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.

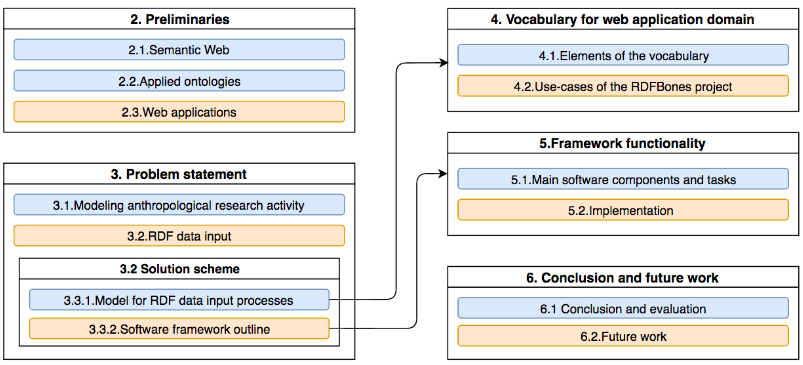

Figure 1.1: Structure of the chapters

Chapter 2

Preliminaries

2.1 Semantic Web

2.1.1 Resource Description Framework

The Resource Description Framework (RDF) is a metadata data model used for representing information on the web. The data in RDF is organized into triples, where each triple consist of a subject, predicate and object. The set of RDF triples constitutes a directed graph, which is referred as RDF graph. The nodes of the graph are the subject and the object, while the edges are the predicates.

The three most important type of nodes are the instances, classes and literals. An instance represent concrete entities like person, institute or even abstract concepts. Classes are general concepts, to which the instances can belong, while literals represent data values assigned to instances. The following Listing shows some example triples that illustrates the basics of information representation by means of triples.

```
Bob        instanceOf      Student   .
Math       instanceOf      Course  .
Bob        attends         Math  .
Bob        avgGrade        1.73  .
```

Listing 2.1: Information in triples

In the example *Bob* and *Math* are instances, and they represent a concrete existing person and Math course of some university. The *Course* and *Student* are classes and the value 1.73 is literal. The values of the predicates are called properties. There are three main types of it, the one which connects instances to class (*instanceOf*), one the expresses relationship between instances (*attends*), and one which assigns a literal value to an instance (*avgGrade*).

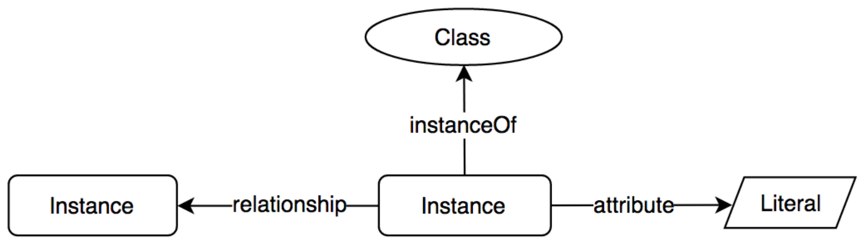

Figure 2.1: RDF Data

Figure 2.1 illustrates the three main type of nodes and properties in RDF graphs. In the following this type of notation will be used on the figures, namely ellipse for the classes, rectangle for the instances and rhombus for the literals.

The example gave just an insight into triple based modeling, but of course the nodes *Bob* and *Student* are not valid RDF nodes. The instances and classes are called resource, and each of them in an RDF graph has to have Internationalized Resource Identifier(IRI). The IRIs have to be unique, thus if someone want if someone wants to represent an information in RDF, then it is necessary to choose an own namespace. For example if the choosen namespace is *<http://myDomain.com#>*, then the IRI of the class *Student* may be *<http://myDomain.com#Student>*.

RDF data does not contain only IRIs from own namespaces, but there are vocabularies that offer a set a built in IRIs. The three most important vocabulary is the RDF, RDF Schema (RDFS), XML Schema (XMLS). RDF and RDFS offer classes and properties, while XMLS contains the datatype IRIs for literal values, and each them of these vocabularies have its own namespace.

RDF data can be stored in text documents. These document can have various serialization format. In RDF documents to improve readability, the namespaces are abbreviated with prefixes, and IRIs are represented with a prefix:suffix syntax. The meaning of this syntax is the concatenation of the prefix and the suffix. The following Listing shows a valid RDF document containing the triples of the example.

```
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns\#>
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
@prefix xsd:     <http://www.w3.org/2001/XMLSchema\#>
@prefix domain:  <http://myDomain.com\#>

domain:1       rdf:type             domain:Student   ;
               rdfs:label           "Bob"^^xsd:string    .
domain:2       rdf:type             domain:Course    ;
               rdfs:label           "Math"^^xsd:string   .
domain:1       domain:attends       domain:2   ;
               domain:avgGrade      "1.72"^^xsd:float  .
```

Listing 2.2: RDF data in N3 serialization format

The *";"* in N3 is used to divide predicate-object pairs that belong to the same subject. With this approach the subject do not have to written as many times as many triples it participates in.

The property *rdf:type* expresses the *instanceOf* relationship, while *rdfs:label* is the most widely used property to assign label to instances. By the literals it can be seen, that they are surrounded with quotation marks and extended with the type notation from the XMLS vocabulary.

2.1.2 RDF and RDFS Vocabularies

In the previous section the RDF and RDFS vocabularies were already mention, but this section provides much more detailed information about their usage and utility.

RDF and RDFS offers classes and properties that allows to express the rules of the RDF data. First of all it has to be defined, what IRIs represent classes and what properties. For this purpose there are two classes the *rdfs:Class* and *rdf:Property*. To define own properties and classes, just like by the instance the *rdf:type* property is used.

```
domain:Student           rdf:type          rdfs:Class  .
domain:Course            rdf:type          rdfs:Class   .
```

Listing 2.3: Class and property definition

In this way it is defined that in our dataset the instances will represent students and courses. The definition in the case of the properties is a bit more complex, because it has to be defined too, what type of subject and object they can connect. For this definition there are two properties in RDFS, the the *rdfs:domain* and *rdfs:range* respectively.

```
domain:attends        rdf:type        rdf:Property ;
                      rdfs:domain     domain:Student ;
                      rdfs:range      domain:Course .
domain:avgGrade       rdf:type        rdf:Property .
                      rdfs:domain     domain:Student .
                      rdfs:range      xsd:float .
```

Listing 2.4: Property definition I.

The previous two Listings contains the RDF triples defining an own vocabulary. The vocabulary is called as well ontology and purpose is to define the scheme of the data.

Figure 2.2 illustrates the essence of the vocabulary definition in general way. It can be seen that property (Property1) and the classes (ellipses without notation) are the instances of the RDF and RDFS classes, and they connected with range and domain predicates. Moreover in the triple where the predicate is *Property1*, the subject is the instance of the domain class, and object is the range class of the property. In the case of properties where the range is an XMLS type, the rule is the same, just the object is not an instance but literal.

Furthermore there is an important modeling concept, the sub class. To express this type of relationship there is property in RDFS, the *rdfs:subClassOf*. So the following triples adds new information to the model.

```
domain:Person         rdf:type        rdfs:Class .
domain:Student        rdfs:subClassOf domain:Person .
```

Listing 2.5: Sub class definition

In order to provide a wider image from the RDF and RDFS vocabulary Figure 2.3 show the further classes and properties.

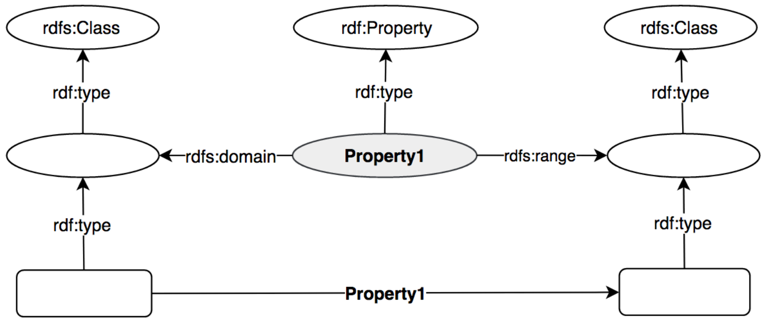

Figure 2.2: RDF/RDFS Vocabulary

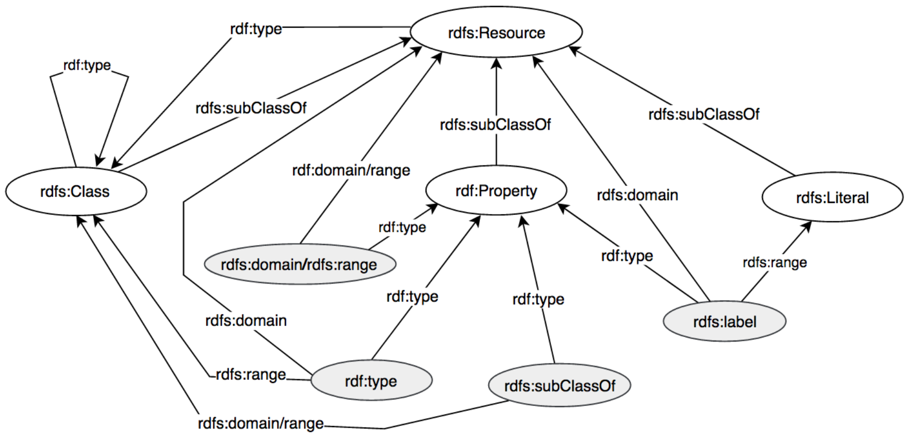

Figure 2.3: Class hierarchy of the RDF/RDFS vocabulary

On the left of Figure 2.3 the class *rdfs:Class* is depicted. It is class since it is the instance of itself. The same way *rdfs:Resource* is class. Furthermore *rdfs:Class* is the sub class of the *rdfs:Resource*, from which it follows that an instance of any class is resource too.

The four grey ellipses (one contains two properties) represent the five most relevant properties in the RDF/RDFS vocabulary. Each of them are the instances of the class *rdf:Property*, and for each the domain and the range classes are depicted.

2.1.3 OWL

Ontology Web Language(OWL) is an extension of the RDFS vocabulary [3]. OWL offers several useful classes and properties, that allows more fine grained modeling. OWL has its own namespace, which is normally denoted with the *owl* prefix.

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
```

Listing 2.6: OWL namespace

Figure 2.4 shows a subset of the OWL vocabulary, that is relevant for the thesis.

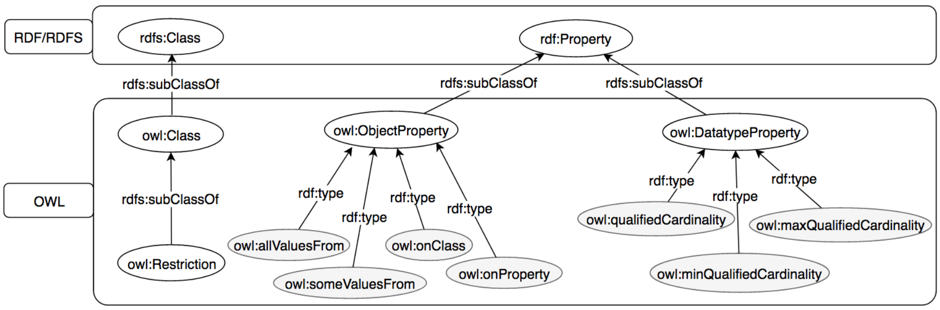

Figure 2.4: A subset of OWL vocabulary

OWL extends the *rdf:Property* with two sub classes. The *owl:ObjectProperty* is represent the properties whose range is a class, while the *owl:DataTypeProperty* is for those properties where the range is XMLS type. Instead of depicting the range and domain of the particular properties in OWL, the following figure show the scheme of the two types of restrictions used in our project.

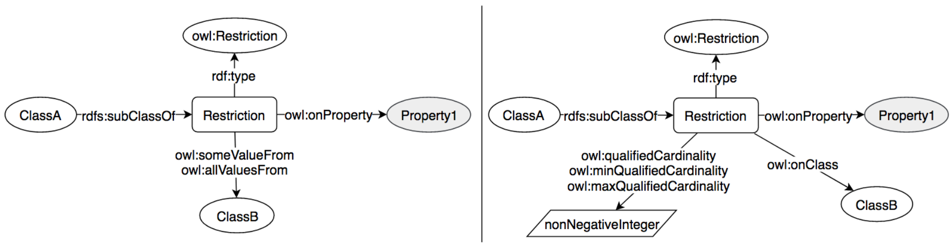

Figure 2.5: Scheme of the restrictions

On the left side the scheme of the value restriction, and on the left side the scheme of the qualified cardinality restriction is depicted. The multiple predicates on the figure shows the possible properties that can be applied for the triple. Important by both cases that the central restriction instance (rectangle) is not an RDF resource but a blank node. Blank nodes are such RDF nodes that do not have a IRI. Later in this section example will be provided about how they can be defined.

The idea is that the restriction node encompasses the triples that describing the rule of the domain. The class on which the restriction is applied is the subclass of the restriction blank node, while the property is connected with the *owl:onProperty* property. By the value restriction the *owl:someValuesFrom*/*owl:allValuesFrom* properties define the instances of *ClassA* can constitute a triple with property *Property1* at least with one instance/and only with instances of *ClassB*. By the qualified cardinality restrictions there is an additional triple that asserts the how many instances have to present at least/exactly/at most, and the second class is assigned to the restriction node with the *owl:onClass* property.

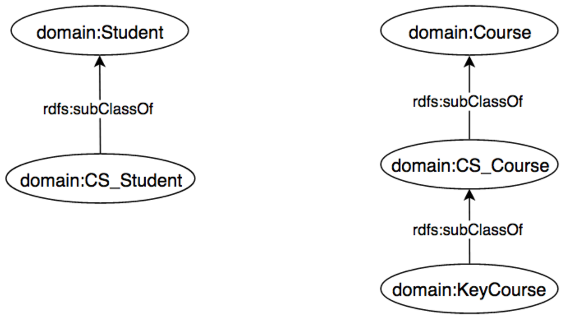

Figure 2.6: Ontology extension

In order to illustrate the utility of restrictions, the ontology from the previous section was extended with three subclasses (Figure 2.6). The abbreviations *CS* stands for computer science. The following two restriction asserts that a computer science student is allowed to attend only computer science course, and has to take at least two key courses.

```
domain:CS_Student rdfs:subClassOf  [
  rdf:type          owl:Restriction  ;
  owl:onProperty    domain:attends  ;
  owl:allValuesFrom domain:CS_Course  .
]
```

```
domain:CS_Student rdfs:subClassOf [
rdf:type                    owl:Restriction ;
owl:onProperty              domain:attends ;
owl:onClass                 domain:KeyCourse;
owl:minQualifiedCardinality "2"^^xsd:nonnegativeInteger .
] .
```

Listing 2.7: Restrictions defined as blank nodes

2.1.4 SPARQL

RDF data is stored most commonly in triplestores. A triplestore is software for storage and retrieval of RDF triples. The retrieval like by other kind of databases happens via queries, that are formulated in a particular query language. The query language for RDF is called SPARQL.

A basic SPARQL query consists of two main parts. Firstly of a triple pattern, which differs only from a RDF dataset that it contains variables. The variable are denoted with a question mark in triple pattern. Secondly it contains a set of variables that the query has to return. Listing 2.8 shows an example query, through which the syntax is demonstrated. After the *SELECT* keyword there is the variable to return and after the *WHERE* keyword inside the parenthesis there is the triple pattern.

```
SELECT ?student
WHERE {
  ?student    rdf:type   domain:Student .
}
```

Listing 2.8: SPARQL Query I.

It can be seen that the namespace abbreviation with prefixes works the same way like in RDF documents. If the query is executed on the dataset from Figure 2.7, then it return the *domain:1* and *domain:3* instances for the variable *?student*.

Furthermore the triple pattern can consists of multiple triples as well, like in the following query:

```
SELECT ?student ?label
```

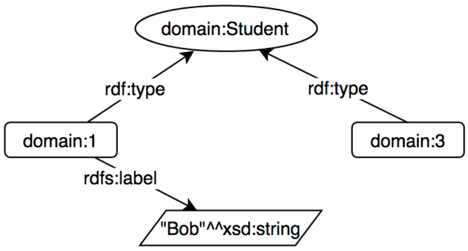

Figure 2.7: Ontology extension

```
WHERE {
  ?student    rdf:type   domain:Student .
  ?student    rdfs:label   ?label   .
}
```

Listing 2.9: SPARQL Query II.

Executed on the same dataset, this query return only the instance *domain:1* because *domain:2* does not have a label, thus there is not matching RDF node for variable *?label*. This means that the triples in a triple pattern are in *AND* relationship. But there is keyword optional that allow the definition of sub triple patterns, which are not required in the RDF dataset. So if the triple with the label variable is in the OPTIONAL sub pattern then the query II. return *domain:3* instance as well, so that the variable for the label will be empty.

```
OPTIONAL {  ?student   rdfs:label   ?label   . }
```

Listing 2.10: Optional sub triple pattern.

Moreover its is possible to define filters on the variables. The most commonly used are the regular expression filter on literals.

```
FILTER regex(?label, "^Bo")
```

Listing 2.11: Regex filter in SPARQL

Finally it is possible in SPARQL to query blank nodes, the same ways with variables. The following query then queries the ontology and returns at least how many key course must be attended by a computer science student.

```
SELECT ?min
WHERE {
  domain:CS_Student  rdfs:subClassOf      ?restriction  .
  ?restriction       rdf:type             owl:Restriction  .
  ?restriction       owl:onProperty       domain:attends  .
  ?restriction       owl:onClass        ?courseType  .
  ?restriction       owl:minQualifiedCardinality    ?min  .
}
```

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 use existing ontologies rather than developing them by ourselves. The main reason for this lies in the fact that the ontology development is a time consuming and tedious process. Two ontologies are taken in the project, one for the human anatomy and one for biomedical investigations.

2.2.1 Ontology for the anatomy of the human body

The ontology modeling the human body is called *Foundational Model of Anatomy* (FMA). 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 [6]. All kind of anatomical entities are represented in FMA, like molecules, cells, tissues, muscles and of course bones. In our project we use only the skeletal system related subset of the FMA. The taken elements are the following two classes (and its subclasses) and 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 belongs to a subclass of the class *Subdivision of skeletal system*. Moreover there are such skeletal subdivisions which are part of another skeletal subdivision. In both cases the relationship is expressed by the property *fma:systemic_part_of*. To define which bone organ belongs to which skeletal subdivision, FMA contains *owl:someValuesFrom* restrictions (Listing 2.13).

```
fma:BoneOrganX   rdfs:subClassOf [
  rdf:type              owl:Restriction .
  owl:onProperty        fma:systemic_part_of .
  owl:someValuesFrom    fma:SkeletalSubdivisionY .
]
```

Listing 2.13: Rules of the skeletal system defined in OWL

These restrictions mean that a bone organ instance cannot stand on its own, but it has to be a systemic part of an appropriate skeletal subdivision instance. Figure 2.8 shows the main structure of the applied subset of FMA by depicting the restrictions with red arrows, and property *rdfs:subClassOf* with dotted arrows.

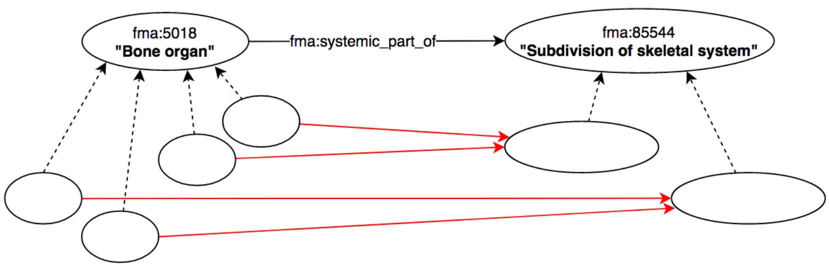

Figure 2.8: Ontology structure for the human skeleton

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 the 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 is continuously evolving as new research methods are being developed. Its main function is to define the rules how biological and medical investigations have to be performed. OBI reuses terms from the *Basic Formal Ontology* (bfo), from the *Information Artifact Ontology* (iao) and from the *Open Biological and Biomedical Ontologies* (obo) [1]. The most important classes and properties adopted by the *RDFBones* project are the following ones:

- Classes

  Investigation - obo:0000015

  Process - bfo:0000015

  Material Entity - bfo:0000040

  Information Content Entity - iao:0000030

- Properties

  has part - bfo:00000051

  has specified input - obi:00000293

  has specified output - obi:00000299

In contrast with FMA, in OBI the subclasses of the applied main classes and restrictions for anthropological investigations are not given. The idea of *RDFBones* project is to define custom investigations by defining subclasses of the above mentioned classes, and restriction on the properties. These further ontological statements are called extensions. The following image illustrates an example ontology extensions for an investigation (the notation is the same like on Figure 2.8, just the restriction are defined on the OBI properties).

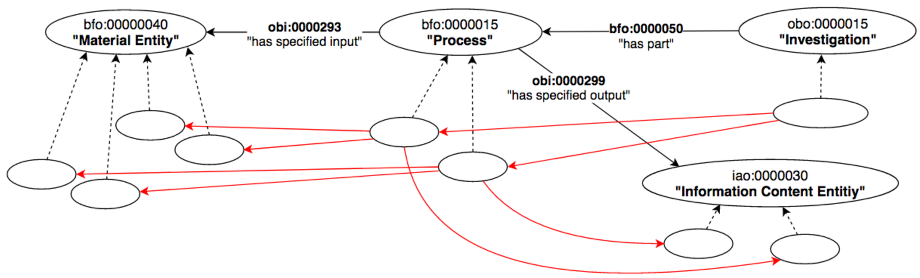

Figure 2.9: Extending OBI ontology

It can be seen that the investigation may contain several processes (*has part* predicate), and each process has various inputs and an output. In our case the material entities are segments of bone organs but they could be any kind of material. During a particular process, an entity is studied and the result of the study is an in information content entity instance, which represents some measurement value. The provided scheme is just an illustrative example, the modeling of the investigation in reality is more complex, and will be discussed in a more detail in Chapter 4.

The conclusion is that with these classes and properties, OBO offers a powerful vocabulary for defining custom investigations. The advantage of such ontological description is that everything is stored by means of RDF triples, and therefore it is possible to develop software application that generate user interfaces by querying these extensions.

2.3 Web applications

2.3.1 Fundamentals

Client-sever architecture

A web applications have two main units, the client and the server. The client is the application, which the user interacts with, while the server is an other application serves the request coming from the client. The client and server programs are running normally on different machines, and they communicate through Hypertext Transfer Protocol (HTTP). The main mechanism is that client, which is a web browser application, sends an HTTP request through the web. The server is found based on the URI of the HTTP request, and upon the content of the request the server returns a document written in Hypertext Markup Language (HTML). The HTML document contains the definition of the elements of the interface and it is interpreted by the web browser.

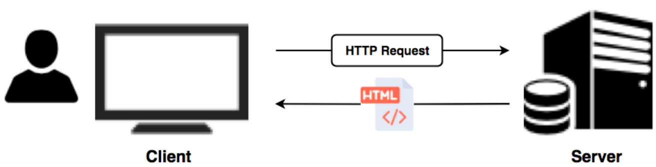

Figure 2.10: Client server communication

An HTML document consist of different elements like buttons, tables input fields. Each element is represented by specific tags. The HTML has a hierarchical structure, and each element is the child of the tag *html*. On listing 2.14 it can be seen that each tag has a opening and closing element.

```
<html>
  <div class = "welcome">  Welcome on the web application </div>
  <a href="http://webapp.com/page1"> Page 1 </a>
</html>
```

Listing 2.14: Example HTML document

The tag, like the *html* in the example can contain further tags, and have normally at least one paramater (i.e *class* and *href*). The tag *div* is the most general element of web pages, while the *a* tag defines link, where the *href* parameter is defines the URI of the HTTP request they initiate. The request of the example link arrives to the same server, and the task of the application is to process the request URI and return the HTML document for the new page.

Data driven web applications

Most of the web applications nowadays incorporates databases. Databases are used to store large amount data in an organized way. Databases are always come along with a database management system (DBMS), that allows to create, edit, delete and retrieve data in the database. So DBMS is software that acts as an interface between the web application and the data. In the following, the database and DBMS together will be referred just as database.

By web applications using databases the most important point, that web pages are not statically defined in HTML files, but generated by the application dynamically using a particular dataset. The process of loading of a web pages showing any data, starts with the execution of a query. This means the web application sends a query to the database and gets a desired data. The result of a query in terms of the web application is conventionally a list of data objects. The term object is used generally, and refers to a data type that organizes its values by keys. The elements of the output list represent the rows of the query result table, and the fields of the object in turn the rows respectively.

To define how the web page has to generated from the dataset a so-called template file are used. The template file is basically an HTML document, that is extended with some additional syntax, which can be interpreted by the template engine. There a lot of template engines and technologies, but in the following I provide an illustrative example from *Freemarker* template language, that is used by as well the VIVO framework.

```
<#list students as student>
  <tr>
    <td>${student.name}</td>
    <td> <@linkButton "grades" ${student.id} /> </td>
  </tr>
</#list>
```

Listing 2.15: Template file example

Important that the data that is passed to the template engine has a name, by which it is referred in the template file. In the example query result is stored in a variable *students*, where each student object has two keys, the *name* and *id*. The tag *#list* represent a loop, that iterates through the input list. The content withing the *#list* tag appear as many times in the resulting HTML, as many elements the input list have. The variables withing normal HTML tags are accessed with *${..}*.

Other useful feature of templates that it allows the definition of macro, which acts like subroutines in programming. In the example the macro *linkButton* takes two input parameter and generates the *<a/>* tag with a certain image. This make the development more convenient and clear.

```
<#macro linkButton urlMapping id>
  <a href="webapp.com/${urlMapping}?id=${id}">
    <img src="webapp.com/images/jump.jpg"/>
  </a>
</#macro>
```

Listing 2.16: Macro definition

In the macro definition it can be seen that the url of the link contains the parameter *id*, which is an additional information in the HTTP request. The idea is that the request is handled by a such server routine that substitutes the value of the parameter into a query, in order to get data about individuals. Then the returned page may contain additional link for further data entries. This is the fundamental method how web pages are used to discover data from databases.

Interactive web pages

The previous section showed the principles of how data can be browsed by means of web pages and links. In such static cases the HTML document was assembled completely by the server, and the links initiated the loading of whole new pages. Nevertheless it is often more efficient and leads to better user experience if the new content is added dynamically to the current web page. Such functionality can be achieved with JavaScript (JS), which is a scripting language run by the browser. The two most fundamental features of JS, that it is capable of storing data in variables and can modify the content of HTML elements of the pages.

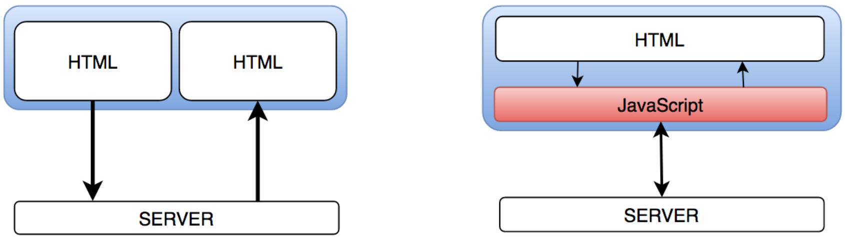

Figure 2.11: Static and dynamic web pages

Figure 2.11 illustrates the difference between static and dynamic web pages, where the blue rectangle stand for the client side. The idea of dynamic web pages in data driven web applications, that if new content has to be shown on the page, then not an HTTP request is sent to the server, but a JS routine is called. JS code is defined in the HTML document within the *<script/>* tag. The routines are assigned to HTML elements by their id-s in the following way:

```
<html>
```

```
  <div id = "button"> Show content </div>
  <div id = "content"></div>
  <script>
    ...
    $("#button").click(function(){
      $("#content").text(data)
    })
    ...
  </script>
</html>
```

Listing 2.17: JavaScript routine assigned to an HTML element

This simple JS code illustrates how it is possible to load new content to the HTML element. In the example the *append* function sets the text of the div with id *button*. The *data* is a JS variable holding some text. The value variable can be either initialized by the server by the page assembly, or by AJAX calls. AJAX is an acronym for *Asynchronous JavaScript and XML*. The AJAX is technology that allows JS to load data from the server asynchronously, which means that the request is initiated through JS routine, and response arrives as well to JS routine. The data by AJAX is mostly is JSON format. JSON stands for *JavaScript Object Notation*, which is a standard data format. A JSON object consists of a set of key-value pairs, where the value can be any data, arrays or even further objects.

```
{ key1 : "data",
  key2 : ["value1", "value2"],
  key3 : { key : "value"}  }
```

Listing 2.18: JSON object example

2.3.2 VIVO framework

VIVO is an open source web application framework, developed particularly for browsing and editing RDF data. VIVO utilizes that the data scheme in RDF is stored by means of triples as well, and it can adopt the pages to the ontology. It offers an ontology editor and there are particular features of the application that can be customized through a specific configuration dataset. This dataset is in RDF format too, and defines the way in which the data is displayed and edited on the web pages. VIVO allows to manipulate this configuration triples via the web interface, which enables the extension of the application to some extent conveniently without coding.

VIVO maintains two basic type of pages. The first is for displaying the existing data, and one for creating and editing data. The former will be referred as profile page, because it shows the data related to one individual RDF instances, and the latter will be referred as entry form.

Profile pages

The task of the profile pages in VIVO, is to display all instances and literals that constitute to a triple with the instance, whose page has been called. These neighbor nodes are shown on the page grouped into the properties. Furthermore the properties are organized into so-called property groups. A property groups can be defined through the following set of configuration RDF triples.

```
vivo:overView          rdf:type        vivo:PropertyGroup  .
vivo:overView          rdfs:label        "Overview"  .
```

Listing 2.19: Property group definition

As we can see there is a class vivo:PropertyGroup, which indicates that VIVO incorporates a small ontology for web application configuration. Any property of a domain ontology loaded into VIVO can be assigned to a property group by an additional RDF triple:

```
obo:OBI_00000299       vivo:inPropertyGroup       vivo:overView  .
```

Listing 2.20: Assigning property to propery group

This means if an applied ontology is extended with further properties then they just have to be assigned to a property groups and VIVO profile pages can display them, and no modification is required in the template file or in the application code.

Other useful configuration possibility are offered by the so-called faux properties, which allow further grouping of RDF instances on the profile pages. They make possible to group instances based on their types even if they are connected to the main instance with the same property. Their definition lies in RDF triples as well, and for each one the base property, the range class and property group has to be set.

Figure 2.12: Faux properties on VIVO profile pages

Figure 2.12 show a profile page of skeletal inventory. The blue entries represent the instances, and they are links to the profile pages of the individual skeletal subdivision. Both the *Skull* and the *Vertebral column* instances are connected to the skeletal inventory with the property *rdf-bones:hasSkeletalSubdivision*, but since two faux properties (*skull* and *vertebral column*) are defined, these instances appear separately on the profile page.

Custom entry forms

Important part of the application is the creation of new RDF data. On Figure 2.12 next to the property field, a button with a + sign can be seen, which is link to a data entry form page. The peculiarity of VIVO that from each property field, the request arrives to the same handler routine on the server. These requests have always three parameters:

- subjectUri
- predicateUri
- rangeUri

Where the first is the URI of the instance to whom the profile page belongs, the second is the URI of the property (can be a faux property), and the third is the URI of the range class of the property. By default VIVO redirects the user to such an entry form where only one instance or literal can be added. However it is possible to define conveniently custom forms for more complex datasets.

By custom data input processes, there are two main parts to define. The first is the elements of data input form in HTML, and the second is the RDF data that has to be created. In VIVO these definition have to placed into a class, and the class have to be assigned to the property with again RDF configuration triples.

```
rdfbones:hasSkeletalSubdivision    vivo:customEntryForm
    "rdfbones.SkeletalSubdivisionFormGenerator.java".
```

Listing 2.21: Entry form generator class definition

Thus if the request from the profile page arrives to the server, VIVO queries the class name by the parameter *predicateUri* and instantiates it. By writing a class during the development incorporates the static definition of its variables. Each customly defined class implements a predefined VIVO interface, thus there is a specific set of variables that have to be initialized for the definition. The most important is the string variable *templateFile*, that holds the name of template file that have to be displayed.

```
  this.templateFile = "subdivision.ftl"
```

Listing 2.22: Form defintion in Java

Of course this template file has to be created too. The following code shows the definition of a template file using *Freemarker* macros.

```
<form>
  <@selector  "type">
  <@textField "nr">
  <@floatField "weight">
  <@submitButton>
</form>
```

Listing 2.23: subdivision.ftl

In the provided example there is one selector on the form, which allows the selection of the type of the new skeletal subdivision. Furthermore there are two literal fields, one for the catalog number and one for the weight. VIVO contains some useful *Freemarker* macros that makes the form definition quicker. Each form element has to have a variable name, through its value after the submission can be identified.

The next fundamental variable of the generator class is the *triplesToCreate*, which contain the RDF triples to be created during the data input process.

```
this.triplesToCreate = "
  ?subjectUri    rdfbones:hasSkeletalSubdivision    ?subDivision.
  ?subDivision   rdf:type                           ?type.
  ?subDivision   eq:catalogNr                       ?nr.
  ?subDivision   eq:weight                          ?weight. "
```

Listing 2.24: RDF Triples to create

It can be seen that there are variables like in SPARQL, defined with question mark. Furthermore it has to be defined for each one variable what types are they. For this purpose there are three lists, on for the URIs on the form, one for the literals on the form, and for the new instances, whose URI do not come from the form but will be generated during the data creation.

```
  this.urisOnForm = {"type"};
  this.literalsOnForm = {"nr", "weight"};
  this.newInstances = {"subDivision"};
```

Listing 2.25: Variable type definition

The last type of the definition in such simple case, is to define the SPARQL query that retrieves the options for the selector fields on the form. This variable is *Map<String, String>* type in Java. The keys of the map are the variable names, and the values are the SPARQL queries.

```
this.formVariables.put("type",
  "SELECT ?uri ?label
    WHERE {
```

```
    ?uri   rdfs:subClassOf    ?rangeUri .
    ?uri   rdfs:label       ?label   . }
```

Listing 2.26: Query for form data

This query is executed before the form page loading, and the result is passed to the template engine, which generates the options withing the *selector* macro. Please note that the variable *rangeUri* coming as an input with the HTTP request. This way it is ensured that the type of the *subDivision* variable, will be the subclass of the range class.

These section gave an insight into how it is possible to define data input processes in a declarative way within the VIVO framework.

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 problem lied in that the number of instances which were created through the process was constant, in particular one, and only their types and literal attributes were set by the user through HTML input elements.

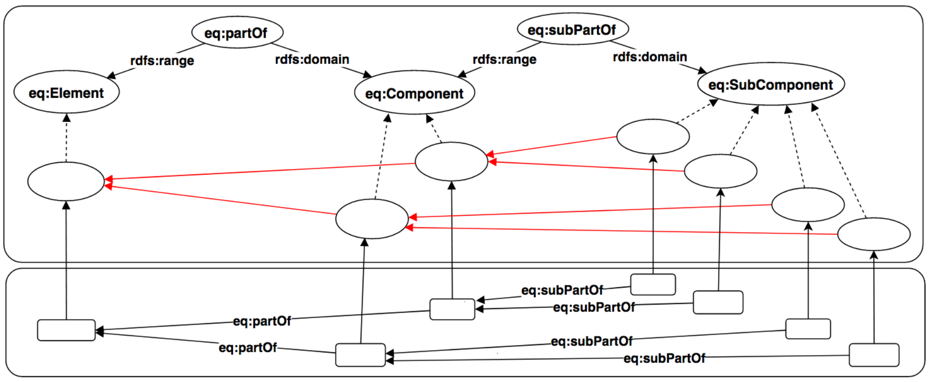

Figure 3.1: Ontology and RDF triples for complex entities

Nevertheless there are more complex entities consisting of several sub parts, where these sub parts 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 names are not relevant.

Such dataset poses the requirement for the input form, that the user has to be offered such interface elements which enable to add the components and subcomponents step by step. Adding a component means in terms of the form that a new sub form appears which contains further input fields for the component instance.

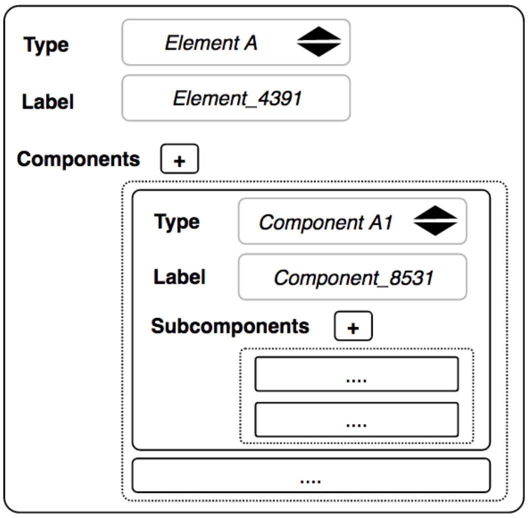

Figure 3.2: Multi level form

Figure 3.2 shows the layout of the form for multi level data. The additional element compared to the static HTML form is the field with an button for adding the sub forms. The dotted rectangle stands for the container element that encompasses the added sub forms. The form data contains the same way the key value pairs for the main form element, but it has an additional array field that contains the data objects of the sub forms. The sub form data object works the same way, if the have sub forms then they contains further arrays. To realize such functionality, JavaScript routine is required on the form, that ads the sub form elements to the container, and generates the appropriate form data upon the user actions.

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 them, 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 side 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 possibility 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.

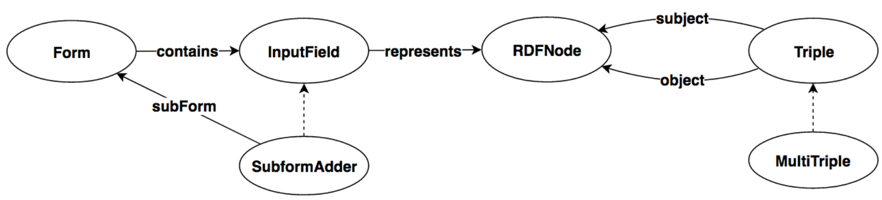

Figure 3.3: Classes and relationships of the vocabulary

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 model the interface, and the other three on the right in turn are for the RDF data definition. 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. The 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.

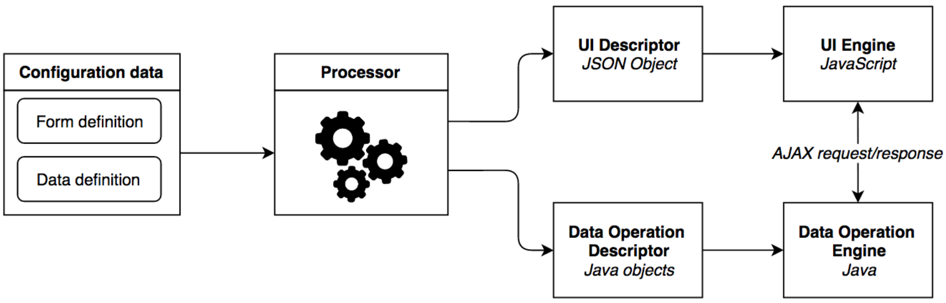

Figure 3.4: Framework functionality outline

The form descriptor data is a JSON object containing the definition of the form elements including the sub forms, and data dependencies between the selector fields, 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 **??**, 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-

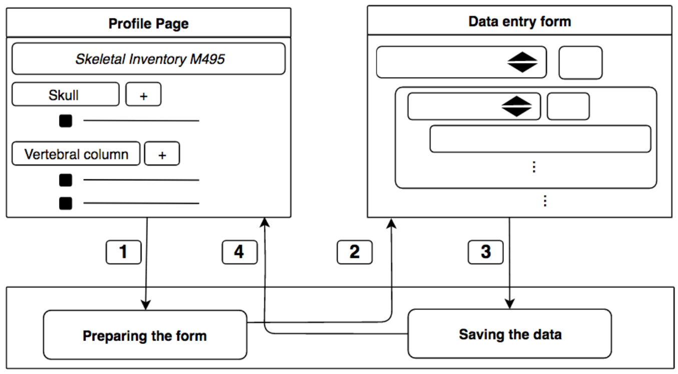

Figure 4.1: Complete workflow of data input process

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.

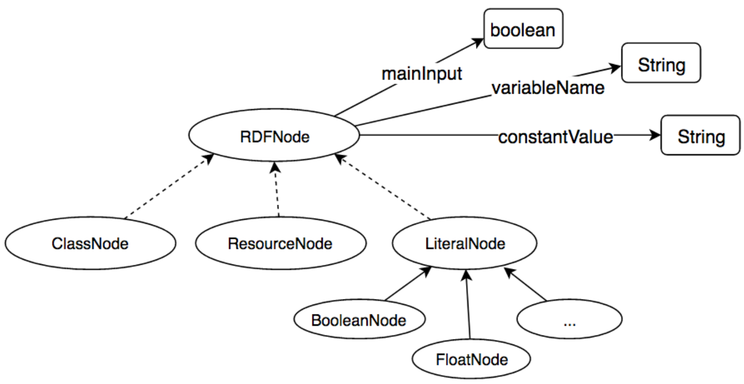

Figure 4.2: Variable types and their attributes

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.

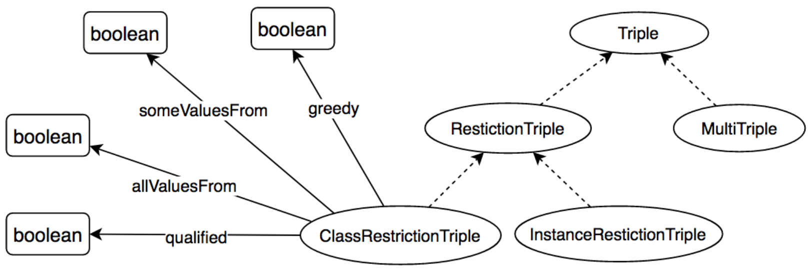

Figure 4.3: Triple types

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.

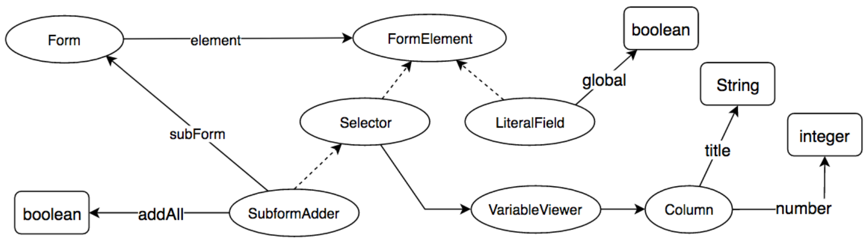

Figure 4.4: Form definition

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.

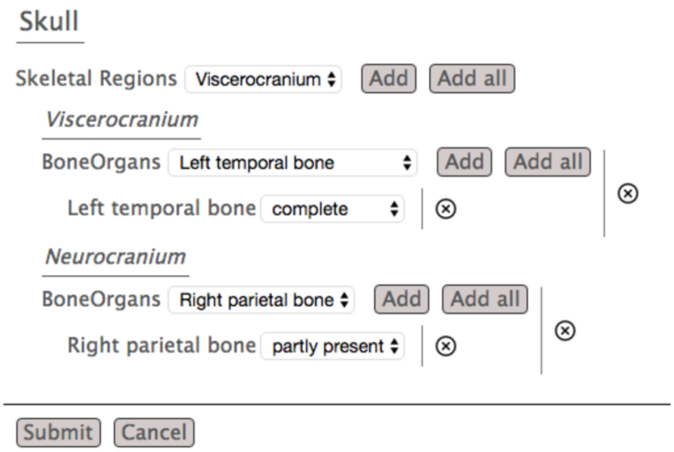

Figure 4.5: Input form for skeletal inventories

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

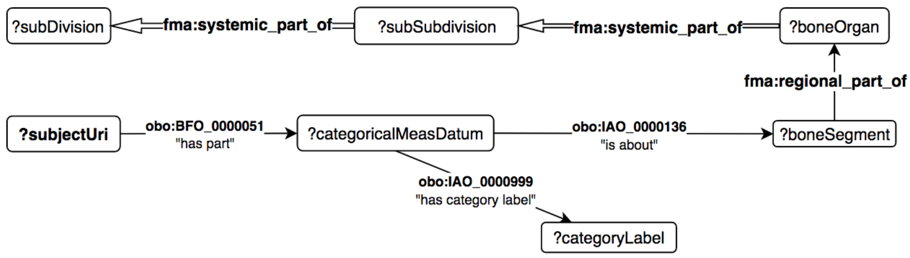

Figure 4.6: Skeletal inventory data triples

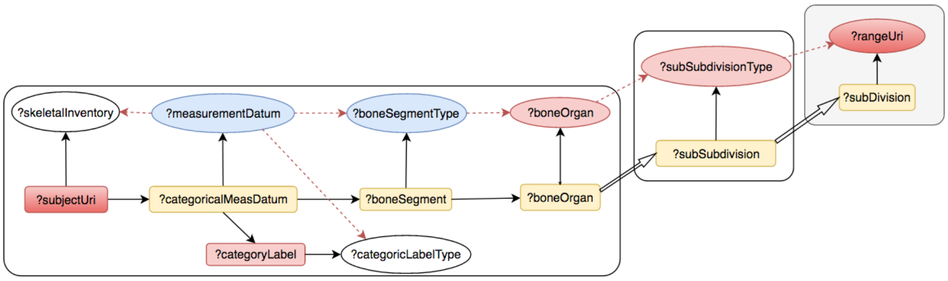

Figure 4.7: Complete data definition

the structure which is followed by the form as well.

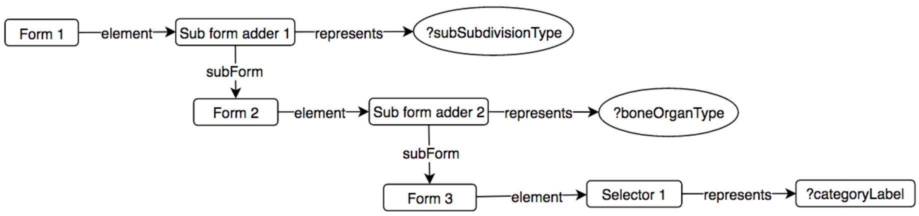

Figure 4.8: Form layout definition

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.

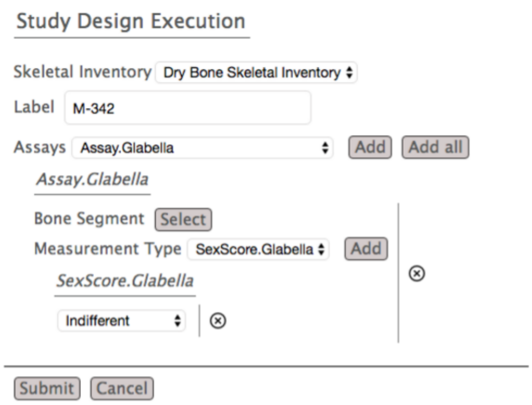

Figure 4.9: Input form for study design execution

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.

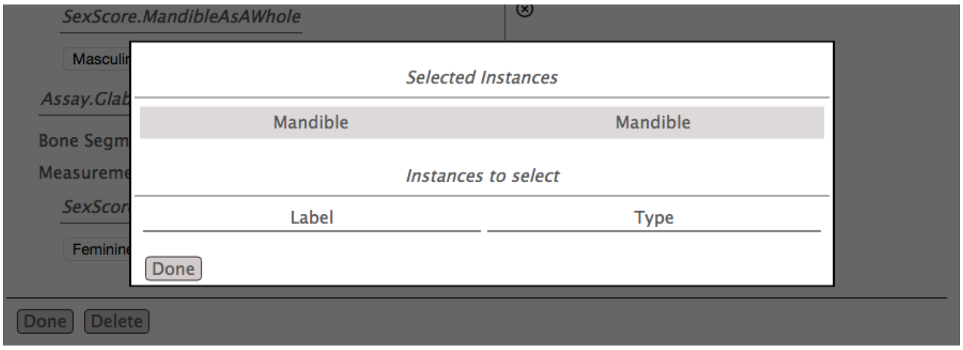

Figure 4.10: Instance selector for existing bone segment

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.

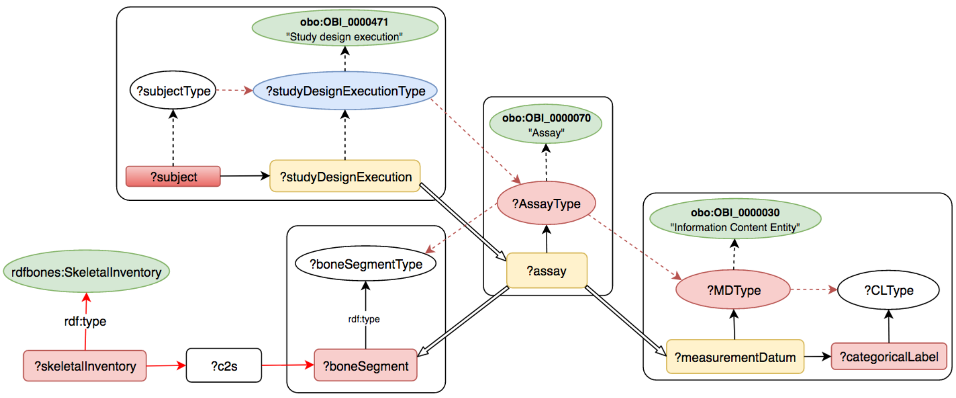

Figure 4.11: Complete data model

- 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.

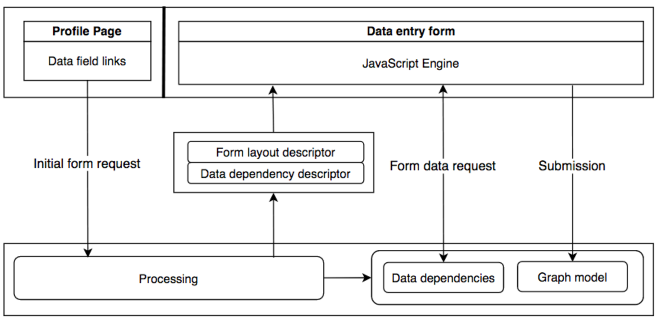

Figure 5.1: More detailed scheme

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.

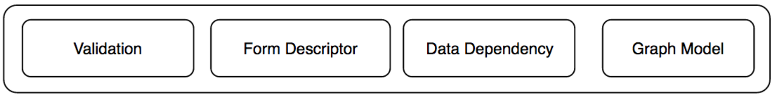

Figure 5.2: Processor tasks

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.

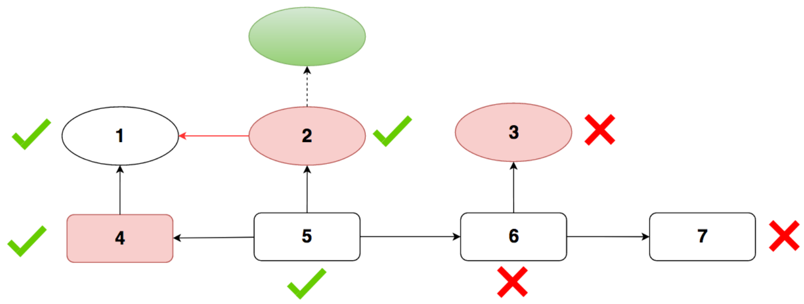

Figure 5.3: Valid and invalid nodes

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.

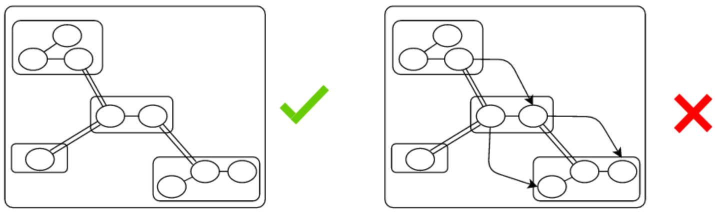

Figure 5.4: Valid and invalid graph

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.

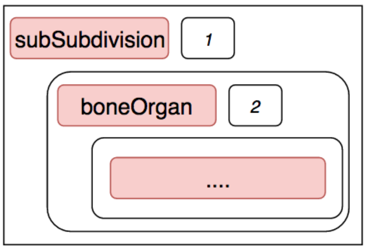

Figure 5.5: Form element order

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

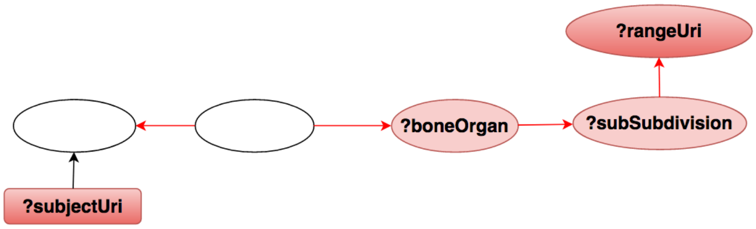

Figure 5.6: Skeletal inventory data constraints

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.

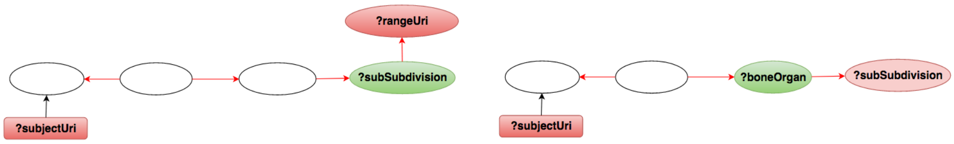

Figure 5.7: Form dependency subgraphs

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

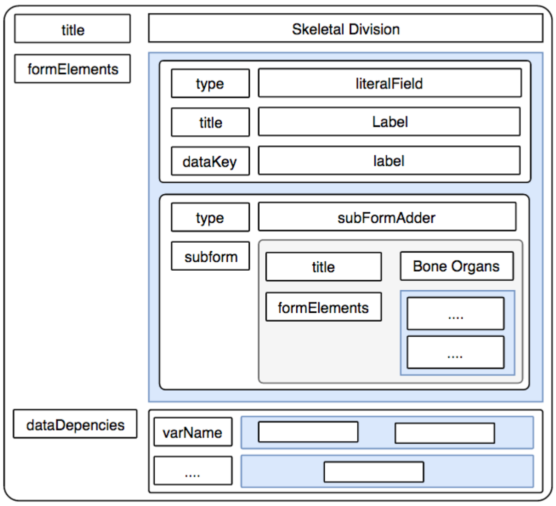

Figure 5.8: Form descriptor JSON

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*.

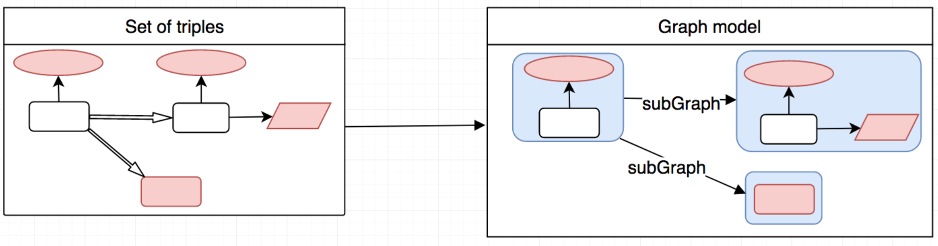

Figure 5.9: Conversion from triples into graph model

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 retrieval, where the graph performs the SPARQL queries based on its triples, and generates the arrays of objects from the result table.

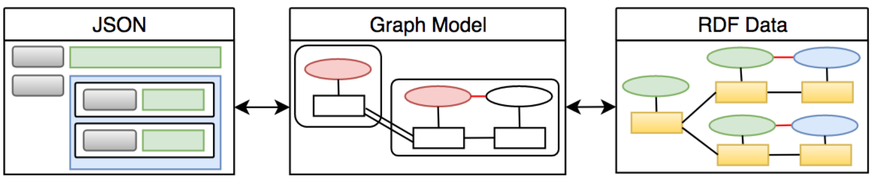

Figure 5.10: Graph decomposition

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

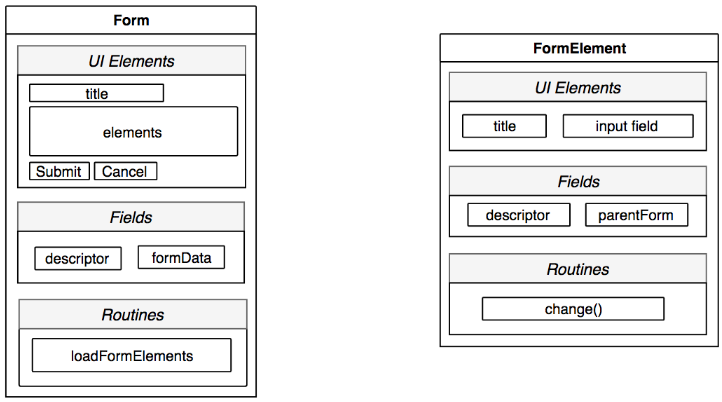

Figure 5.11: Form and form element classes

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.

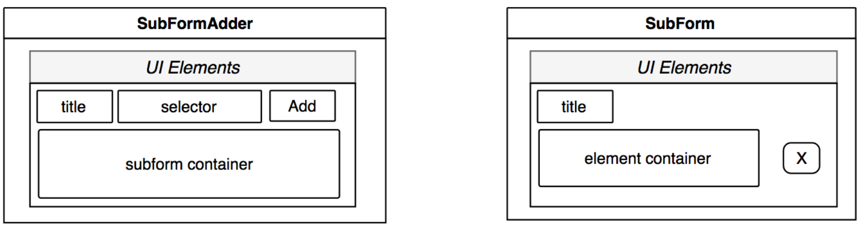

Figure 5.12: SubForm and sub form adder

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.

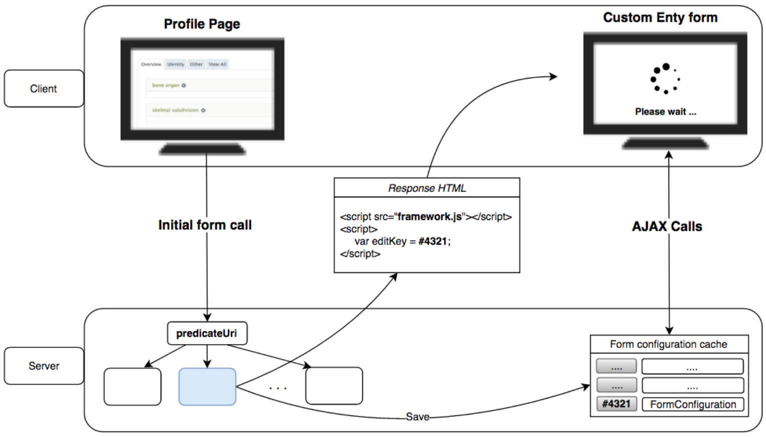

Figure 5.13: Form loading process

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.

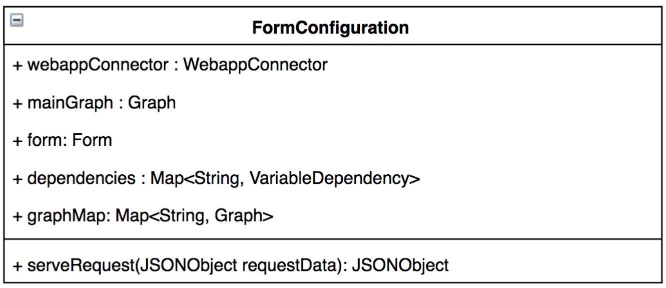

Figure 5.14: UML class diagram for FormConfiguration

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.

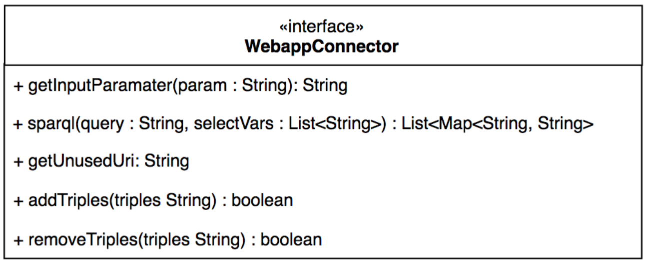

Figure 5.15: UML class diagram for WebappConnector

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()*.

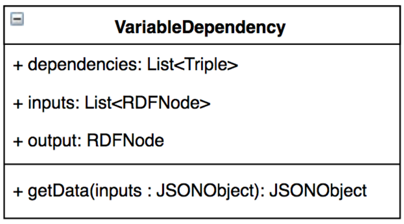

Figure 5.16: UML class diagram for VariableDependency

```
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)
...
}
```

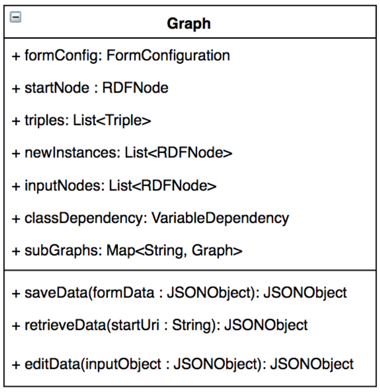

Figure 5.17: UML class diagram for Graph

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.