Master's Thesis

Configurable Schema-Aware RDF Data Input Forms

Dávid Konkoly

April 2017

Albert-Ludwigs Universität Freiburg
Department of Computer Science
Chair of Databases and Information Systems

Candidate
Dávid Konkoly
Matr. number
3757311
Working period
18. 10. 2016 – 18. 04. 2017
Examiner
Prof. Dr. Georg Lausen
Supervisor
Victor Anthony Arrascue Ayala

Abstract

Kurzfassung

Kurzfassung auf Deutsch

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 RDFBones Project

The master thesis is written in the frame of the project called *RDFBones*. The goal of the project is to develop a data standard, as well a web application for documenting research activity related to biological anthropology. The challenge is that in anthropology each institute has different set of skeletal remains and have different research interest and scopes. So to achieve that the developed application can be used by various cases it has to extensible. For this reason RDF data model is applied, since it is more suitable for such purposes than relational data model.[4] Building web application is large effort, therefore the software is not developed from scratch, but an existing Semantic Web application framework, called VIVO is used. The advantage of VIVO that the it has a capability to adopt his interface to the ontology, which is highly desirable feature for our project.

1.2 Goal of the thesis

The problem of data creation through web application can be divided into two main parts. The first is the layout of the interface, that user interacts with, and the second is the dataset that has to be created by the process. The thesis work incorporates the design of a data model that is suitable for the declarative definition of the whole input problems, as well the development of software modules both for the client and server side that is capable to operate such high-level definition. The goal is achieve a such web application framework that can be employed for various problems without coding, so that the system can be developed rapidly by scientist without programming knowledge.

1.3 Thesis outline

The second chapter contains the background information, that is necessary to understand the problem the thesis aims to solve. The first two subsection handles the RDF data model and the ontologies applied in the project, while the third section is about the basics of the web applications.

The third chapter presents the problem statement. The first section discusses the scheme of the data that has to be created to document anthropological research, while the second section show what does it mean in terms of the web application programming.

The fourth and the fifth chapter are describing the proposed solution for the problem. The fourth chapter outlines the higher-level modeling of the application, and fifth in turn describes the how the implemented framework can operate upon the declarative definition.

The last, sixth chapter covers the conclusion and the evaluation of the achieved system, and present the further potential in the idea. Figure 1.1 illustrates the structure of the thesis, where the blue denotes the chapter with conceptual content, and the yellow in turn the ones with practical content.

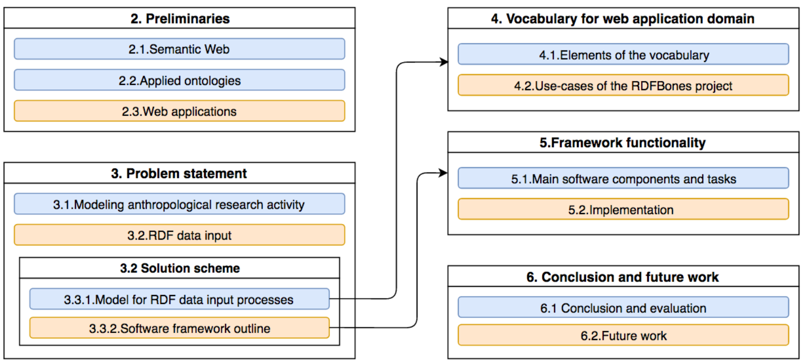

Figure 1.1: Structure of the chapters

Chapter 2

Preliminaries

2.1 Semantic Web

2.1.1 RDF

In RDF, abbreviation for Resource Description Framework, the information of the web is represented by means of triples. Each triple consists of a subject, predicate and object. The set of triples constitute to an RDF graph, where the subject and object of the triples are the nodes, the predicates are the edges of the graph. An RDF triple is called as well statement, which asserts that there is a relationship defined by the predicate, between subject and the object. The subjects and the objects are RDF resources. A resource can be either an IRI (Internationalized Resource Identifier) or a literal or a blank node (discussed later). A resource represents any physical or abstract entity, while literals hold data values like string, integer or datum [5].

Basically there are two types of triples, the one that links two entities to each other, and the other that links a literal to an entity. The former expresses a relationship between two entities, and the latter in turn assign an attribute to the entity. Common practice is to represent IRI with the notation prefix:suffix, where the prefix represents the namespace, and the expression means the concatenation of the namespace denoted by the prefix, with the suffix. This convention makes the RDF document more readable. The namespace of RDF is the http://www.w3.org/1999/02/22-rdf-syntax-ns#, whose prefix is in most cases "rdf". This is defined on the following way:

```
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns\#>
```

Listing 2.1: Prefix notation in RDF

Literals are strings consisting of two elements. The first is the lexical form, which is the actual value, and the second is the data type IRI. RDF uses the data types from XML schema. The prefix (commonly xsd) is the following :

```
@prefix xsd:   <http://www.w3.org/2001/XMLSchema\#>
```

Listing 2.2: XML Schema prefix

So a literal value in RDF looks as follows:

```
"Some literal value"^^xsd:string
```

Listing 2.3: String value as RDF data

The RDF vocabulary provides some built-in IRIs. The two most important are, the rdf:type property, and the rdf:Property class. The meaning of the triples, where the predicate is the property rdf:type is that the subject IRI is the instance of the class denoted by the object. Therefore the following statement holds in the RDF vocabulary:

```
rdf:type        rdf:type        rdf:Property
```

Listing 2.4: rdf:type is a property

It is maybe confusing that an IRI appears in a triple as subject and predicate as well, but we will see by the RDFS vocabulary that it is inevitable to express rules of the language. To be able to represent information about a certain domain, it is necessary to extend the RDF vocabulary with properties and classes. The classes will be discussed in the next section, but here it is explained how custom properties can be defined. The namespace of the example is the following:

```
@prefix example:   <http://example.org\#>
```

Listing 2.5: Example prefix

The example dataset intends to express information about people, which university they attend and how old are them. To achieve this two properties are needed:

```
eq:attends      rdf:type      rdf:Property  .
eq:age          rdf:type      rdf:Property  .
```

Listing 2.6: Example properties

The actual data about a person:

```
eg:JanKlein   eq:attends    eq:UniversityOfFreiburg  .
eg:JanKlein   eq:age        "21"^^xsd:integer
```

Listing 2.7: Example prefix

2.1.2 RDF Schema

The previous section gave an insight into RDF world by showing how can information stored by means of triples. However the explanation did not mention that each RDF dataset has to have scheme, which is also called ontology. The ontology describes the set of properties and classes and how are they are related to each other. RDFS provides a mechanism to define such ontologies using RDF triples. The most important elements of the RDFS vocabulary can be seen on the following image.

The two most important classes in the RDFS vocabulary is the rdfs:Class and the rdfs:Resource. The rdfs:Class is class, because it is the instance of itself, and the same way the rdfs:Resource is a class. The rdf:Propery and the rdfs:Literal are both classes as well. The rdfs:domain, rdfs:range, rdfs:subPropertyOf and rdfs:subClassOf are properties. Important to note that these properties are subjects and predicates in the same time in the RDFS vocabulary graph. Also they describe themselves like rdf:type. The properties rdfs:domain and rdfs:range describe for the property the type

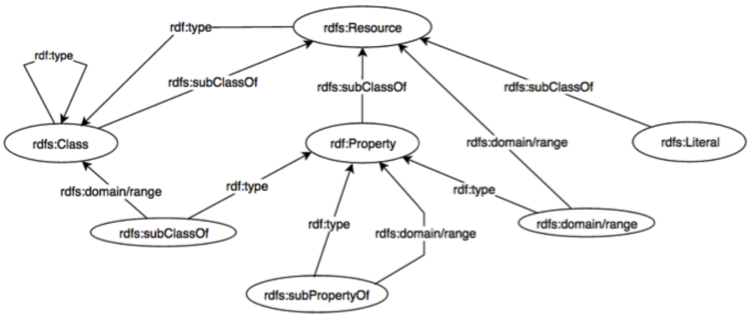

Figure 2.1: Main structure of the RDFS vocabulary

of the subject and object respectively, which with it can build a triple as predicate. The following image illustrates their meaning:

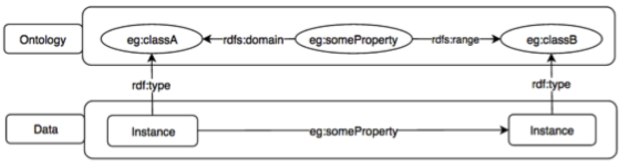

Figure 2.2: RDFS domain and range definition

Since both the rdfs:domain and rdfs:range are properties themselves, they have as well their domain and range, which is the class rdfs:Resource. The property rdfs:subClassOf expresses subclass relationship between classes. It means if a class B is a subclass of class A, and resource R is the instance of class B, then resource R is the instance of class A as well. Since it describes the relationship between two classes its both domain and range is the class rdfs:Class. The property rdf:subPropertyOf expresses the relationship between two properties. If property P2 is sub property of P1 and two instances are related by P2 then they are related by P1 as well. Its domain and range is the class rdf:Property. Now everything is given to define the ontology for the example of the previous section. [2]

2.1.3 OWL

OWL, abbreviation for Ontology Web Language is an extension of the RDFS vocabulary [3]. OWL allows expressing additional constraints on the data,

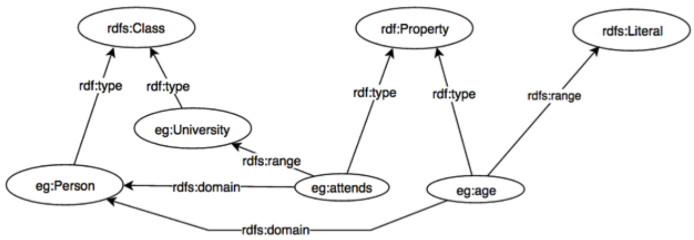

Figure 2.3: RDFS domain and range definition

above the range and domain definitions. These constraints are called restrictions. Restrictions are conventionally expressed by blank nodes. Blank nodes do not have IRIs, but it is defined through the triples in which they participate as a subject. For example a restriction stating that the instances of the class eg:FootballTeam can build a triple through the eg:hasPlayer property only with the instances of eg:FootballPlayer class can be expressed the following way:

```
eg:FootballTeam rdfs:subClassOf [
  rdf:type      owl:Restriction ;
  owl:onProperty     eg:hasPlayer ;
  owl:allValuesFrom eg:FootballPlayer .
]
```

Listing 2.8: OWL restriction in N3 format

owl:Restriction is class and owl:onProperty and owl:allValuesFrom are properties. It can be seen that class, on which the restriction applies is the subclass of the restriction blank node. Furthermore OWL is capable of expressing qualified cardinality restriction. For example the statement that a basketball team has to have exactly five players, look as follows in OWL:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX eg: <http://example.org>

eg:BasketballTeam rdfs:subClassOf [
  rdf:type   owl:Restriction ;
  owl:onProperty   eg:hasPlayer ;
  owl:onClass    eg:Player ;
```

```
  owl:qualifiedCardinality "5"^^xsd:nonnegativeInteger
] .
```

Listing 2.9: OWL restriction in N3 format

These two examples cover the thesis related features of OWL. The next image depicts the OWL vocabulary.

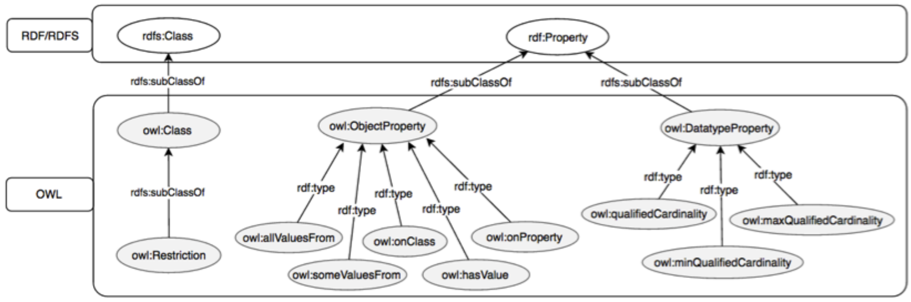

Figure 2.4: A subset of OWL vocabulary

There are two new class types are the owl:Class and the owl:Restriction. The rdf:Property has two subclasses, the owl:ObjectProperty and owl:DataTypeProperty. owl:ObjectProperty represent the properties that links instances to instances, and the owl:DataTypeProperty is those that link instances to literals. The following two images shows the domain and range definitions of the OWL properties used to describe restrictions.

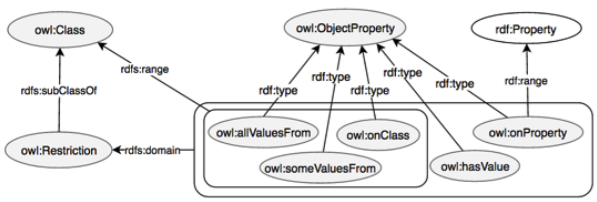

Figure 2.5: OWL object properties

2.1.4 SPARQL

SPARQL is a query language for querying data in RDF graphs. A SPARQL query is a definition of a graph pattern through variables and constants. The following example query returns all IRIs that represent a football player:

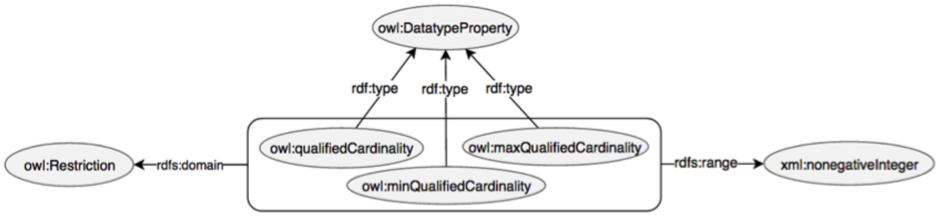

Figure 2.6: Properties for qualified cardinalities

```
SELECT ?player
WHERE {
  ?player    rdf:type   eg:FootballPlayer  .
}
```

Listing 2.10: SPARQL Query I.

In the example the query consist of only one triple. The subject is a variable and the predicate and the object are constant. Therefore the triple store looks all the triples and checks the predicate is rdf:type and the object is eg:FootballPlayer. It is well possible to not just ask the IRI of the players but further information by adding additional triples to the query in order to ask the name for example of the player:

```
SELECT ?player ?name
WHERE {
  ?player    rdf:type    eg:FootballPlayer  .
  ?player    eg:name     ?name  .
}
```

Listing 2.11: SPARQL Query II.

The result table in this case will contain two columns, one with the IRI of the person and one with their name. Important that it is as well possible to query blank nodes by introducing a variable for it. So if we want to list all the instances that are coming into question as player to a football team we can formulate the following query:

```
SELECT ?person ?name
WHERE {
```

```
  eg:FootballTeam  rdfs:subClassOf     ?restriction .
  ?restriction     rdf:type            owl:Restriction .
  ?restriction     owl:onProperty      eg:hasPlayer .
  ?restriction     owl:allValuesFrom   ?playerType .
  ?player          rdf:type            ?playerType .
  ?player          eg:name             ?name .
}
```

Listing 2.12: SPARQL Query III.

2.2 Applied Ontologies

Ontologies are used to describe types, relationships and properties of objects of a certain domain. It is a common practice to used already defined ontologies rather developing an own. The first reason is, that the development of an ontology is a complex and a tedious process, and requires a lot of resource. Secondly, it is reasonable to use standardized vocabularies, in order to make data from same domain but different sources inter-operable.

2.2.1 Foundational Model of Anatomy - *FMA*

The foundational Model of Anatomy ontology is an open source ontology written in OWL. FMA is a fundamental knowledge source for all biomedical domains, and it provides a declarative definition of concepts and relationships of the human body for knowledge based applications. It contains more than 70 000 classes, and 168 different relationships, and organize its entities into a deep subclass tree [6]. All types of anatomical entities are represented in FMA, like molecules, cells, tissues, muscles and of course bones. In our project we use only the subset of the FMA. The taken elements are the subclasses of the following two classes and the three properties:

- Classes

  Subdivision of skeletal system - fma:85544

  Bone Organ – fma:5018

- Properties

  fma:systemic_part_of

  fma:constitutional_part_of

  fma:regional_part_of

The class *Bone Organ* is the superclass of all bones in the human skeleton. Each bone belong to a skeletal subdivision and a skeletal subdivision can be a part of another skeletal subdivision. This relationship in both cases is expressed by the property *fma:systemic_part_of.* To define which bone organ belongs to which skeletal subdivision FMA uses OWL restrictions (see Figure 2.7). The properties *fma:constitutional_part_of* and *fma:regional_part_of fma:constitutional_part_of* are discussed later.

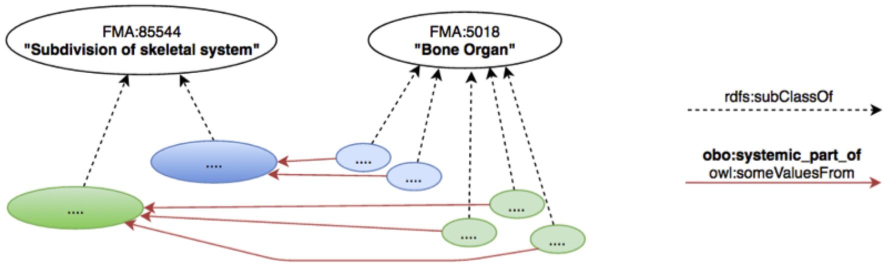

Figure 2.7: Ontology structure for skeleton

Finally the advantage of using the FMA ontology is that, if in the future further elements of the human body have to be addressed by the research processes, i.e. muscles, then these classes can be easily integrated to the currently applied subset.

2.2.2 Ontology for Biomedical Investigations - *OBI*

The aim of OBI ontology, is to provide the formal representation of the biomedical investigation in order to standardize the processes among different research communities. It is a result of a collaborative effort of several working groups, and it continuously evolving as new research methods are being developed. Its main function to describe the rules how biological and medical investigations have to be performed. OBI reuses terms from BFO *Basic Formal Ontology* IAO *Information Artifact Ontology* and OBO *Open Biological and Biomedical Ontologies* [1]. To define processes OBI uses the following three general classes:

- *Information Content Entity* - obo:IAO_0000030
- *Material Entity* - obo:BFO_0000040
- *Process* - obo:BFO_0000015

Information Content Entity represent results of a specific measurement, while Material Entity stands for the objects, on which the meausurements have been performed. The Process could mean any kind of step within an investigation, from the planning, through execution till the conclusion.

- *Planning* - obo:OBI_0000339
- *Study Design Execution* - obo:OBI_0000471
- *Drawing a conclusion* - obo:OBI_0000338

In our project the following three properties are used:

- *has part* - obo:BFO_00000051
- *has specified input* - obo:OBI_00000293
- *has specified output* - obo:OBI_00000299

2.3 Web applications

This chapter contains practical information about how web applications work. In section 2.3.1 the basic mechanism of data driven applications are discussed, like navigation between page, data display and creation. Section ?? then focuses on the applications that are using semantic technologies, and addresses what kind of architectural changes that means.

2.3.1 Client-sever architecture

A web application is program that runs on a machine, which is accessible through the web. The machine is called server, because its main purpose is to server request that are coming from the web browser. Web browsers are as well programs, but they run on personal computers, tablets, etc, and they are capable of sending request through web to the servers. The response to these requests are HTML document, which can be displayed by the browser.

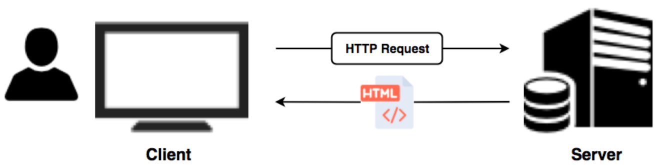

Figure 2.8: Client server communication

An HTML document contains definition of the elements of the pages, such as tables, buttons, etc. It contains as well so called CSS docuements

(Cascading Style Sheet), which is responsible for the definition of the style of the elements. Moreover to make the web pages more interactive, JavaScript (JS) de can be embedded to HTML as well.

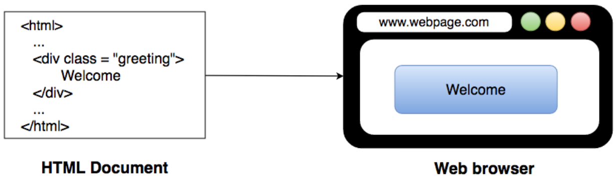

Figure 2.9: HTML document is interpreted by the browser

Initially web pages were static, which means that their only function was to show certain set of information. These applications usually web applications do not consist of one single page, but of several different pages. Like a web page for news, have normally a main page, and different sub pages for the particular topics. In order to navigate between the pages of the application, the HTML document contains links that trigger further HTTP requests. Links in HTML can be defined by means of the <a/> tag. The most important parameter of this tag is href, whose value contains the URL of the HTTP request. Let assume that an application's main page is accessible through the URL http://newsPortal.com. Common practice that subpages of the application can be called through various url-mappings, which means the main URL is extended with a keyword that denotes the page to be requested.

```
<a href="http://newsPortal.com/politics"> Politics </a>
<a href="http://newsPortal.com/sport"> Sport </a>
```

Listing 2.13: Example link definitions

If the user clicks on of these link (with the label 'Politics' and 'Sport') then these request will be sent to the news portal page. Each such request has to be served, differently to each mapping some routine has to be assigned. For example by Java web applications, the classes of the server that process the request are called servlets. On the next image it is shown, how the XML file defines, which class is responsible for the the mapping '/politics'.

```
<servlet-mapping>
  <url-pattern>/politics</url-pattern>
  <servlet-class>servlets.PoliticsController</servlet-class>
</servlet-mapping>
```

Listing 2.14: Java servlet mapping definition

Then the responsibility of the class *servlets.PoliticsController* is to respond the corresponding HTML page for the client. Figure 2.10 show the main structure of the applications, where the rectangles on the client side represent the different pages of the application.

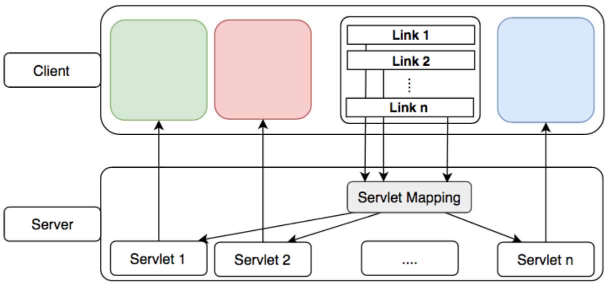

Figure 2.10: Navigation through the web application

2.3.2 Data driven web applications

This section aims to present the fundamentals of the web technologies that allows to build application for browsing and creating data. Modern web applications do not store the information in HTML documents. So the page loading process is not just the sending the HTML document, but a retrieval of a particular dataset, and the substitution into a web page. First of all the task of responding requires a query that retrieves that data from the database. By applications using relational data model, the tables and attributes are always modeled by classes of the used object oriented programming (OOP) language. So the data retrieval is the instantiation of the classes in scope.

Let assume that articles of a news portal is stored in a table with the attributes, id, type, title, summary and text. Then there has to be a class defined in the server code with the same attributes. To instantiate instances of the class, it is necessary to perform an SQL query.

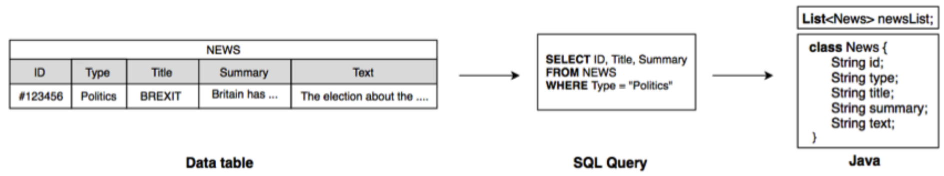

Figure 2.11: Data flow

The query results not only one instance of the News class, but a list (*List<News> newList*). To generate from this a HTML page that shows the articles, normally so called template engines are used. Templateing enables to define the HTML documents parametric, and passing them data, and they generate the result page automatically.

```
<#list newsList as news>
  <h3> ${news.title} </h3>
  <p> ${news.summary} </p>
  <a href = "http://newsPortal.com/wholeNews?id=${news.id}">
    Read more
  </a>
</#list>
```

Listing 2.15: Template file example

The template file is a description of how the data has to be converted into HTML document. It can be seen that it is possible for instance to declare a list on the input variable newsList. Then the template engine iterates through the News objects and by accessing its fields (title, summary, id) and generates the HTML for each element. So the complete flow of data from the database to the client looks as follows:

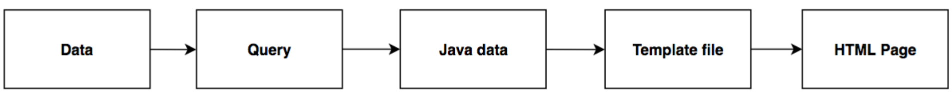

Figure 2.12: Flow of information from DB to client

The template shows only the summary of the article, but offers the following link:

```
http://newsPortal.com/wholeNews?id=${news.id}
```

The new feature is that after the url mapping there is a parameter *id*, and its value will be the database id of the web application. The idea is that this link redirects to the page where the whole article can be seen. So there has to be a servlet class defined to the mapping /wholeNews, which to perform the following query where the id is the input.

```
SELECT Text
FROM NEWS
WHERE ID = ${id}
```

Figure 2.13: SQL query with parameter

Thus it is achieved that different links are programmed to get access not only to different other pages, but to specific data items.

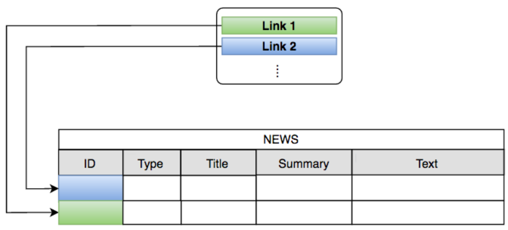

Figure 2.14: Links to data items

Web applications do not only just display existing data, but they allow the users to enter their new data. In HTML the element used for data input is called form. Form is a container, and it consists of particular form elements according to the data to be added.

Submitting the form to the server send an HTTP request with multiple parameters, where they are divided through the & character.

By the data entry creation the task of the controller is to get the values from the request an instantiate the class representing the data to be created. Then initialized class instance is passed to the database where the entered data will be persistently stored.

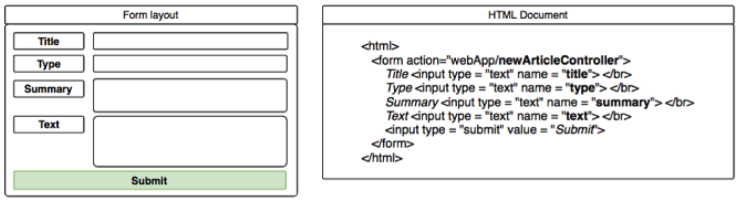

Figure 2.15: Form layout and HTML document

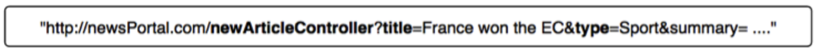

Figure 2.16: Request with parameters

2.3.3 Applications with RDF Data

This section aims to give an insight to web application that are based on RDF data. It will be covered what kind of requirements do the software have on the server side to create RDF data, and what is the difference between the RDF model based applications and the relational ones. The most important feature of RDF that the data scheme, namely the ontology is stored in RDF triples too, thus can be queried.

Therefore it is possible to generate web pages that can adapt to the ontology. The following query demonstrates that how it is possible to get all the instances, which are connected to a particular instance (#459828).

```
SELECT ?property ?relatedInstance
WHERE {
  ?instance       rdf:type        ?class .
  ?property       rdfs:domain    ?class .
  ?instance       ?property      ?relatedInstance .
  FILTER (?instance = #459828 ) .
}
```

Listing 2.16: Dynamic SPARQL query

The first two lines of the query defines the properties whose domain class is the type of the input instance, while the third asks for all triples with the possible properties. If the result of the query is then grouped based on properties, then the dataset can be displayed by a template using two lists.

```
String title = request.getParameter("title");
....
News news = new News(title, type, summary, text);
DatabaseConnector.insert(news);
```

Figure 2.17: Example Java routine for data storage

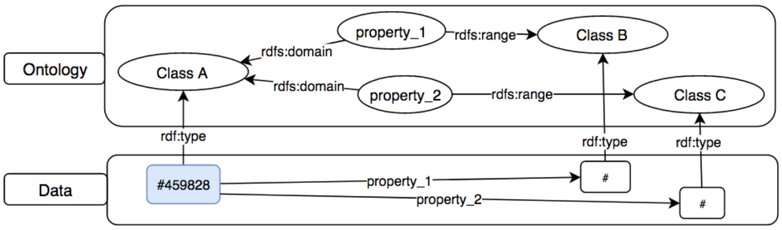

Figure 2.18: Ontology and data in RDF

The outer list ceates fields for the properties, and the inner show all the instances with that property.

```
<#list properties as property>
  <#list property.dataSet as instance>
  <#list>
</#list>
```

Listing 2.17: Ontology adaptive template file

The VIVO framework applied in the RDFBones project generates the pages for instances this way.

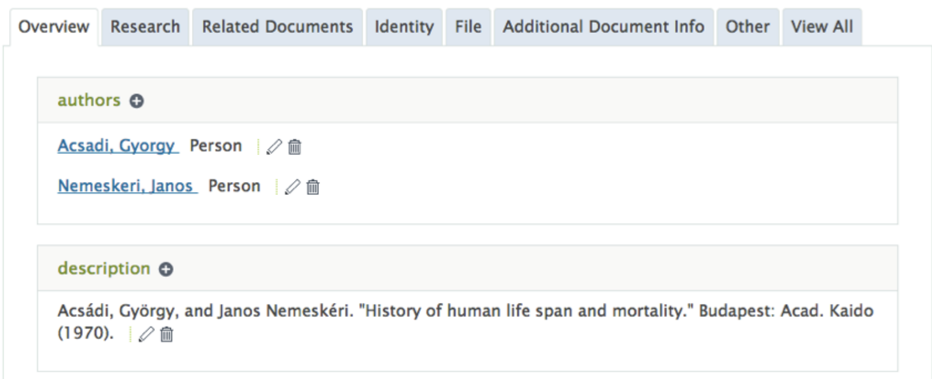

Figure 2.19: VIVO Profile page

Above the display of the existing data, RDF based applications are different as well in the data input mechanism. First of all, due to the fact that ontologies can contain thousand of class it is not an option to represent them all as classes of the server application language as well. It is time consuming and the system would loose its flexibility in the cases when new ontology subsets are supposed to be loaded. Therefore there are neither for each type of the database an entry form with a unique controller servlet, but more generic approaches are used. To define what dataset has to be created, semantic web based applications simply define them as a set of triples, with variables like in SPARQL, just the data flows the other way around.

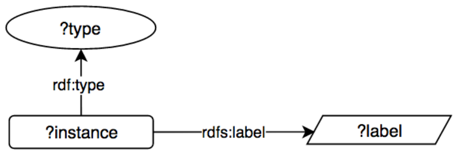

Figure 2.20: Triples representing a new instance

Figure 2.20 simple set of triples that have to be created by new data entry generation. The value of the variable textit?instance will be IRI that have not been used. This is an essential part of every triple store implementation that they provide unused uris for the server application for the new set of triples. The values of the *type* and label, are coming from the input form. The label is just like in the previous example, it is a string typed by the user it is stored as an attribute of the new entity. But the type of the instance is class IRI. The point is that the options of the selector field, from which the type value is coming, is filled with the results of a SPARQL query on the ontology. So for example if the entry form provide the possiblity to create any type of processes from the OBI ontology then the before the form loading the following query has to be executed.

```
SELECT ?class ?label
WHERE {
  ?class      rdf:subClassOf    obo:OBI_0000339 .
  ?class      rdf:label         ?label .
}
```

Listing 2.18: SPARQL query for the input form

will be a new unused IRI of the triple store, while the ?type variable come from the client from a selector field. The following image depicts this simple scheme of the data input process.

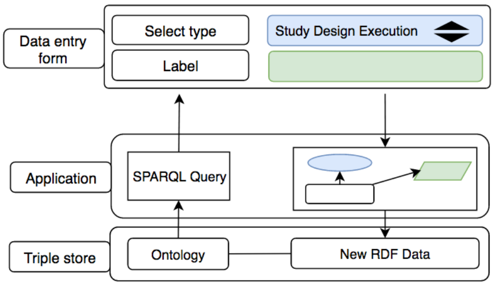

Figure 2.21: Data input scheme by RDF

Chapter 3

Problem Statement

3.1 Multi level data input

In the previous chapter it has been discussed how can we implement simple data input forms within the VIVO framework. The simplicity of the illustrated problems lied in that the number of instances which were created through the forms was constant, in particular one, and only their types and literal attributes were set by the user through HTML input elements.

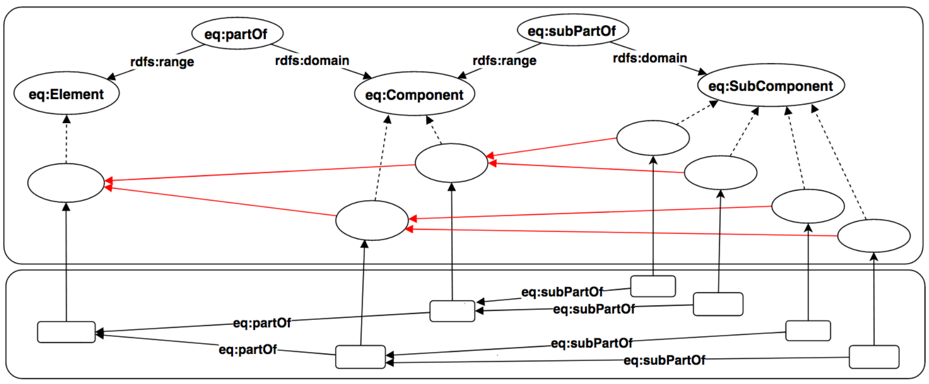

Figure 3.1: Ontology and RDF triples for complex entities

Nevertheless there are such entities that are more complex, and they consist of several sub part, that are represented in the ontology with further classes. Such as a skeletal subdivision consists of bone organs, or a study design execution consists of assays, which consist of input bones and output data. Consequently the RDF dataset for such entities incorporates multiple instances organized into a tree structure. Figure 3.1 shows and example ontology and an RDF dataset. The classes (ellipses) without notation are subclasses of the three main classes and their name does not play any role.

Such dataset means that the data input form has to offer such interface elements which allows the user to add the components step by step. Adding a component means in term of the form that a new sub form appears where in the simplest case its type and label can be set.

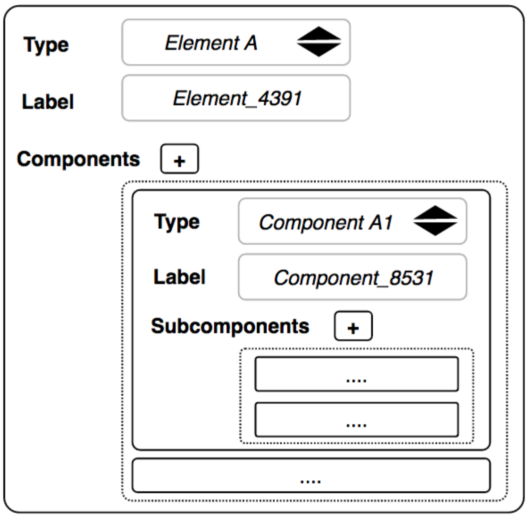

Figure 3.2: Multi level form

Figure 3.2 shows the layout of the form for multi level data. The additional element is the button for adding the components. The dotted rectangle stands for the container element that encompasses the sub forms. To realize such functionality a JavaScript routine is needed that adds the sub forms and put the data set by the user into a JSON object. The difference compared to the key-value pairs generated by the HTML *<form/>* tag, is that in this case the form data contains arrays to store the dynamically added elements. So JavaScript by each addition of sub component, creates a new data object and pushes into the appropriate arrays. The data of the added object will be values from the sub form. This is the basic principle how the multi level data is being created on the form. Listing 3.1 shows the JSON object generated during by form from Figure 3.2, where the objects are surrounded with ("{}"), while the arrays with ("[]"). After the submission the server has to process this object by iterating through the arrays of the objects, and generate the appropriate RDF triples.

```
{ type : "eq:elementA",
  label : "Element_4391",
  components : [
    { type : "eq:componentA1",
      label : "Component_8531",
      subComponents : [ { ... } ],
    },{
      ...
    }]
}
```

Listing 3.1: Multi level form data in JSON

Further challenge is that the options of the type selector on the sub forms, are dependent on the selected type of the parent form (the form which the sub form has been added from). This means that the client has to load asynchronously the values, by sending AJAX request containing the selected type value to server. The task of the server is to perform the query that retrieves the classes defined through restrictions in the ontology. The goal of this functionality is firstly to ensure that only such data is created that conforms to the rules defined in the ontology, secondly the interface is much more usable if not all the components are listed, just the ones that are belong to the selected element. Moreover in this way the validation on the server after submission can be omitted.

Finally important part of problem is the editing of the data. The first step by the editing is the restore the submitted data and send to the client. In the previous chapter we have seen that it is currently solved by defining SPARQL queries that retrieves the form variables. But since the form data is not just a set of key value-pairs but a multi level data object, this approach is not sufficient. An algorithm is required that generates the multi level JSON object from the existing triples iteratively. Furthermore after the arrival of the existing data to the client, an other routine has to reset the state of the form with the appropriate sub forms and certain selector options as well.

This section gave an insight into challenges of the multi level data input. The emphasis lied on that these problem require more complex form functionality and server algorithm. For such cases unfortunately VIVO does not provide any libraries or option for some declarative definition, thus these advanced forms has to be implemented almost from scratch, and poorly documented parts of the source code have to be understood.

3.2 Solution scheme

The aim of the thesis is to develop a web application framework that allows the definition of the data input processes on high-level declarative way. This means that the application development, which normally incorporates coding in HTML, JavaScript and Java, is reduced to the creation of simple descriptor dataset defining the form layout and the RDF dataset that has to be created.

The first task is to design a vocabulary that models the whole data input process. This vocabulary is the scheme of the descriptor dataset. The vocabulary, like an ontology contains the classes representing the entities of the problem domain, and their relationships and attributes.

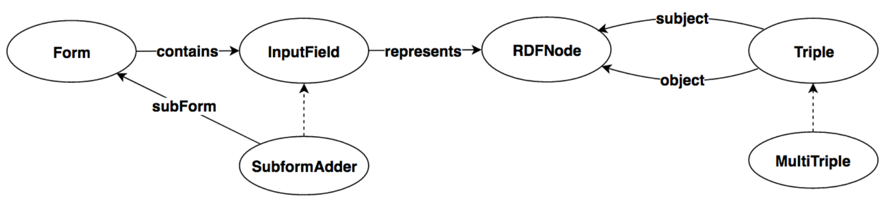

Figure 3.3: Classes and relationships of the vocabulary

Figure 3.3 depicts the main classes of the vocabulary with ellipses, and the relationships with arrows. The attributes and some further subclasses are not displayed here for the sake of readability, but they will be covered in the following chapter. The three classes on the left models the interface, and the other three on the right in turn are for the data model. The purpose of the class *Form* form is to encompass the input fields. To achieve the multi level layout explained in the previous section, the vocabulary contains an input field class *SubformAdder*, to which a sub form can be defined with the *subForm* relationship. This connection between the form and data model is expressed with the *represents* relationship, which establish the basis of the substitution of the values of the input fields into the RDF nodes after the submission. To model the dataset the two main classes are the *Triple* and *RDFNode*, and the triples the appear in the resulting dataset multiple times trough to the hierarchical data structure are represented with the class *MultiTriple*.

The implemented framework takes the descriptor dataset, which is currently set of Java objects, and the processes it in order to generate further descriptor objects of both for the client and the server. These generated descriptor objects are such datasets that contains additional information regarding the functionality, that are not directly defined in the original descriptor.

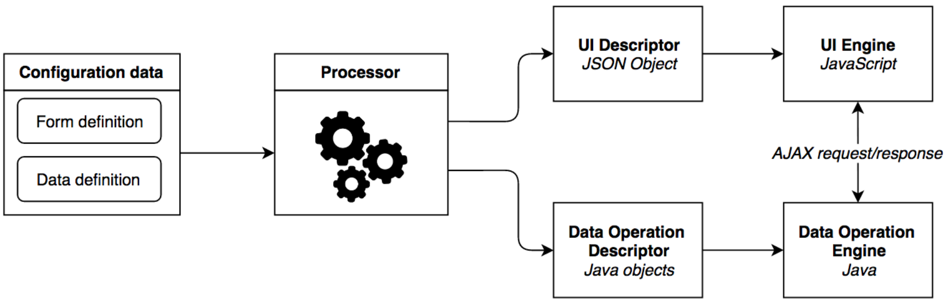

Figure 3.4: Framework functionality outline

The form descriptor data is a JSON object containing the definition of the form elements including the sub forms, and data dependencies between the form elements, so that it can get the options of the sub selectors dynamically. The essence of the routines running on the client is to convert the descriptor JSON object into the form fields, and initiate the necessary AJAX calls.

The server has three main tasks, loading the form data upon the AJAX calls, convert the submitted JSON object to the RDF data, and retrieve the stored data by the edition. The utility of the implemented framework that it exploits that the SPARQL query and the RDF datasets are both RDF graph patterns, and thus they both can be generated automatically from the triple definition part of the descriptor dataset. This feature enable a more compact data definition then what is required in the current VIVO framework.

Chapter 4

Vocabulary for web application domain

4.1 Elements of the vocabulary

4.1.1 Data definitions

To understand the necessity of certain elements of the vocabulary, further details of web the applications have to be explained. In VIVO, the display of the existing and the creation of the new data happens in individual pages. The display is done by so-called profile pages, that show the information about one particular instance. As it was in section 2.3.3, the information is grouped by predicates, and each predicate field contains a link that can call the data input pages. The link contains three parameters, *subjectUri*, *predicateUri* and the *rangeUri*.The *subjectUri* hold the value of the instance on whose profile the link is, the *predicateUri* is for the predicate, with which the new dataset is connected to the subject, and the *rangeUri* is an optional paramater.

Figure 4.4 shows the workflow of a data entry process. On the right profile page of a skeletal inventory can be seen, which lists the added skeletal elements. The profile page is configured that the *rangeUri* parameter holds in the links the URI of the classes of skull and vertebral column respectively. These parameters have to be considered by the form loading because they influences the options of the first selector which let the user add the skele-

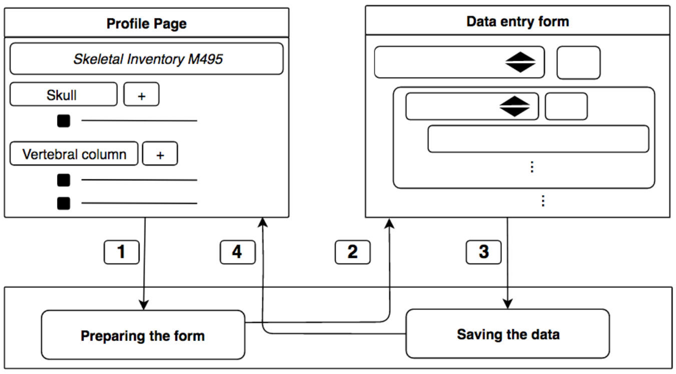

Figure 4.1: Complete workflow of data input process

tal sub sub division. So therefore it necessary to introduce a flag into the vocabulary that sign if a variable is coming with the HTTP request for the form loading, or with the JSON object after the submission.

Figure 4.3 shows all the nodes types and their attributes. Above the *mainInput* boolean flag, there is the variable name, which is required, and the constant value in case. There are three types of variable, the class, resource and literal. The literal variable itself denotes a string value and it has several subclasses for the other primitive types.

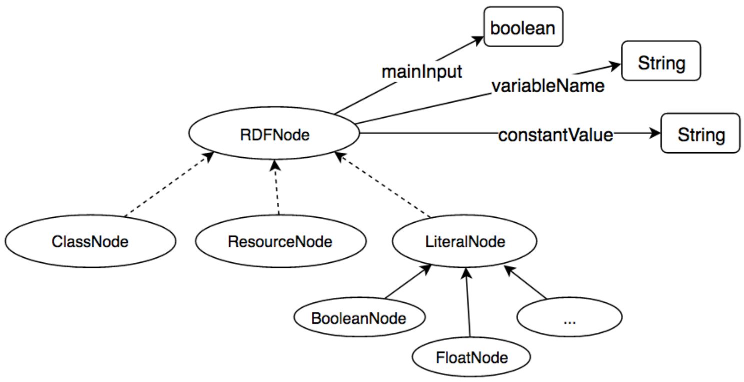

Figure 4.2: Variable types and their attributes

The other, more important is the modeling of the triples in the dataset. The vocabulary for triples has the purpose to express different constraint on the data scheme as well. Above the class *Triple* and *MultiTriple*, which were addressed in the end of the last chapter, there are two types of restriction triples. One for the classes and one for the instances. As we have seen in the description of the OWL ontologies, there are different types of restrictions can be defined. For this reason it should be possible to allow the definition of which restriction is used by the ontology, upon the entry form should operate. This can be expressed by the three boolean, types of the *classRestrictionTriple*. Moreover the greedy boolean flag means that the SPARQL query that queries the ontology has to return not only the result class but their superclasses too. Finally the instance restriction triples as the name indicates, expresses constraints between instances on the form. The examples in the next section will make the usage of the vocabulary more clear.

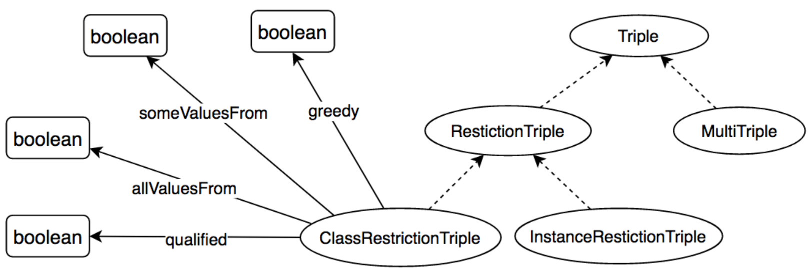

Figure 4.3: Triple types

4.1.2 Form definition

The class *Form* acts like a container for the form elements. There are two main types of form elements, the literal field and the selector. The literal field do not have further subclasseses because its type, is defined through the type of the variable it represents. It would be a sort of over definition. This is the simples case where the form adopts to the data model.

The selector can refer both instances and classes. If it represent and instance, so it let the selection of an existing instance, then it is possible to define an *InstanceViewer*. This feature allows to define a table with several columns. The utility is that the application can show more information about an instance than only the label in the selector field. Each column has a title and a number, and they refer to as well *RDFNodes*, whose values they show in the entries.

The class *SubformAdder* is the subclass of the selector, and has a relationship to the form with the predicate sub form. With this connection it is possible to define the sub form as new form instance. Moreover it has boolean flag, that allow to define a button add all should appear on, which adds all the possible subforms. This feature is useful be the skeletal subdivisions that contains much bones.

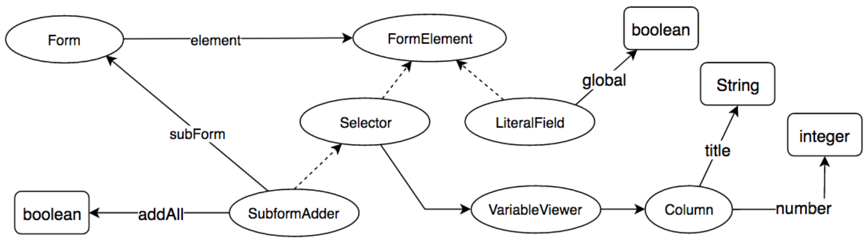

Figure 4.4: Form definition

4.2 Use-cases of the *RDFBones* project

The aim of this section is to show how it is possible to solve different problems with designed vocabulary. It covers the two main tasks of the project, the skeletal inventories and the study design execution. These two examples are sufficient to show the utility of the particular elements in the vocabulary and exemplify how their assembly can lead to a compact definition of complex web application problems.

4.2.1 Skeletal Inventories

Skeletal inventories were already addressed in section **??**. Their goal is to define what kind of skeletal elements are present. In this part the creation of the primary skeletal inventories are discussed in more detail. This use-case addresses some additional challenges of the software, that were not mentioned yet. The explanation starts with the illustration of the implemented interface, to show what problem the high level logic has to define. As it was mentioned in figure 4.4, the entry forms can have inputs, which in this case the input is the class URI of the skeletal subdivision that has to be added with it sub subdivisions and bone organs. Therefore the first element of the entry form is the selector of the sub subdivision.

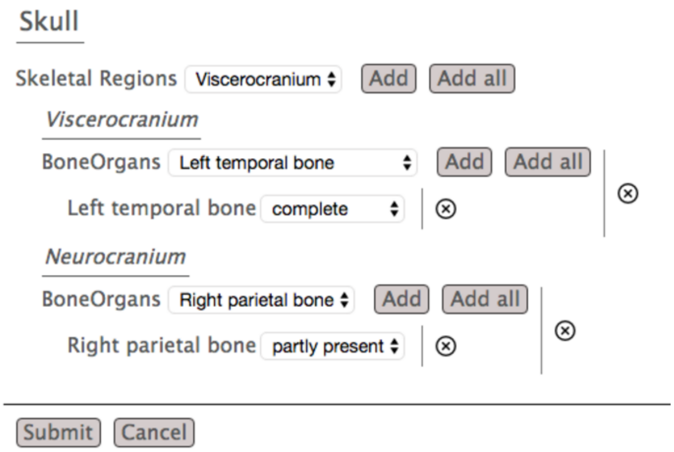

Figure 4.5: Input form for skeletal inventories

Next to the selector the buttons *Add* and *Add all* can be seen, that let the user add the sub forms. Figure 4.5 shows the layout, when two sub sudivision were added, to each of them, a bone organ. The bone organ selector works exactly the same as the sub sudvision selector, but the selector for the completeness state is simple selector, without a sub form.

Figure 4.5 shows the triple scheme representing the skeletal inventories, where the nodes are representing variables. All the rectangle are representing instances. All of them will be newly created instead of the subjectUri, which comes as a main input, and the arrows with double line depicts the multi triples. Important to note that between the variable *boneOrgan* and *boneSegment* there is only a single triple (*fma:regional_part_of*) because in this use-case is simplified version and only entire bone segments will be added.

Above the instances to be created the classes have to be represented in the model too, because they can be the subjects of the class restriction triples. Figure Figure 4.7 is depicts the complete data model of the problem.

For the better readability the predicates are not denoted, but their value can be found in figure Figure 4.6. Each instance (rectangles) is connected to the class variable (ellipses). The red dotted arrows indicates restriction statements. The large three rectangles, that encompass set of triples are the graph, which are connecting to each other by means of multi triples. This is

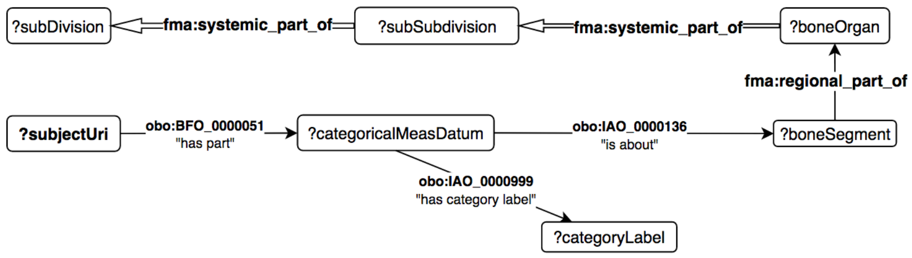

Figure 4.6: Skeletal inventory data triples

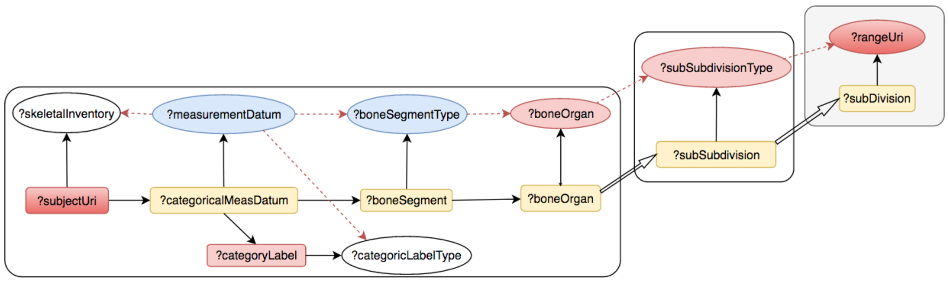

Figure 4.7: Complete data definition

the structure which is followed by the form as well.

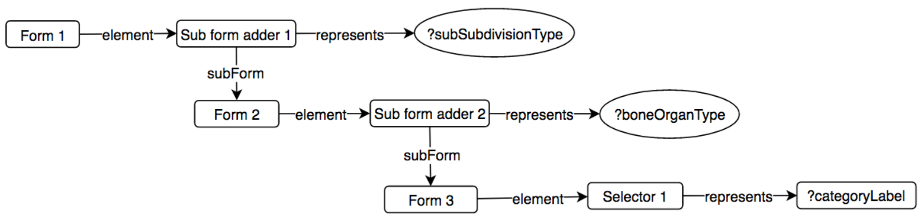

Figure 4.8: Form layout definition

Above the auxililiary rectangles for the graph,the nodes are colored to indicate their role in the process. The information that the colors hold is not defined in the vocabulary but it can be inferred from the whole data and form model. The first rule is that the main input nodes ($subjectUri$ and $rangeUri$) are denoted with red, while the variables appear on the interface are light red. Base on that information is it already possible to determine to which instances it is required to assign an unused URI. Those instance are denoted with yellow. Furthermore there are two classes in the data model that do not appear on the interface, but their values can be evaluated through SPARQL queries. These are denoted with blue. And the classes with without color, do not appear on the final dataset, but they indicates constraint on the existing instances. Finally Figure 4.8 display the configuration data describing the form structure.

4.2.2 Study Design Execution

The entry form for study design execution has as well the hierarchical layout like the one for skeletal inventories, but there are additionally two elements on the main form. The first is the selector from skeletal inventories. It plays a role by the selection of the bone organs as input for the assays. To each assay a set bone segment types is defined in the extensions, that can be can be assigned to them as input. These bone on this form are not created newly but existing ones are selected, that were already added in the frame of the skeletal inventory data input. However there can be a large amount bone segments stored in the system, and thus the search is facilitated by showing only the ones that belong to the preselected skeletal inventory. The second is a global label field, whose value will be the label of all newly created instances.

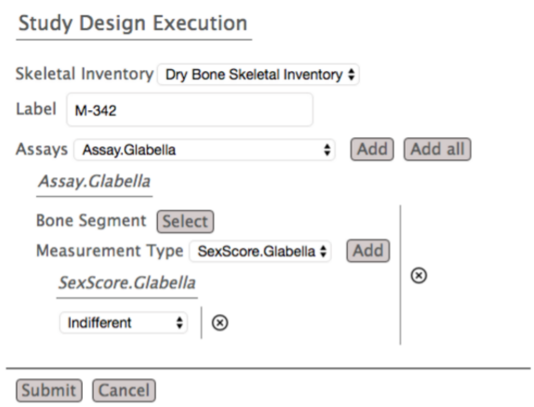

Figure 4.9: Input form for study design execution

Moreover the bone segment selector is not just a HTML selector input field, but a floating window implemented by JavaScript that allows the convenient browsing (Figure 4.10). It has two advantage with respect to the conventional selector. Firstly it allows to display additional information about the instances above their labels, like their types or longer descriptions. Secondly it does not loads the form layout with additional subform for the selected instances, which by large amount assays and measurement datums is an important aspect.

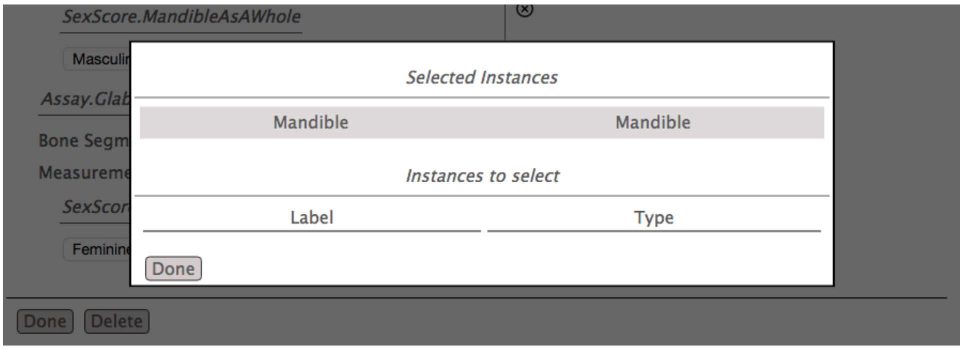

Figure 4.10: Instance selector for existing bone segment

On Figure 4.10 can be seen that the there are two section, one for the selected instances, and one for the instances to select.

The complete data structure of the form can be seen on the following image.

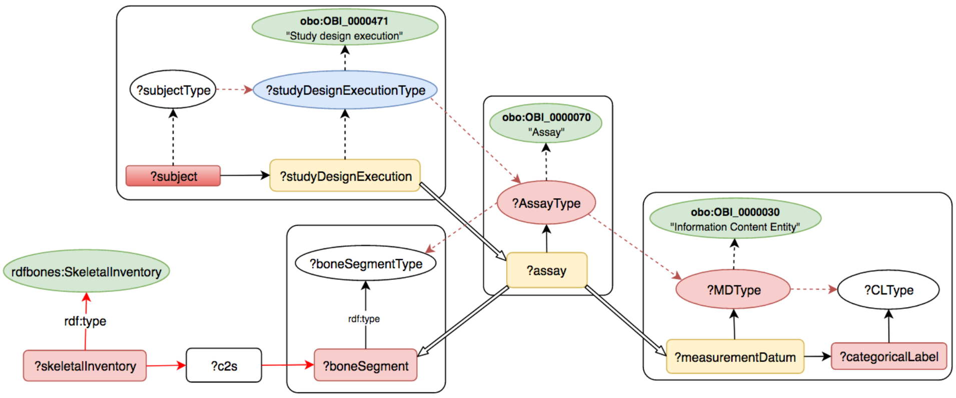

Figure 4.11: Complete data model

- Validation - cyclic graph - multiple dependencies
- subgraph subform dependencies

- Descriptor for data and processing

Chapter 5

Framework functionality

The aim of this chapter is to present the main mechanisms of the implemented software framework, which is capable of operating on high-level configuration data. Section 5.1 contains a more abstract description of the functionality, while section 5.2 goes into the implementation details both of the server and client side programming, including how the framework can be integrated into the applied web application. In both sections the explanation refers to the examples discussed in the previous chapter.

5.1 Main software modules and tasks

In Figure **??** we have seen the main work flow and components of the framework. Figure 5.1 is in turn a more detailed depiction of the software modules and processes of the application.

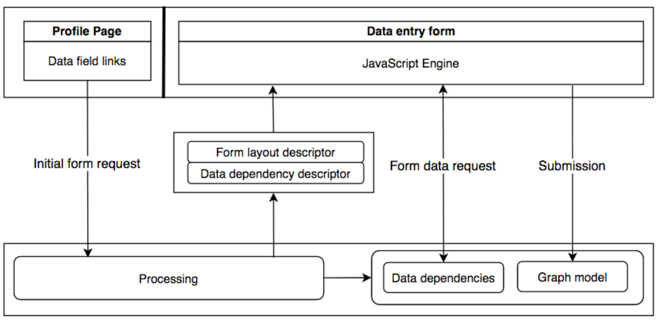

Figure 5.1: More detailed scheme

This section is divided into three subsections. First is the part (section 5.1.1) discusses the processing of the configuration data and generation of the functionality descriptor objects for the client and the server. The second part (section 5.1.2) is about the client functionality including the asynchronous communication with the server. Finally the third (section 5.1.3) part covers the process after the submission, namely how the RDF data is generated based on the data coming from the client, as well as how the existing data is retrieved from the triples store.

5.1.1 Validation

The processor algorithm has four main tasks to solve. The validation of the configuration data, generation of the form descriptor JSON object, and the Java object for data dependencies and for the graph model.

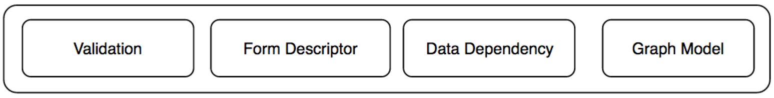

Figure 5.2: Processor tasks

The input of the algorithm is the set of triples describing the data model with its constraints, and form model, which refers to the nodes in the triple set. The first task to do is the validation because the descriptors are not supposed to be generated based on incorrect configuration data. The validator process has three scope of the checking, the nodes, the graph and the form.

Figure 5.3 depicts an example data model and illustrates the cases of valid and invalid nodes (Figure **??** contains the meaning of the shapes and colors). The explanation starts with the discussion of the form input nodes. Node *2* is valid because it is possible to generate a SPARQL query that retrieves the possible values of it. The query contains one triple which ask the subclasses of the constant class. Furthermore node *4* is valid as well, because there is path to it from a valid class node, therefore for there is again a SPARQL query for its values. However the variable *3* is not valid, because it does not contribute to any triple in path with valid input node or constant. Here it is important to note that the path cannot come from the instance to the class, just the other way around. So the path 2 –> 1 –> 4 counts in the processor routine, but 2 –> 5 –> 6 –> 3 does not.

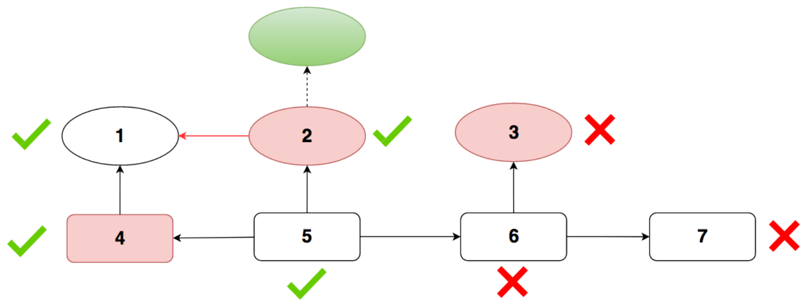

Figure 5.3: Valid and invalid nodes

The next task regarding the nodes is to check if each has a value by the RDF triple creation. The instances coming from the interface are automatically valid, because their URI is an input value. But the ones that have to generated newly and get as value a new unused URI from the triple store, must constribute to triple as subject, where the predicate is *rdf:type* and the object is a valid class. For this reason the node *5* is valid, since its type class is valid, but node *6* is not. Moreover node *7* is not valid as well, since it does not have any type class defined in the data model. Finally regarding the literals the validation is the simplest, either they appear on the form or have constant value, otherwise they are invalid.

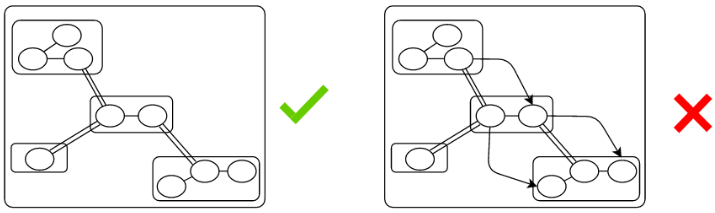

Figure 5.4: Valid and invalid graph

Above the nodes important itself the whole graph built by the triples have to be investigated. In the previous chapter the triple type *MultiTriple* were introduced. The rule regarding this type of triple that it divides the graph into subgraphs, and the subgraphs can connect to each other only be these triples. Figure 5.4 illustrates the valid and invalid graph arrangements.

The reason is that only to this type of scheme can the JSON object of the created by the data input process be mapped.

5.1.2 Dependencies and form functionality

The set of triples describing the data model builds a graph where the various input nodes are connected to each other. Further task of the processor algorithm to determine the subgraphs that define the SPARQL queries for the node appear on the form. The elements on the form has a specified order, and the rule that have to be considered during the processing, is that a variable can be dependent only of the main input variables or such variables that are before it on the form. The reason is that the dependency is practically SPARQL query with one output and with one or more inputs, and input node have to available by the execution of the query.

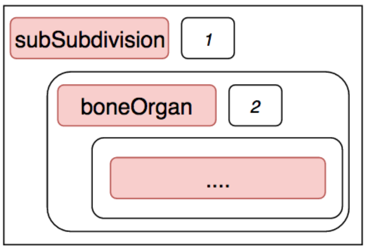

Figure 5.5: Form element order

Figure 5.5 shows the simplified form layout of the skeletal inventories and emphasizes the order of the elements with the number. The idea is that values of the selector for the node *subSubdivision* can be only dependent on the main input nodes because it is the first node. The *boneOrgan* in turn can be dependent on the *subSubDivision* too because its value has to be set before the options of the subform is loaded, thus it can be substituted into a SPARQL query.

Figure 5.6 depicts the scheme of the data constraints of the skeletal inventory, based on which the dependency retriever algorithm can be exemplified. As it was mentioned the task is to get one or more path from the variable of to an other node whose value is available for it. As the form's first element refers to the node *subDivision* its dependency has to be evaluted first. Since the node *subSubDivision* contributes to two restriction triples in the data

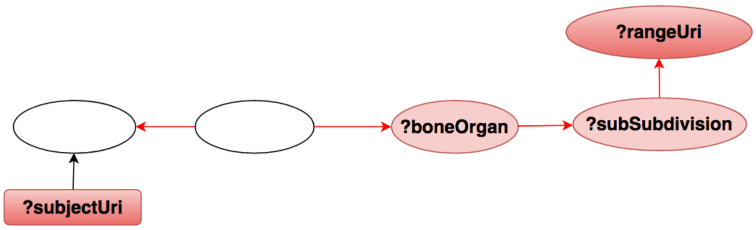

Figure 5.6: Skeletal inventory data constraints

scheme it is necessary to check the both paths. Since this node cannot be dependent on any other form node, it is dependent on the two main input nodes (*subjectUri* and *rangeUri*). While the by dependency for the node *boneOrgan* the *rangeUri* does not count, since the algorithm terminates already by the *subSubdivison* variable. Figure 5.7 shows the results subgraphs of the algrithm, where the output is depicted with green and the inputs with red and light red respectively.

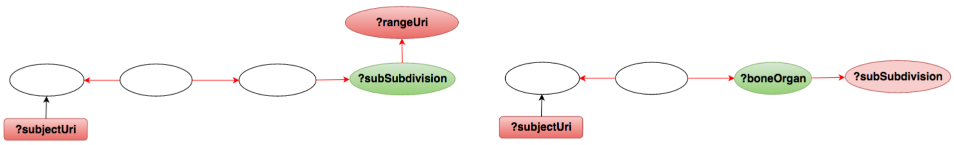

Figure 5.7: Form dependency subgraphs

This result of the processor concerns both the descriptor of the client, and the server. On the server the depencies are stored a classes that have the necessary fields, for the inputs and output and for the triples. Then these initialized dependency descriptor class objects are stored in a data map, where the key is the variable name of the output node. This map is a field of the form configuration class, thus if the client asks for the appropriate form variable through AJAX these SPARQL query assemble by the triples for can be executed with the incoming values.

For the client the dependency description is just the an assignment of an array, which contains the input node of the dependency, to the form node. In the case of the *subSubdivision* it is an empty array, because it is not dependent on any form element, but for example the *boneOrgan* in the use-case for the study design execution the is dependent from the *assayType* and the skeletal inventory. The task of the form by loading new sub form is to check this array and get the variables required values from the form. This mechanism will be discussed in more detail in section **??**.

Above the dependency the descriptor the JavaScript framework obtains the descriptor data file for the form elements upon which it can generate the form. The mechanism of the form description retrieval a quite simple because it is practically the convestion of a Java object into a JSON object. The fields of the Java object appear in the JSON objects as key. In

```
public JSONObject getDescriptor(){
  JSONObject object = new JSONObject();
  object.put("title", this.title)
  ...
  return object;
}
```

Listing 5.1: Java to JSON

Where the *this.title* has contains the value for the title of the form object in the descriptor Java object. The same way if the Java class has list field, like the form has list of form elements, then getDescriptor() routines of each element is called and inserted into a JSON array. Moreover if the form element is sub form adder then it has a field *subForm*. This comes as well of course into the descriptor, and this is the way how the multi level JSON is created.

```
  object.put("subForm", this.subForm.getDescriptor());
```

Listing 5.2: Subform descriptor

Figure 5.8 illustrates the generated JSON object for a form with the data dependencies too. The task of the JavaScript routine to interpret this configuration data and generate the form and subform upon user action.

5.1.3 Graph model generation

The previous section outlined how the set of Java objects are converted into the form decriptor JSON object, and into Java objects describing the variable dependencies for the AJAX requests. Further task of the framework

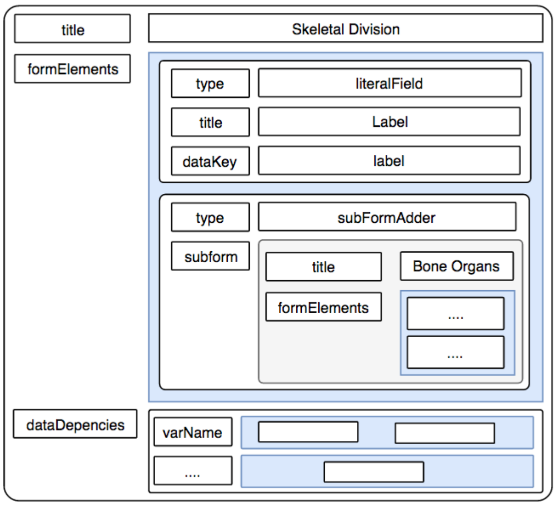

Figure 5.8: Form descriptor JSON

is to receive the submitted multi level JSON object coming from the client and generate the appropriate set of RDF triples. So in order to prepare the server for the reception of the form data, the same object structure have to be generated, which is coming from the form. We have seen in the previous chapters, that the form data object has the same scheme as the form has, and form follows the scheme defined by the multi triples in the triple set. Therefore the task of the last part of the processor algorithm is to decompose the set of triples by multi triples into graphs. The graph structure is represented in the server by a Java class called *Graph*.

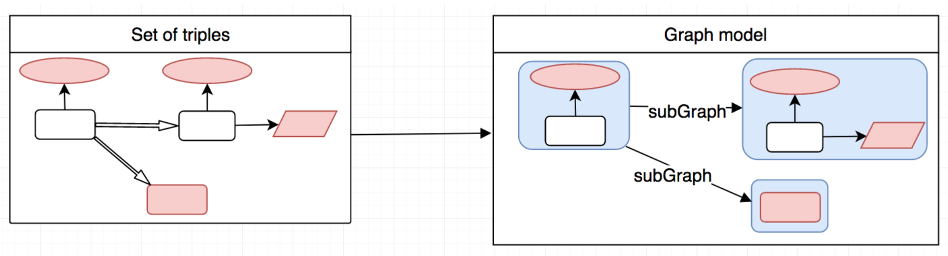

Figure 5.9: Conversion from triples into graph model

The decomposition starts by the initial RDF node, which defines the main graph. The class *Graph* has a Map<String, Graph> field where the subgraphs are stored. The keys of the map are equivalent to the keys of the keys of the arrays in the incoming JSON object. The other keys of the JSON are the variables of the corresponding graph. The following code snippet shows the basics of the RDF data generation from multi level JSON object.

```
saveData(JSONObject formData){
  this.save(formData);
  for(String keys : this.subGraphs.keys()){
    JSONArray array = formData.get(key);
    Graph subGraph = this.subGraphs.get(key);
    subGraph.saveArray(array)
  }
}
```

Listing 5.3: Subform descriptor

The save routine creates the RDF triples of the graph based on the data fields of the JSON object. The a loop iterates through the subgraphs of the graph and gets the arrays with the same key from the JSON and passes it to the corresponding subgraphs, that perform the same algorithm as many times, as many elements of the input array have. This is the way how the multi dimensional JSON is processed by the same structure of graph model on the server.

FiguregraphProcess illustrates the process of the JSON-RDF conversion by means of graph model. The advantage of this graph model that it can be applied the same way for the data retriaval, where the graph performs the SPARQL queries based on its triples, and generates the arrays of objects from the result table.

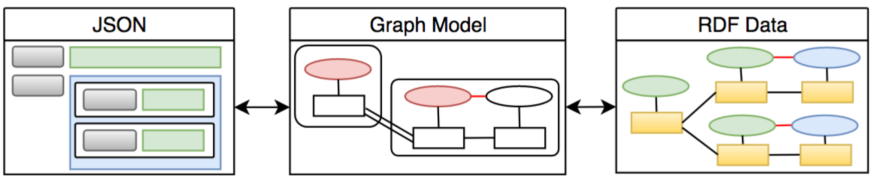

Figure 5.10: Graph decomposition

5.2 Implementation

The description of the implementation starts with the client side because it is more independent from the other parts, and it sufficient to understand properly the functionality of the server. While in the last subsection it is discussed how can the developed framework integrated into the VIVO web application.

5.2.1 Client side

This subsection presents the basics of the JavaScript implementation that realizes the dynamic form generation and event handling based on JSON form configuration data. The first part covers the creation of a form itself, and how the data is set to form data object, while the second part is about how the form enables multi dimensional data input by means of sub form adders.

Form loading

In contrast to Java, JavaScript codes are not necessarily built up in an object-oriented manner. On pages where the elements are statically defined in HTML, it is sufficient to assign event handler routines to them. However in our case none of the elements of the page is coded into HTML, but everything is dynamic and thus added by JavaScript. In the implementation JavaScript classes are applied, whose input is the descriptor object, based on which they generate the corresponding data input fields, and handles the data entered through them by the user. In this section the functionality of the two main classes, the *Form* and *Formelement* is discussed.

Figure 5.11 illustrate the structure of the two main classes. The most fundamental difference betwen these JavaScript classes to Java classes, that they do not contain only fields and routines (or methods) but UI elements as well. The UI elements can represent and HTML tag, and can be added or removed any time by the routines. Each class of the implemented JavaScript library has a defined set of UI elements.

The form generation processes starts with the initialization of a *Form* object, where the constructor (like in Java) gets the descriptor JSON object coming from the server. As it was described in the previous chapter

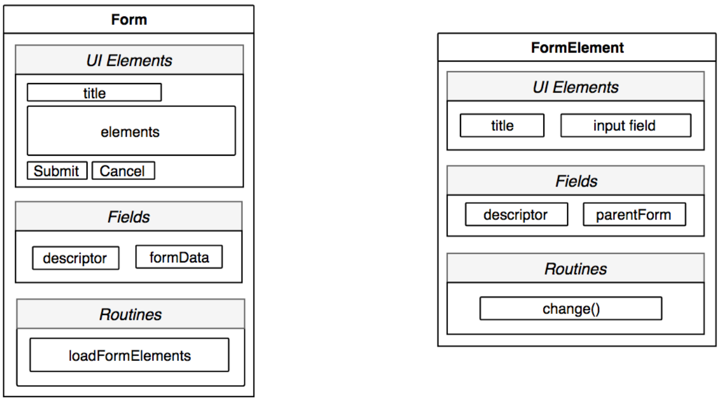

Figure 5.11: Form and form element classes

the descriptor contains a list of the form element descriptor objects. The *loadFormElements()* routine iterates through on this array, and initiates the form element objects.

```
var formData = new Object()
for(var i = 0, i < formElements.length; i++){
  switch(formElements[i].type){
    case "stringField":
      var element = new StringField(formElements[i], formData)
      break;
    case "selector" : ...
  }
  this.elements.append(element.container)
}
```

Listing 5.4: Form generation based on configuration data

Each form element type is represented as subclass of the *formElement* class. They all have a container UI field, that contains their title and input field HTML element. This container field is added to the *elements* field of the *Form* object.

Listing 5.5 show a small cut from the code of the StringField, which is the subclass of the *FormElement* class. The field *inputField* is the HTML <input/> tag, and if its value changes then the *editHandler* routine is called. The editHandler is the function that realizes the dynamic form data creation, by setting the value of the input field into the form data object with the key defined in its the descriptor. The key is stored in the *dataKey* field of the descriptor, which is the variable name of the RDFNode the input element represents.

```
class StringField extends FormElement{
  constructor(descriptor, formData){
    super(descriptor, formData)
    this.inputField = $("<input/>").type("text").change(this.
        editHandler)
    ...
  }
  editHandler(){
    this.formData[this.descriptor.dataKey]=this.inputField.val()
  }
}
```

Listing 5.5: Form element

This is the basic mechanism of how object-oriented JavaScript can be employed to generate forms, and put the entered values in to JSON object based on configuration data.

Sub forms

The previous section explained how the form algorithm creates the JSON object of the form data. This section extends the explanation of how it is possible to add the multi level data by sub form adders. To this two new JavaScript class functionality is outline, the *SubFormAdder* and the *SubForm*. The former is the subclass of the *FormElement* and the latter of the *Form* class.

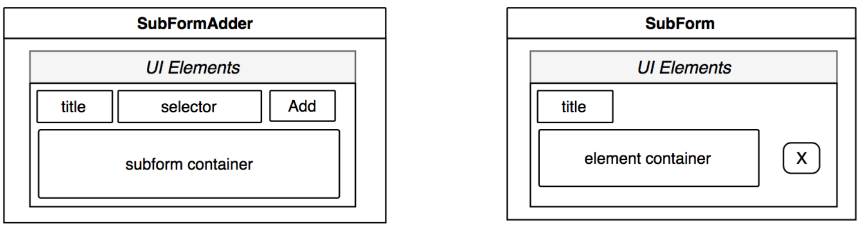

Figure 5.12: SubForm and sub form adder

Figure **??** depicts the UI elements of the two classes. The routines and fields are inherited from the parent classes. The class *SubFormAdder* has a button, which lets the user add new sub forms, which are appended into the sub form container. The class *SubForm* has additionally to the parent class a delete button for the cases if the user wants to delete the added dataset.

Listing 5.10 shows the relevant part of the code in the class *SubFormAdder*. The essence of the class is that the constructor initiates an array (with "[]") in the form data object, to which the sub form data object will be added dynamically upon the click events. So if the user clicks the add button, then new object is initialized (*subFormDataObject*), which will be the data object of the sub form. Important to note that this object will contain value of the selected option of the sub form adder with the key defined in the *dataKey* field of the descriptor. After the initialization of the object, it is pushed to the array, and the new *SubFormAdder* instance is created, whose container is appended to the sub form container of the sub form adder.

```
class SubformAdder {

  constructor(descriptor, formData){
    this.addButton = $("<div/>").text("Add").click(this.add)
    this.subFormDescriptor = this.descriptor.subForm
    this.formData[this.descriptor.predicate] = []
  }

  add(){
    var subformDataObject = new Object()
    subFormDataObject[this.descriptor.dataKey] = this.selector.
        val()
    this.formData[this.descriptor.predicate].push(
        subFormDataObject)
    this.subFormContainer.append(
        new SubForm(this, this.subFormDescriptor,
            subFormDataObject).container)
  }
}
```

Listing 5.6: Sub form adder routine

The class *SubForm* works almost the same way as its parent, but with the difference that it checks if there is such selector among its elements, whose data has to be loaded dynamically through AJAX, because its value is dependent on one or more previously set elements of the form.

5.2.2 Server side

This section covers the functionality of the server. On the client side the implemented JavaScript classes were just included into the form template file, but the framework integration to the server is a more complicated issue, therefore the first subsection is dedicated to it. Furthermore the implementation of the form data calls with the variable dependency calls are discussed here, as well as the routines how the graph model can save, edit and retrieve the or subsets of the form data.

VIVO integration

As it was mention in the previous chapter the process of the form loading start with the profile pages of VIVO. In VIVO the entry form loading is initiated by the property fields. Normally the programming in VIVO happens through so called generator classes. The task of the generator classes is to define the dataset that have to be created in the form, and reference the Freemarker template file for entry form.

```
<someProperty>  vivo:customEntryFormAnnot
                     "rdfbones.DrawingAConclusionGenerator.java"
```

Listing 5.7: Custom entry form definition in VIVO

The approach implemented in the frame does not replace this structure, but makes it simpler. In our cases we use as well generator classes, but the definition of the data happens through the intialization of an instance of the class *FormConfiguration* instance (listing **??**).

Figure 5.13 shows the process of loading an entry form in VIVO. The first step is to find the generator class based on the value of the *predicateUri* parameter of the initial request. If the class has been found the processor algorithm is executed, and the necessary JSON and Java object are generated. Afterwards the server saves the form configuration object its cache with a key, which is called in VIVO *editKey*. The box in the middle of the image shows, that the response web page includes the JS library (simplified notation *framework.js*), and the value of the *edit*. This is the value will be sent in each AJAX request to the server, and based on this the can the server find the form configuration instance that returns the JSON object for the client.

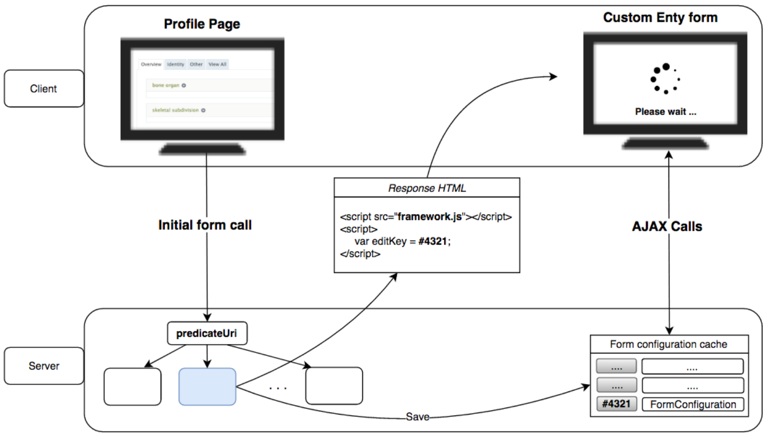

Figure 5.13: Form loading process

Figure 5.14 show the UML class diagram *FormConfiguration* class. The fields of the classes were already discussed in the previous chapter, but here it is important to note that each AJAX request is server by the method *serveRequest()*, where both the input and the output is JSON object.

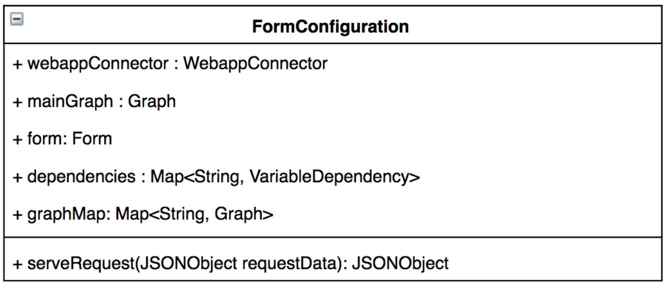

Figure 5.14: UML class diagram for FormConfiguration

Listing 5.11 shows the main scheme of *serveRequest()* routine. Each request coming from the client has the key *task*, which defines what data JSON object has to be served for the client.

```
SONObject serveRequest(JSONObject requestData){
    switch (requestData.get("task")) {
      case "formSubmission":  ... break;
```

```
        case "formDescriptor":  ... break;
        case "editData":     ... break;
        case "deleteAll":    ... break;
    }
}
```

Listing 5.8: AJAX request server routine

Finally a really important part of the integration is the access to the used triple store. For this purpose an interface, called *WebappConnector* is defined. It defines the functions that allows the querying and manipulation of the triples.

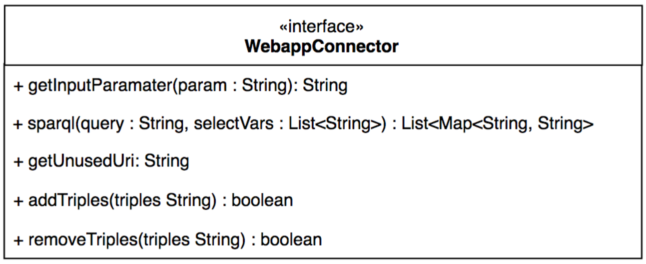

Figure 5.15: UML class diagram for WebappConnector

Form data loading

Important part of the framework functionality is the loading of the dependent data to the forms. This is defined through the keyword *formData* in the task parameter. Moreover it has parameter, the *variableToGet*. Based on this variable, the form configuration finds the variable dependency instance and passes the incoming JSON object to its method *getData()*.

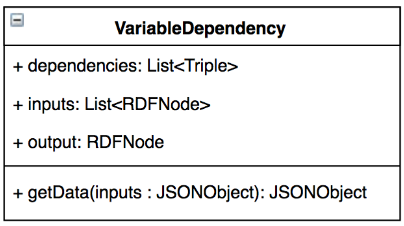

Figure 5.16: UML class diagram for VariableDependency

```
JSONObject serveRequest(JSONObject requestData){
switch (requestData.get("task")) {
  ...
  case "formData":
    return this.dependencies.get(requestData.get("varToGet")).
        getData(requestData)
  ...
}
```

Listing 5.9: Loading form data from FormConfiguration

We have seen that the dependency is set of triples, which build a graph. This graph can be built by class restriction triples, which constitutes to the generated SPARQL query not with one triple but with three triples.

```
SELECT ?outputVar ?label
WHERE {
  ?inputVar       rdfs:subClassOf      ?restriction .
  ?restriction    owl:onProperty       fma:systemic_part_of .
  ?restriction    owl:someValuesFrom   ?outputVar .
  ?outputVar      rdfs:label           ?label .
  FILTER (?inputVar = fma:5058)
}
```

Listing 5.10: SPARQL query generated by class restriction triple

Saving, editing and retrieval of RDF Data

```
JSONObject serveRequest(JSONObject requestData){
switch (requestData.get("task")) {
...
case "saveData":
return this.mainGraph.saveData(requestData.get("formData"))
case "retrievedData":
return this.mainGraph.getData(requestData.get("startUri"))
case "editData":
return this.graphMap.get(requestData.get("varToEdit")))
.editData(requestData)
...
}
```

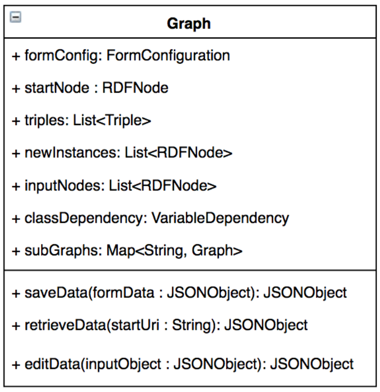

Figure 5.17: UML class diagram for Graph

Listing 5.11: Loading form data from FormConfiguration

Appendix A

Glossary

Just comment `\input{AppendixA-Glossary.tex}` in Masterthesis.tex if you don't need it!

Symbols

$	US. dollars.

A

A	Meaning of A.

B

C

D

E

F

G

H

I

J

M

N

P

Q

R

S

T

U

V

W

X

Appendix B

Appendix

B.1 Something you need in the appendix

Just comment `\input{AppendixB.tex}` in Masterthesis.tex if you don't need it!

Erklaerung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum

Unterschrift

Bibliography

[1] Anita Bandrowski, Ryan Brinkman, Mathias Brochhausen, Matthew H. Brush, Bill Bug, Marcus C. Chibucos, Kevin Clancy, Mélanie Courtot, Dirk Derom, Michel Dumontier, Liju Fan, Jennifer Fostel, Gilberto Fragoso, Frank Gibson, Alejandra Gonzalez-Beltran, Melissa A. Haendel, Yongqun He, Mervi Heiskanen, Tina Hernandez-Boussard, Mark Jensen, Yu Lin, Allyson L. Lister, Phillip Lord, James Malone, Elisabetta Manduchi, Monnie McGee, Norman Morrison, James A. Overton, Helen Parkinson, Bjoern Peters, Philippe Rocca-Serra, Alan Ruttenberg, Susanna-Assunta Sansone, Richard H. Scheuermann, Daniel Schober, Barry Smith, Larisa N. Soldatova, Christian J. Stoeckert, Jr., Chris F. Taylor, Carlo Torniai, Jessica A. Turner, Randi Vita, Patricia L. Whetzel, and Jie Zheng. The ontology for biomedical investigations. *PLOS ONE*, 11, 11 2015.

[2] Dan Brickley and Ramanathan Guha. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[3] Mike Dean and Guus Schreiber. OWL web ontology language reference. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[4] Felix Engel, Stefan Schlager, and Ursula Witwer-Backofen. An infrastructure for digital standardisation in physical anthropology. 11, 04 2016.

[5] Markus Lanthaler, David Wood, and Richard Cyganiak. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[6] Cornelius Rosse and José L.V. Mejino Jr. A reference ontology for biomedical informatics: the foundational model of anatomy. *Journal of Biomedical Informatics*, 36(6):478 – 500, 2003. Unified Medical Language System.