostí. Available from: https://www.cuzk.cz/Uvod/Produkty-a-sluzby/RUIAN/7-Publicita-projektu/RUIAN/letak_publicita.aspx

[34] ČÚZK - Výměnný formát RÚIAN (VFR). Available from: http://www.cuzk.cz/vfr

[35] Cron Trigger Tutorial. Available from: http://www.quartz-scheduler.org/documentation/quartz-2.x/tutorials/crontrigger.html

[36] DCAT-AP v1.1. Available from: https://joinup.ec.europa.eu/release/dcat-ap-v11

[37] SHACL Property Paths. Available from: https://www.w3.org/TR/shacl/#property-paths

[38] Klímek, J. MI-SWE.16 Semantic web Lecture – Linked Data Patterns [online]. [cit. 2018-03-19]. Available from: https://edux.fit.cvut.cz/courses/MI-SWE.16/_media/lectures/mi-swe-ld-patterns.pdf

[39] Spring Framework. Available from: https://projects.spring.io/spring-framework/

[40] SPARQL 1.1 Federated Query. Available from: https://www.w3.org/TR/sparql11-federated-query/

[41] andruian/indexer. Available from: https://github.com/andruian/indexer

[42] Apache Jena - TDB. Available from: http://jena.apache.org/documentation/tdb/

[43] UML 2 Component Diagram. Available from: https://www.sparxsystems.com.au/resources/uml2_tutorial/uml2_componentdiagram.html

[44] Properties File Format. Available from: https://docs.oracle.com/cd/E23095_01/Platform.93/ATGProgGuide/html/s0204propertiesfileformat01.html

[45] Heatmap Faceting — Spatial Search — Apache Solr Reference Guide 6.6. Available from: https://lucene.apache.org/solr/guide/6_6/spatial-search.html#SpatialSearch-HeatmapFaceting

[46] OkHttp. Available from: http://square.github.io/okhttp/

[47] Butter Knife. Available from: http://jakewharton.github.io/butterknife/

[48] Room Persistence Library — Android Developers. Available from: https://developer.android.com/topic/libraries/architecture/room.html

[49] Mockito framework site. Available from: http://site.mockito.org/

[50] Espresso — Android Developers. Available from: https://developer.android.com/training/testing/espresso/index.html

[51] Color wheel — Color schemes - Adobe CC. Available from: https://color.adobe.com

[52] andruian — GitHub. Available from: https://github.com/andruian

[53] Maven Repository: xerces » xercesImpl » 2.11.0. Available from: https://mvnrepository.com/artifact/xerces/xercesImpl/2.11.0

[54] Spring Boot. Available from: https://projects.spring.io/spring-boot/

[55] Google Maps JavaScript API — Google Developers. Available from: https://developers.google.com/maps/documentation/javascript/

[56] AsyncTask and error handling on Android. Available from: https://stackoverflow.com/questions/1739515/asynctask-and-error-handling-on-android

[57] Google Maps Android Marker Clustering Utility. Available from: https://developers.google.com/maps/documentation/android-api/utility/marker-clustering

[58] JDK 1.6 and Xerces. Available from: https://stackoverflow.com/questions/7794281/jdk-1-6-and-xerces

[59] xerces-for-android. Available from: https://code.google.com/archive/p/xerces-for-android/

[60] eclipse/rdf4j XMLReaderFactory.java. Available from: https://github.com/eclipse/rdf4j/blob/4c2b7e755e8375486248647e3977c0a30f8ddf45/util/src/main/java/org/eclipse/rdf4j/common/xml/XMLReaderFactory.java#L24

[61] Travis CI - Test and Deploy Your Code with Confidence. Available from: https://travis-ci.org/

[62] andruian/ddfparser. Available from: https://github.com/andruian/ddfparser

[63] andruian/viewlink: Android client app for the Andruian framework. Available from: https://github.com/andruian/viewlink

[64] Google URL Shortener. Available from: https://goo.gl/

[65] Pastebin.com - #1 paste tool since 2002! Available from: https://pastebin.com/

[66] LinkedPipes ETL. Available from: https://etl.linkedpipes.com/

[67] ČÚZK - Atributy stavebního objektu. Available from: https://www.cuzk.cz/Uvod/Produkty-a-sluzby/RUIAN/2-Poskytovani-udaju-RUIAN-ISUI-VDP/Ciselniky-ISUI/Atributy-stavebniho-objektu.aspx#CE_VYBAVENI_VYTAHEM

[68] stain/jena-fuseki - Docker Hub. Available from: https://hub.docker.com/r/stain/jena-fuseki/

[69] library/mongo - Docker Hub. Available from: https://hub.docker.com/_/mongo/

[70] melkamar/solr-indexer - Docker Hub. Available from: https://hub.docker.com/r/melkamar/solr-indexer/

[71] melkamar/indexer - Docker Hub. Available from: https://hub.docker.com/r/melkamar/indexer/

[72] Docker Cloud. Available from: https://cloud.docker.com/swarm/

[73] Andruian ViewLink - Android Apps on Google Play. Available from: https://play.google.com/store/apps/details?id=cz.melkamar.andruian.viewlink

[74] Working with SPARQL endpoint constraints via LIMIT & OFFSET. Available from: http://vos.openlinksw.com/owiki/wiki/VOS/VirtTipsAndTricksHowToHandleBandwidthLimitExceed

[75] Server crashes with Segmentation fault when running SPARQL federated query. Available from: https://github.com/openlink/virtuoso-opensource/issues/734

Appendix A

Acronyms

FAB Floating Action Button

GUI Graphical User Interface

HTTP HyperText Transfer Protocol

JVM Java Virtual Machine

POJO Plain Old Java Object

RDF Resource Description Framework

RÚIAN Registr Územní Identifikace, Adres a Nemovitostí

SPARQL SPARQL Protocol and RDF Query Language

URI Uniform Resource Identifier

VFR Výměnný Formát Ruian

XML Extensible Markup Language

Appendix B

Resources

B.1 Index SPARQL query template

This is the SPARQL query template used for incremental indexing. Elements enclosed in curly braces ({}) are placeholders that have to be replaced before the query is sent to an endpoint.

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX eu: <http://eulersharp.sourceforge.net/2003/03swap/log-rules#>
PREFIX ru: <http://purl.org/imbi/ru-meta.owl#>
prefix ex: <http://example.org/>
prefix s: <http://schema.org/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix skos: <http://www.w3.org/2004/02/skos/core#>

SELECT distinct ?dataObj ?locationObj ?lat ?long
     ?dataClassType ?__prefLab__ ?__name__ {selectProps}
WHERE {
  BIND(<{dataClassUri}> as ?dataClassType)

  ?dataObj a <{dataClassUri}>;
         {pathToLocClass} ?locationObj;
         .

  OPTIONAL {
    ?dataObj skos:prefLabel ?__prefLab__.
  }
  OPTIONAL {
    ?dataObj s:name ?__name.
  }
```

```
  #
  # Optional filter when reindexingto exclude all
  # objects that already exist
  #
  # Example contents of excludeDataObjects:
  #    ?dataObj != <http://example.org/linkedobject-24481611> &&
  #    ?dataObj != <http://example.org/linkedobject-72715057> &&
  #
  # This will filter out the two objects listed.
  # - Note that each line/expression MUST end with
  #   the && operator, including the last one, because
  #   there is a trailing True expression in the query template.
  #   The reason for that is to avoid parsing error
  #   thrown by FILTER() - there must be something in the parentheses.
  #
  FILTER(
    {excludeDataObjects}
    True
  )

  #
  # Mapping of selectProps - name of any selectProp
  #   must NOT be any of the reserved ones
  #   (dataObj, locationObj etc.)
  # Example mapping:
  #   ?dataObj ex:a/ex:b/ex:c ?selectPropA .
  #
  # There will be one line per each selectProp
  {selectPropsMapping}

  #
  # Federated query for the location sparql controller.
  #   [lat,long]LocationPathForLocationClass will contain a
  #   property path from the Location class to its coordinates.
  SERVICE <{locationSparqlEndpoint}> {
    ?locationObj {latLocationPathForLocationClass} ?lat;
                 {longLocationPathForLocationClass} ?long;
    .

  }
}
```

B.2 Data definition vocabulary

This is the data definition vocabulary in a Turtle serialization defined using RDF schema language. It formally defines the vocabulary as proposed in section 2.5 and shown in Figure C.1.

```
@prefix andr: <http://purl.org/net/andruian/datadef#> .
@prefix sp: <http://spinrdf.org/sp#> .
@prefix s: <http://schema.org/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

############
# Classes #
############

andr:DataDef a rdfs:Class;
    rdfs:label "DataDef";
    rdfs:comment """The definition of a single data mapping in
the Andruian framework. That is a mapping of Source class
instances to instances of the Location class. See
https://github.com/andruian/example-data for more information.""";
    .

andr:LocationClassDef a rdfs:Class;
    rdfs:label "LocationClassDef";
    rdfs:comment """The definition of a Location class and its
metadata. That includes the SPARQL endpoint where instances
of this class may be found and property paths specifying how to
get the Location class' latitude and longitude.""";
    .

andr:SourceClassDef a rdfs:Class;
    rdfs:label "SourceClassDef";
    rdfs:comment """The definition of a Source class and its metadata.
That includes the SPARQL endpoint where instances of this class
may be found, the property path leading to the linked Location
class and properties that should be indexed on the index server
for faster retrieval.""";
    .

andr:SelectPropertyDef a rdfs:Class;
    rdfs:label "SelectPropertyDef";
```

```
    rdfs:comment """Definition of a Source class' property that
should be indexed on the index server. The definition specifies
name to use for the property and a property path leading to it.""";
    .

andr:IndexServer a rdfs:Class;
    rdfs:label "IndexServer";
    rdfs:comment """An index server for the Andruian framework.
It is optional - if not present, clients fall back to naive queries.
They are slower though.""";
    .

andr:LocationClassPathsSource a rdfs:Class;
    rdfs:label "LocationClassPathsSource";
    rdfs:comment """A source of property paths leading from an
instance of a class to its latitude and longitude.""";
    .

andr:ClassToLocPath a rdfs:Class;
    rdfs:label "ClassToLocPath";
    rdfs:comment """A property path for a particular class
leading to its latitude and longitude.""";
    .

andr:PropertyPath a rdfs:Class;
    rdfs:label "A path of one or more properties";
    rdfs:comment """This class is equivalent to a predicate sequence
SHACL property path. SHACL does not define a RDFS class for property
paths, but only refers to them as to a plain rdf:Resource.
andr:PropertyPath class is defined in Andruian data definition schema
to make the diagram and property domains easily readable.

A predicate or sequence path may be used wherever andr:PropertyPath
is expected. A predicate path is a single property in place of the object;
a sequence path is a RDF list containing two or more properties in
place of the object.""".

###############
# Properties #
###############

#
# DataDef
#
```

```
andr:sourceClassDef a rdf:Property;
    rdfs:label   "sourceClassDef";
    rdfs:comment """A definition of the Source Class in the Andruian
framework. This class is typically user-defined and links to a
Location class.""";
    rdfs:range   andr:SourceClassDef;
    rdfs:domain  andr:DataDef;
    .

andr:indexServer a rdf:Property;
    rdfs:label "indexServer";
    rdfs:comment """A definition of an Index Server in the Andruian
framework. This is an optional property.""";
    rdfs:range andr:IndexServer;
    rdfs:domain  andr:DataDef;
    .

andr:uri a rdf:Property;
    rdfs:label "uri";
    rdfs:comment """A link where the subject of the property
can be accessed.""";
    rdfs:range rdf:Resource;
    .

andr:version a rdf:Property;
    rdfs:label "version";
    rdfs:comment """An integer representing version of the subject
of the property.""";
    rdfs:range xsd:integer;
    .

andr:class a rdf:Property;
    rdfs:label "class";
    rdfs:comment """A specification of a resource type (class)
relevant to the subject of this property.""";
    rdfs:range rdfs:Class;
    .

#
# SourceClassDef
#
andr:selectProperty a rdf:Property;
    rdfs:label "selectProperty";
    rdfs:comment """Defines zero or more andr:SelectProperty
```

```
instances for the subject of this property. This is an optional
property, it may be specified multiple times.""";
    rdfs:range andr:SelectProperty;
    rdfs:domain  andr:SourceClassDef;
    .

andr:propertyPath a rdf:Property;
    rdfs:label "propertyPath";
    rdfs:comment """Defines a property path. The object
of this property must be a SHACL predicate or sequence path. See
https://www.w3.org/TR/shacl/#property-path-predicate for more details.""";
    .

andr:pathToLocationClass a rdf:Property;
    rdfs:label "pathToLocationClass";
    rdfs:comment """Defines a property path that, when applied
to the subject of the property, leads to an instance of the Location
Class linked to the subject. The object of this property must be
a SHACL predicate or sequence path. See
https://www.w3.org/TR/shacl/#property-path-predicate for more details.""";
    rdfs:domain  andr:SourceClassDef;
    .

andr:sparqlEndpoint a rdf:Property;
    rdfs:label "sparqlEndpoint";
    rdfs:comment """A definition of an URI where a SPARQL endpoint
relevant to the property subject may be reached.""";
    rdfs:range rdf:Resource;
    .

andr:sourceClassDef a rdf:Property;
    rdfs:label "sourceClassDef";
    rdfs:comment """A Source Class definition for this DataDef.""";
    rdfs:range andr:SourceClassDef;
    rdfs:domain  andr:DataDef;
    .


#
# LocationClassDef
#
andr:locationClassDef a rdf:Property;
    rdfs:label "locationClassDef";
    rdfs:comment """A Location Class definition for this DataDef.""";
```

```
    rdfs:range andr:LocationClassDef;
    rdfs:domain  andr:DataDef;
    .

andr:locationClassPathsSource a rdf:Property;
    rdfs:label "locationClassPathsSource";
    rdfs:comment """Defines a source object for property paths
describing the path from an entity of a given class to the
latitude/longitude coordinates associated with such class. The
object of this property will link to one or more property path definitions.""";
    rdfs:range andr:LocationClassPathsSource;
    rdfs:domain  andr:LocationClassDef;
    .

andr:includeRdf a rdf:Property;
    rdfs:label "includeRdf";
    rdfs:comment """Specifies an URL where an RDF file is
located. The consumer of this dataset shall download the linked file
and include the data contained there in their internal model
before processing any other properties of the subject.

This property is an analogy to import statements in programming
languages - only it imports a remote file.""";
    rdfs:range rdf:Resource;
    rdfs:domain andr:LocationClassDef;
    .

andr:classToLocPath a rdf:Property;
    rdfs:label "classToLocPath";
    rdfs:comment """Defines property paths for a single resource
type (class), leading from an object of such class to its latitude/longitude.""";
    rdfs:range andr:ClassToLocPath;
    .

andr:latPath a rdf:Property;
    rdfs:label "latPath";
    rdfs:comment """Defines a property path linking to the latitude
coordinate. The object of this property must be a SHACL predicate
or sequence path. See https://www.w3.org/TR/shacl/#property-path-predicate
for more details.""";
    .

andr:longPath a rdf:Property;
    rdfs:label "longPath";
```

```
    rdfs:comment """Defines a property path linking to the longitude
coordinate. The object of this property must be a SHACL predicate
or sequence path. See https://www.w3.org/TR/shacl/#property-path-predicate
for more details.""";
    .
```

B.3 Example data definition

This is an example of a data definition using the data definition vocabulary.

The data definition is named *Vowel street names*, has an index server at url `http://andruian.melkamar.cz` where objects of class *http://example.org/SourceObjectAEIOU* may be queried. The same objects are available from a SPARQL endpoint at url `http://andruian.melkamar.cz:3030/streets/query`. The objects are linked to location objects of class *ruian:AdresníMísto* through a property path. The path consists of three predicates, twice `http://a.property` and then `http://link.to.ruian`.

The property path defining how to get latitude and longitude coordinates for objects of class *ruian:AdresníMísto* is provided by an object identified by IRI `<http://purl.org/net/andruian/location-sources/ruian#locClassPathsSource>`. This object is contained in a RDF file at url `http://purl.org/net/andruian/location-sources/ruian`.

```
@prefix andr: <http://purl.org/net/andruian/datadef#> .
@prefix ruian: <https://ruian.linked.opendata.cz/slovník/> .
@prefix sp: <http://spinrdf.org/sp#> .
@prefix s: <http://schema.org/> .
@prefix ex: <http://example.org/> .
@prefix : <http://foo/> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

:dataDefVowel
    a                  andr:DataDef;
    andr:locationClassDef  :locationDef;
    andr:sourceClassDef :sourceClassDef;
    andr:indexServer  :indexServer;
    skos:prefLabel "Vowel street names";
    .

#
# INDEX SERVER
#
:indexServer
    a        andr:IndexServer;
    andr:uri <http://andruian.melkamar.cz>;
```

```
    andr:version 1;
    .

#
# DATA DEFINITION
#
:sourceClassDef
    a                          andr:SourceClassDef;
    andr:sparqlEndpoint
        <http://andruian.melkamar.cz:3030/streets/query>;

    andr:class
        <http://example.org/SourceObjectAEIOU>;

    andr:pathToLocationClass (
            <http://a.property>
            <http://a.property>
            <http://link.to.ruian>
    );

    andr:selectProperty [ a andr:SelectProperty;
                          s:name "PSC";
                          andr:propertyPath (
                              <http://a.property>
                              <http://psc.org>
                          )
                        ];

    andr:selectProperty [ a andr:SelectProperty;
                          s:name "StreetNum";
                          andr:propertyPath (
                              <http://a.property>
                              <http://a.property>
                              <http://number.prop>
                          )
                        ];

    andr:selectProperty [ a andr:SelectProperty;
                          s:name "StreetName";
                          andr:propertyPath (
                              <http://a.property>
                              <http://a.property>
                              <http://name.prop>
                          )
```

```
                        ];
    .


#
# LOCATION DEFINITON
#
:locationDef
    a                   andr:LocationDef;
    andr:sparqlEndpoint
        <https://ruian.linked.opendata.cz/sparql>;

    andr:class          ruian:AdresníMísto;

    andr:includeRdf
        <http://purl.org/net/andruian/location-sources/ruian>;

    andr:locationClassPathsSource
        <http://purl.org/net/andruian/location-sources/
                            ruian#locClassPathsSource>;
    .
```

B.4 Property path source for RÚIAN objects

The following listing defines property paths from all RÚIAN objects to their latitude and longitude. This file is also available online[22] and may be used in data definitions through the `andr:includeRdf` property. See data definition vocabulary in section 2.5 for more details.

```
@prefix andr: <http://purl.org/net/andruian/datadef#> .
@prefix ruian: <https://ruian.linked.opendata.cz/slovník/> .
@prefix s: <http://schema.org/> .
@prefix locsrc: <http://purl.org/net/andruian/location-sources/ruian#> .
@prefix ogcgml: <http://www.opengis.net/ont/gml#> .

#
# A source for location class -> coordinates paths for the RÚIAN registry.
#

locsrc:locClassPathsSource
    a andr:LocationClassPathsSource;
    andr:classToLocPath locsrc:stat;
```

[22] http://purl.org/net/andruian/location-sources/ruian

```
    andr:classToLocPath locsrc:kraj;
    andr:classToLocPath locsrc:okres;
    andr:classToLocPath locsrc:regionSoudrznosti;
    andr:classToLocPath locsrc:vusc;
    andr:classToLocPath locsrc:obec;
    andr:classToLocPath locsrc:castobce;
    andr:classToLocPath locsrc:katastralniuzemi;
    andr:classToLocPath locsrc:zsj;
    andr:classToLocPath locsrc:parcela;
    andr:classToLocPath locsrc:momc;
    andr:classToLocPath locsrc:mop;
    andr:classToLocPath locsrc:spravniobvod;
    andr:classToLocPath locsrc:stavebniobjekt;
    andr:classToLocPath locsrc:adresniMisto;
    .

locsrc:stat
    a           andr:ClassToLocPath;
    andr:class ruian:Stát;
    andr:lat    ( ruian:definičníBod s:geo s:latitude );
    andr:long   ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:kraj
    a           andr:ClassToLocPath;
    andr:class ruian:Kraj1960;
    andr:lat    ( ruian:definičníBod s:geo s:latitude );
    andr:long   ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:okres
    a           andr:ClassToLocPath;
    andr:class ruian:Okres;
    andr:lat    ( ruian:definičníBod s:geo s:latitude );
    andr:long   ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:regionSoudrznosti
    a           andr:ClassToLocPath;
    andr:class ruian:RegionSoudrznosti;
    andr:lat    ( ruian:definičníBod s:geo s:latitude );
    andr:long   ( ruian:definičníBod s:geo s:longitude );
    .
```

```
locsrc:vusc
    a           andr:ClassToLocPath;
    andr:class ruian:Vusc;
    andr:lat    ( ruian:definičníBod s:geo s:latitude );
    andr:long   ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:obec
    a           andr:ClassToLocPath;
    andr:class ruian:Obec;
    andr:lat    ( ruian:definičníBod s:geo s:latitude );
    andr:long   ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:castobce
    a           andr:ClassToLocPath;
    andr:class ruian:ČástObce;
    andr:lat    ( ruian:definičníBod s:geo s:latitude );
    andr:long   ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:katastralniuzemi
    a           andr:ClassToLocPath;
    andr:class ruian:KatastrálníÚzemí;
    andr:lat    ( ruian:definičníBod s:geo s:latitude );
    andr:long   ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:zsj
    a           andr:ClassToLocPath;
    andr:class ruian:Zsj;
    andr:lat    ( ruian:definičníBod ogcgml:pointMember s:geo s:latitude );
    andr:long   ( ruian:definičníBod ogcgml:pointMember s:geo s:longitude );
    .

locsrc:parcela
    a           andr:ClassToLocPath;
    andr:class ruian:Parcela;
    andr:lat    ( ruian:definičníBod s:geo s:latitude );
    andr:long   ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:momc
    a           andr:ClassToLocPath;
```

```
    andr:class ruian:Momc;
    andr:lat   ( ruian:definičníBod s:geo s:latitude );
    andr:long  ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:mop
    a          andr:ClassToLocPath;
    andr:class ruian:Mop;
    andr:lat   ( ruian:definičníBod s:geo s:latitude );
    andr:long  ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:spravniobvod
    a          andr:ClassToLocPath;
    andr:class ruian:SprávníObvod;
    andr:lat   ( ruian:definičníBod s:geo s:latitude );
    andr:long  ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:stavebniobjekt
    a          andr:ClassToLocPath;
    andr:class ruian:StavebníObjekt;
    andr:lat   ( ruian:definičníBod s:geo s:latitude );
    andr:long  ( ruian:definičníBod s:geo s:longitude );
    .

locsrc:adresniMisto
    a          andr:ClassToLocPath;
    andr:class ruian:AdresníMísto;
    andr:lat   ( ruian:adresníBod s:geo s:latitude );
    andr:long  ( ruian:adresníBod s:geo s:longitude );
    .
```

B.5 SPARQL query for creating incremental testing data

```
CONSTRUCT {
?sourceObj a <http://example.org/SourceObjectATest>;
  <http://www.w3.org/2004/02/skos/core#prefLabel> ?aLabel;
  <http://link.to.ruian> ?addrPlace.

}
```

```
WHERE {
    ?obec a ruian:Obec;
          schema:name ?obecName.

    FILTER (STR(?obecName) = "Praha")

    ?ulice a ruian:Ulice;
           ruian:obec ?obec;
           schema:name ?uliceName .

    ?addrPlace a ruian:AdresníMísto;
               ruian:ulice ?ulice;
               ruian:psč ?psc;
               ruian:číslOrientační ?streetNum.

    FILTER (?streetNum < 20)
    BIND ( IRI(CONCAT("http://src.com/incrementaltest/", ENCODE_FOR_URI(?addrPlace))
    BIND ( CONCAT(?uliceName, " ", ?streetNum) as ?aLabel)

    FILTER (REGEX(?uliceName, "^[A]"))
}
```

B.6 Data definition created during a user scenario

```
@prefix andr: <http://purl.org/net/andruian/datadef#> .
@prefix ruian: <https://ruian.linked.opendata.cz/slovník/> .
@prefix sp: <http://spinrdf.org/sp#> .
@prefix s: <http://schema.org/> .
@prefix ex: <http://example.org/> .
@prefix : <http://foo/> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

:dataDefTEST
    a                   andr:DataDef;
    andr:locationClassDef  :locationDef;
    andr:sourceClassDef :sourceClassDef;
    andr:indexServer  :indexServer;
    skos:prefLabel "TEST";
    .

#
# INDEX SERVER
#
```

```
:indexServer
    a         andr:IndexServer;
    andr:uri <http://andruian.melkamar.cz>;
    andr:version 1;
    .

#
# DATA DEFINITION
#
:sourceClassDef
    a                          andr:SourceClassDef;
    andr:sparqlEndpoint
        <http://andruian.melkamar.cz:3030/NEW-DS/query>;

    andr:class
        <http://example.org/SourceObjectATest>;

    andr:pathToLocationClass (
            <http://link.to.ruian>
    );

    andr:selectProperty [ a andr:SelectProperty;
                          s:name "myNameForLabel";
                          andr:propertyPath skos:prefLabel
                        ];

    .


#
# LOCATION DEFINITON
#
:locationDef
    a                    andr:LocationDef;
    andr:sparqlEndpoint
        <https://ruian.linked.opendata.cz/sparql>;

    andr:class           ruian:AdresníMísto;

    andr:includeRdf
        <http://purl.org/net/andruian/location-sources/ruian>;

    andr:locationClassPathsSource
        <http://purl.org/net/andruian/location-sources/ruian#locClassPathsSource>;
```

.

Appendix C

Diagrams

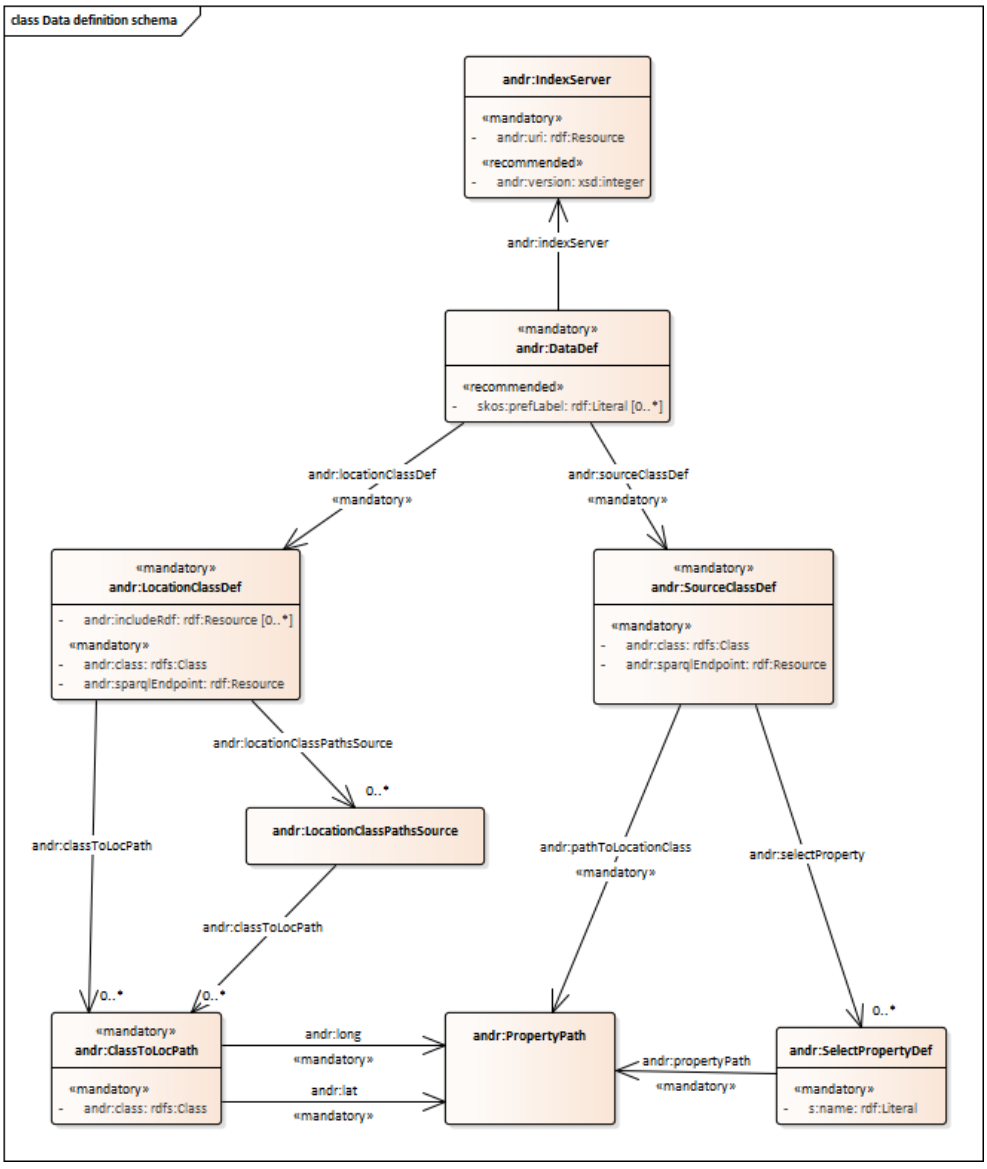

Figure C.1: Andruian Framework Data Definition schema

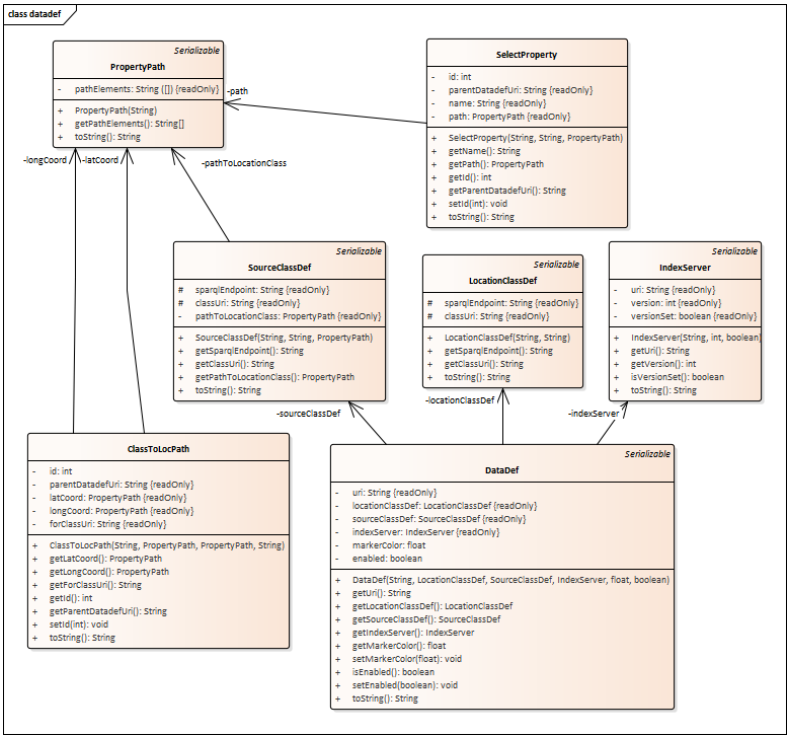

Figure C.2: Class diagram of the DataDef class of the ViewLink app

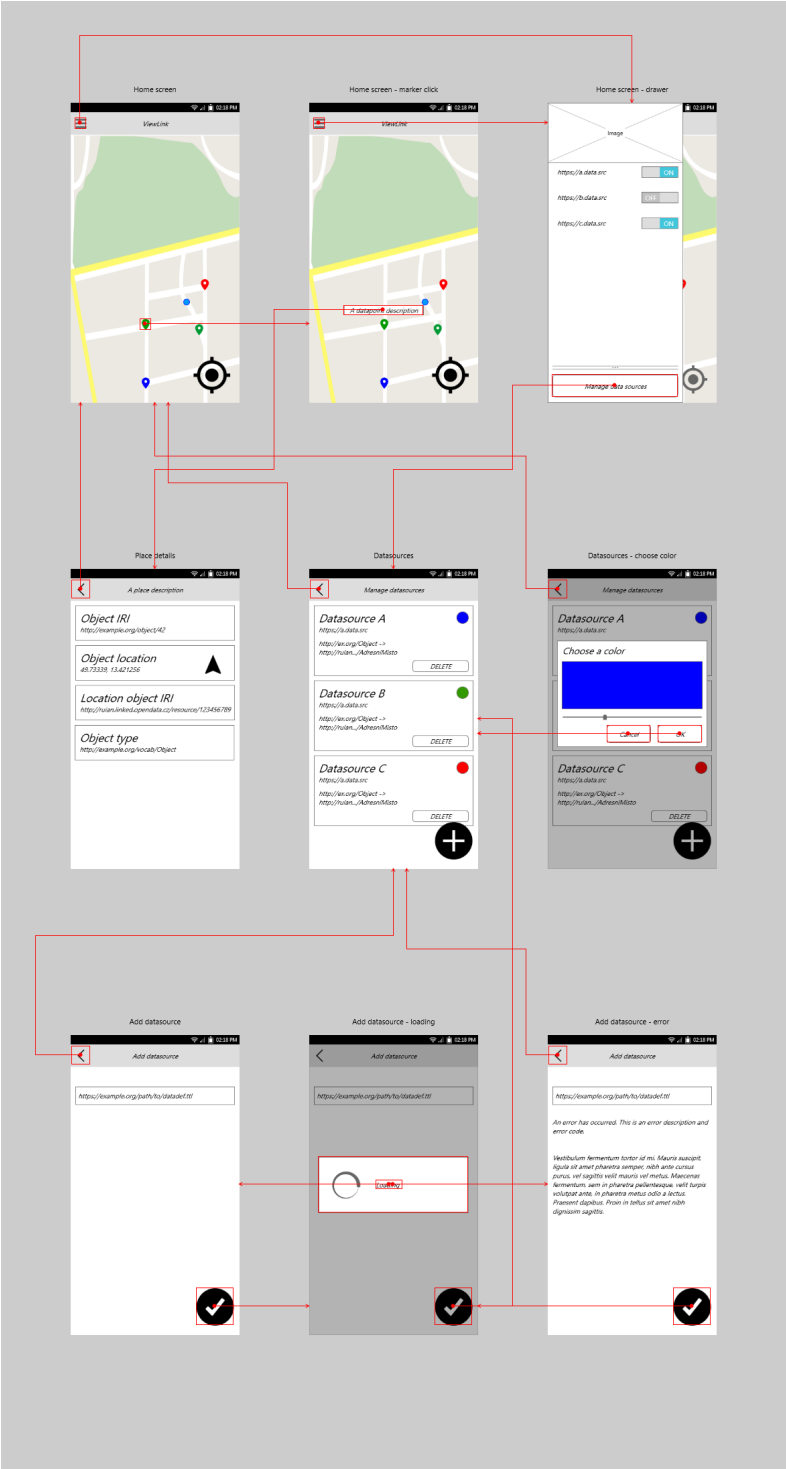

Figure C.3: UI flow of the ViewLink app

Appendix D

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format