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

Ontologies describing biological research processes are getting prevalent nowadays. They are used to define the rules of the investigation, and establish a basis for their standardized. The goal is to replace textual documentation of the processes, with structured data. The advantage of such representation that information can be exchanged and software agents can perform reasoner algorithm. The data describing the investigations the ontologies classes for the entities, measurement analysis steps etc. Furthermore the rules of these processes are described by means of restrictions. As these ontologies are based on formal logic, and they are machine readable format, it is possible to build such applications that are building their interfaces upon them. However the data structure itself is not sufficient to build GUI-s that allows to convenient digitalisation of the research processes. It is an additional problem and time consuming and tedious. The thesis aims to build such application framework that allows the definition of the input processes on a higher level logic the abstract from the implementation details. The system is supposed to accelerate the development and should be able to define.

# 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. In anthropology each institute has different set of skeletal remains and they conduct their investigations differently, thus it is not possible to cover during the project period the problem of other institute as well. Therefore the core idea of the project is that to make possible the definition of custom processes through ontology extensions. This is only possible by means of RDF data, because the ontologies in OWL can be extended, and specific rules can be defined through restrictions. We are using Semantic Web application framework called *VIVO*, which is capable of editing not only the RDF data itself but the ontology as well. So the idea of the project that not only the data will be exchangeable but the extensions as well. The task of the application is to generate the data input forms based on the ontology extensions.

Web application so that they do not

## 1.2 Goal of the thesis

The data input forms and the data model is tightly coupled. Which means that the application has limited tasks of exercises. The The goal is to implement a such vocabulary that allows abstracts from the low implementation.

This would allow the developers the implementation of data input processes of people without programming experience. This

The vocabulary incorporates the definition of the input form layout, and the underlying data structure. Based on this compact definition the framework has to be able to generate the interfaces and perform the necessary data operations. T

## 1.3 Thesis outline

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. 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#>.

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#>.

So a literal value in RDF looks as follows:

"Some literal value"^^xsd:string

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.

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 eg: <http://example.org#>.

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 .

The actual data about a person:

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

### 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.

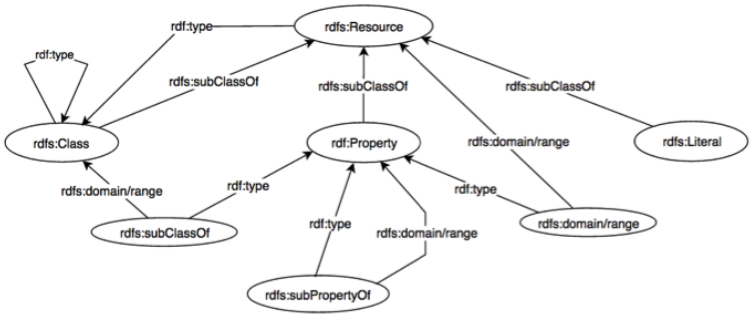


Figure 2.1: Main structure of the RDFS vocabulary

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

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

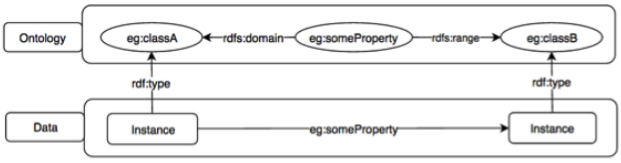


Figure 2.2: RDFS domain and range definition

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.

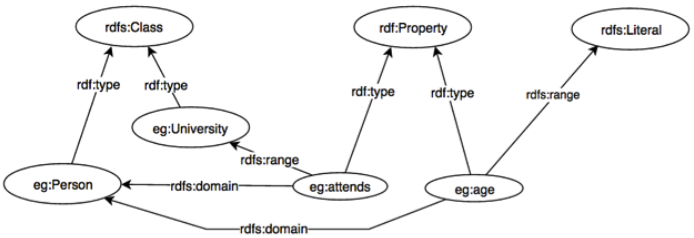


Figure 2.3: RDFS domain and range definition

### 2.1.3 OWL

OWL, abbreviation for Ontology Web Language is an extension of the RDFS vocabulary. OWL allows expressing additional constraints on the data, 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 2.1: 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 2.2: OWL restriction in N3 format

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

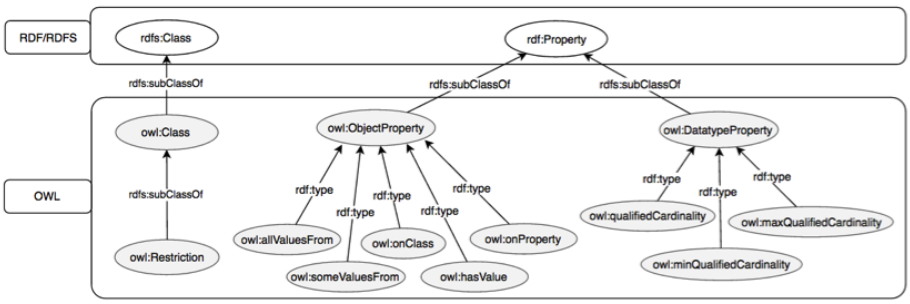


Figure 2.4: A subset of OWL vocabulary

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.

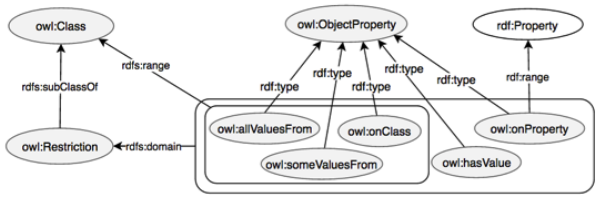


Figure 2.5: OWL object properties

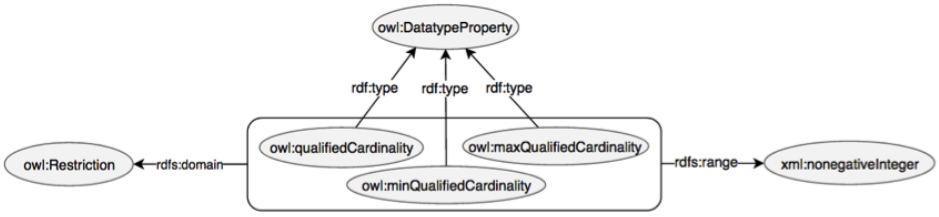


Figure 2.6: Properties for qualified cardinalities

### 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:

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

Listing 2.3: 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 2.4: 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 2.5: 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 [4]. 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.

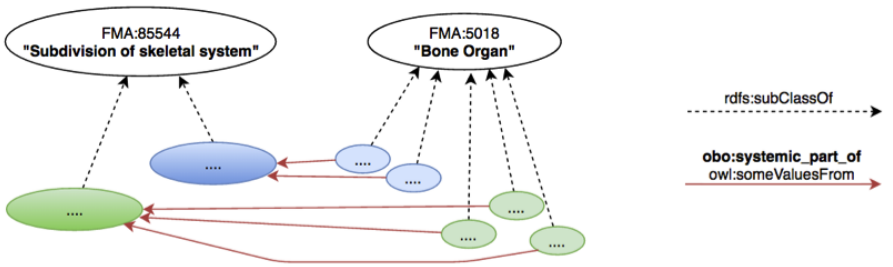


Figure 2.7: Ontology structure for skeleton

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*[3]. 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.

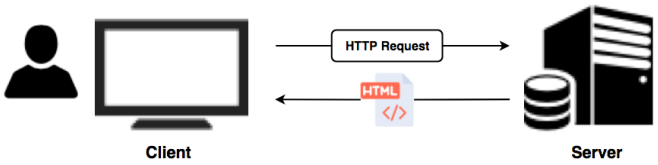


Figure 2.8: Client server communication

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.

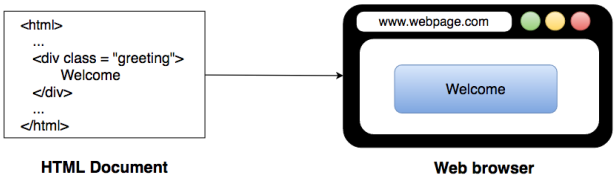


Figure 2.9: HTML document is interpreted by the browser

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 2.6: 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 2.7: 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.

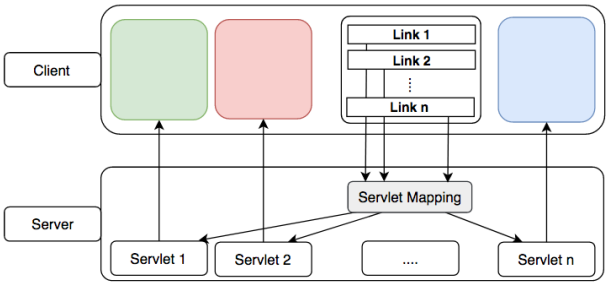


Figure 2.10: Navigation through the web application

### 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.

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

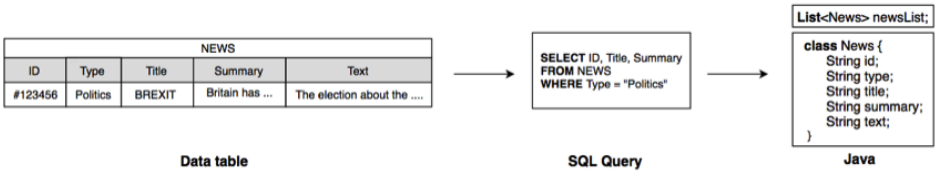


Figure 2.11: Data flow

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 2.8: 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:

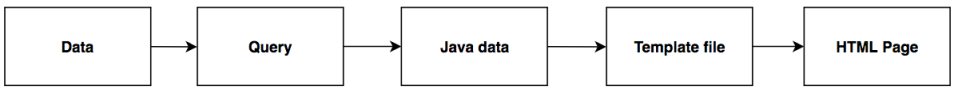


Figure 2.12: Flow of information from DB to client

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.

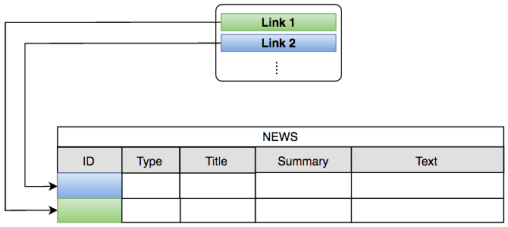


Figure 2.14: Links to data items

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.

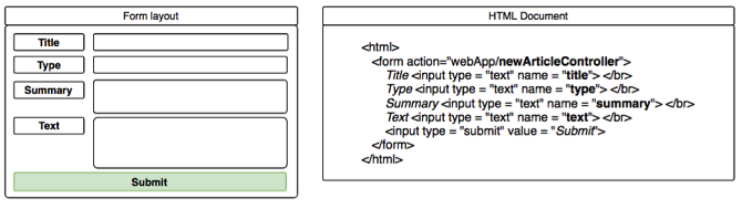


Figure 2.15: Form layout and HTML document

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.

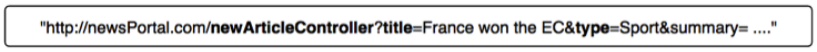


Figure 2.16: Request with parameters

Then initialized class instance is passed to the database where the entered data will be persistently stored.

```
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

### 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.

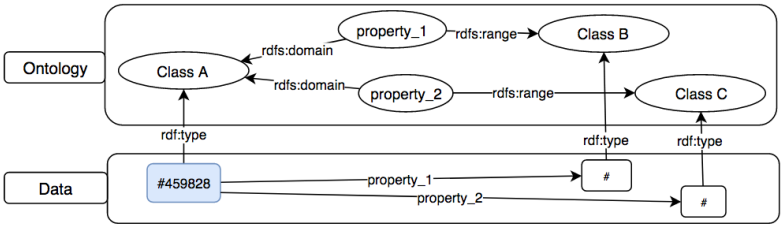


Figure 2.18: Ontology and data in RDF

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 2.9: 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. 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 2.10: Ontology adaptive template file

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

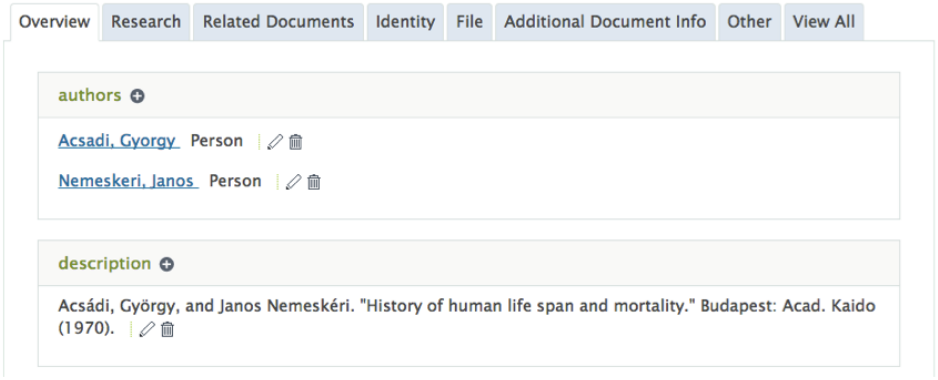


Figure 2.19: VIVO Profile page

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.

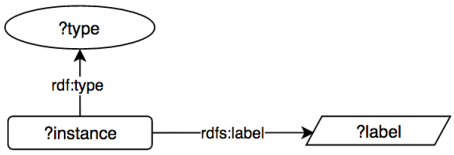


Figure 2.20: Triples representing a new instance

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 2.11: 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.

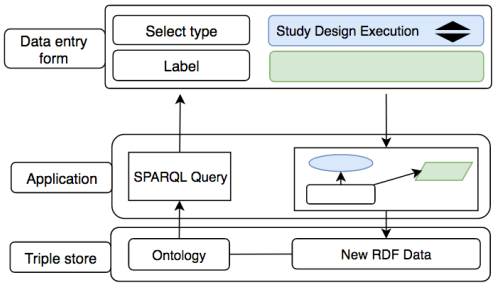


Figure 2.21: Data input scheme by RDF

Chapter 3

# Problem Statement

This chapter is divided into three sections. As the application is highly dependent on the underlying data model, the first section is dedicated to the data scheme describing the investigations in scope. The second chapter in turn addresses the problem of web applications that allows the creation of RDF data explained in the first section. It covers the issues of both the client and server side implementation and their communication. Finally section 3.3 outlines the scheme of the solution proposed and implemented by the thesis work.

## 3.1 Modeling anthropological research activity

This section consist of three subsections. The first two (3.1.1 and 3.1.2) describes how the RDFBones ontology (developed during the project) integrates the *FMA* and *OBI* ontologies for describing research processes related to anthropology. While the third section (3.1.3) discusses how can the core ontology be extended to define custom bone segments and processes.

### 3.1.1 Data on skeletal remains

We have seen in section 2.2.1 the base structure of the human skeleton. The most important point is that not only individual bones will be represented in the data we create, but the skeletal regions as well, like skull or vertebral column. The institute where these investigations are conducted posses mainly skeletal remains of skulls. The skull has the peculiarity that it does

not consists directly of bones organs, but from two sub skeletal divisions, and these two subdivisions contain the bone organs. Figure 3.1 shows the ontology subset for the skull and the data instances (each denoted with #). The red arrows denotes restrictions on the properties *fma:systemic_part_of*.

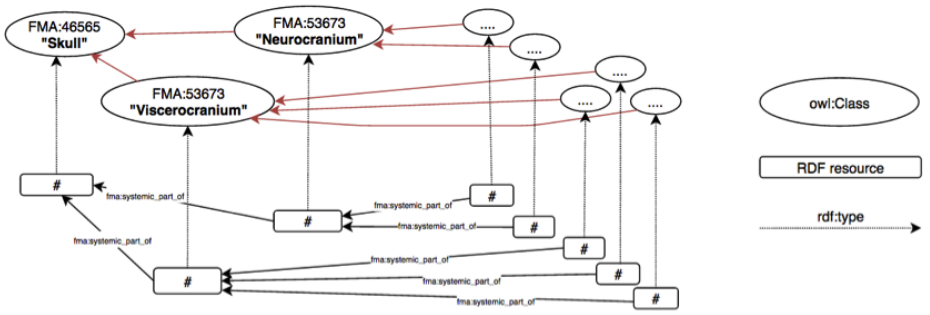


Figure 3.1: Ontology and triples of the skull

So we know how skull and its subdivisions and bone organs are represented by RDF, but there are processes where specific bone segments have to be addressed as well. Therefor RDFBones have the class *rdfbones:SegmentOfSkeletalElement*. This instances of this class is connected to the instances of the class *Bone Organ* with the property *fma:regional_part_of*.

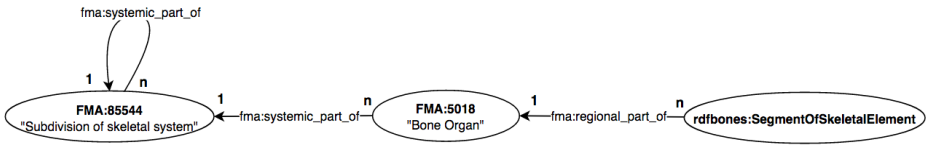


Figure 3.2: Bone segment in RDFBones

Furthermore these instances representing skeletal remains do not stay in the database individually. If a researcher takes a specific skull from the collection of the institute, it makes a so-called skeletal inventory, which records what bones segments are complete, partly present or missing. To store these information in the RDFBones ontology introduces three further classes (above the one for bone segment), which are all the subclasses of *OBI* classes.

Figure 3.4 illustrates the dataset through a simple problem. The upper left part of the figure shows that a specific bone is divided into three bone

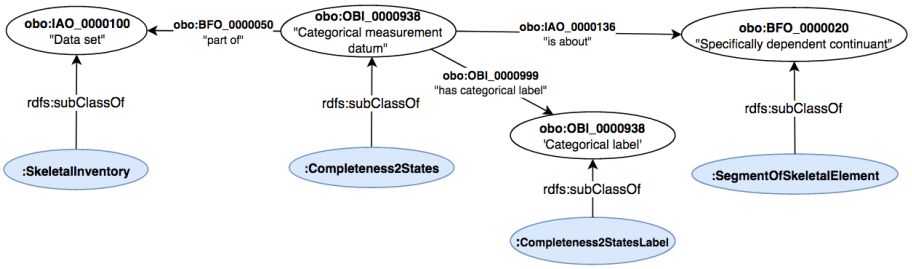


Figure 3.3: RDFBones as extension of OBI

segments, while the right shows the existing bone from which the data has to be stored. It can be seen the section I. is complete, section II. is just partly present and the III. is missing. The lower part of the figure then shows that the three segment are represented as subclass of the *:SegmentOfSkeletalElement* class (denoted with blue), and from the third type there are no instance have been created, while the first two are connected to the skeletal inventory and compleness label instances through *:Completeness2States* instances.

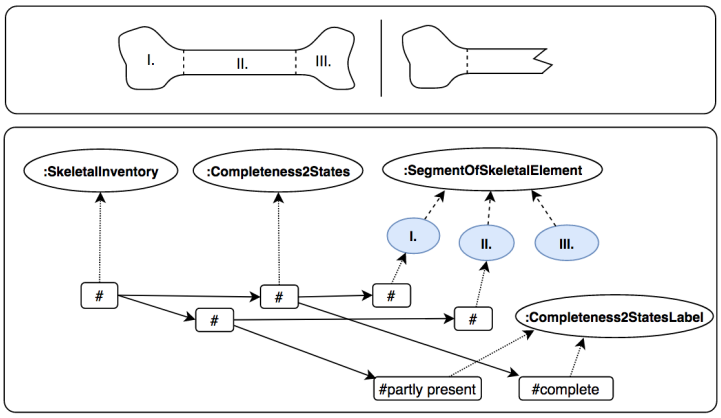


Figure 3.4: Custom bone segment example

This is the way how the information is stored about skeletal remains using the RDFBones ontology.

### 3.1.2 Investigation process

An investigation is done by execution of the a study design. It has three parts, the assay, the data transformation and the drawing of a conclusion.

From assays and data transformations there can be more in one execution, but there is only one conclusion. The inputs of the assays are always segments of skeletal elements, an their output is always a measurement datum, while the data transformation's input and output are both measurement datums.

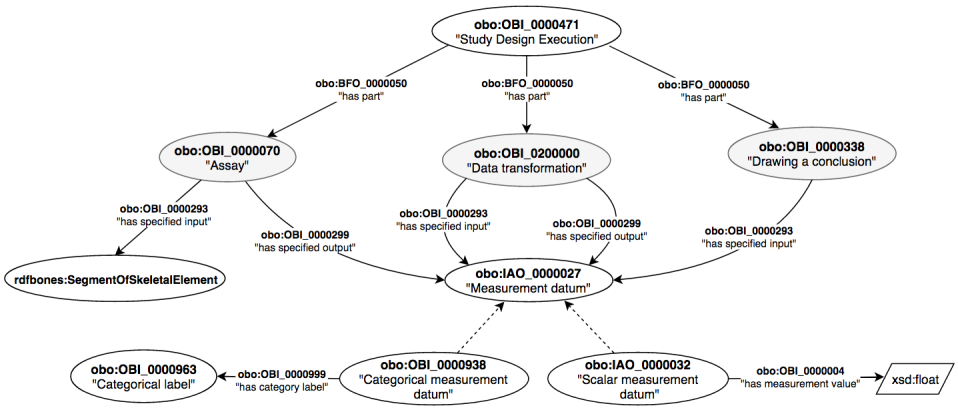


Figure 3.5: Applied subset of OBI ontology

To understand a bit more what this data model can be actually used, let us take the example of an investigation, whose goal is to determine if a taken skull belonged to a male or female. The basis is that the male and female skeleton has different peculiarities that can be quantified, how expressed they are. Figure 3.6 illustrates an example the token *Glabella*, which is on the *Nasal bone*, and its expressions.

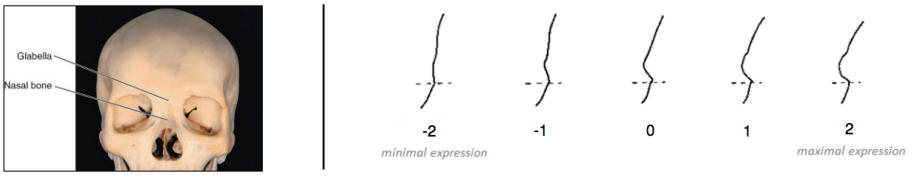


Figure 3.6: Glabella and its expressions

The larger the numbers for masculine and the lowers are feminine expression. An assay in this case is an assignment of a scalar value to a bone segment. The investigation process does not take only one bone segment but several different ones, to reduce the possibility of the erroneous output. Fig-

ure 3.7 shows a dataset of a study design execution where the green arrows stand for *has specified input* and the blue ones for the *has specified output* predicates. The idea is simple, the output of the assays are aggregated, and if the output is smaller then zero then it was a male, otherwise a female.

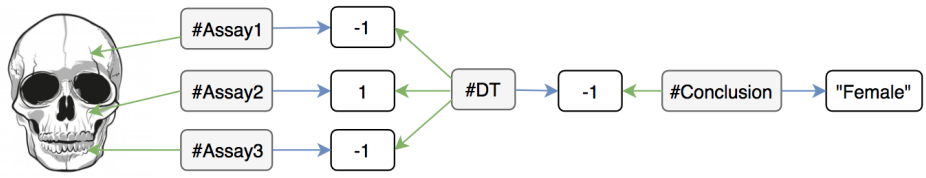


Figure 3.7: Study design execution dataset

Where the light grey boxes are the instances of *Assay*, *Data Transformation* and *Drawing conclusion* classes respectively.

### 3.1.3 Ontology Extensions

The previous two sections introduced the data scheme of the problems of the project. This part provides a more detailed explanation about how exactly these ontologies can be extended to tackle custom problems. By skeletal inventories *RDFBones* ontology allows to define custom bone segments of the bone organs in order to enable more fine-grained representation of the research activity. However by most of the cases it is sufficient to address the bone as a whole. For such cases *RDFBones* has the class *:PrimarySkeletalInventory* which encompasses all the entire bone organs. The definition of this inventory can be seen on the upper part of Figure 3.8.

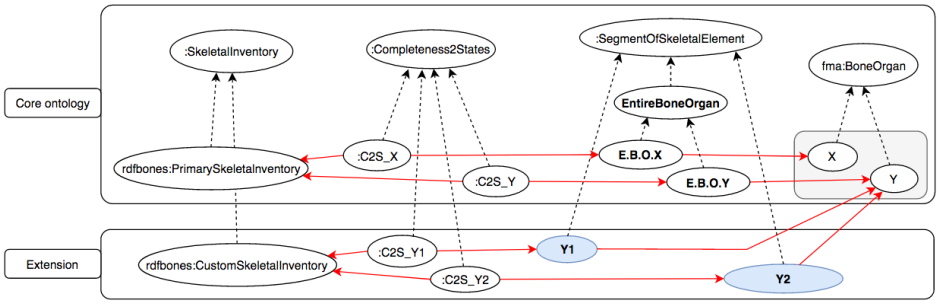


Figure 3.8: Ontology extension for skeletal inventories

The class *:EntireBoneOrgan* has as many subclass as many bone organs are there. The classes $X$ and $Y$ represents the set of all bone organs. Furthermore each of the entire bones are connected to the custom subclasses of *:Completeness2States*, to establish the connection to the primary skeletal inventory. The reason why there is not only one entire bone class for all bone organs, is that the input of the assays must be a bone segment, and it must be possible to address each of them individually. The lower part of Figure 3.8 shows a custom inventory definition, by new completeness and bone segment classes.

By the study design execution only the assay part will be covered, because the data transformation has exactly the same structure. As it was already mentioned through the example from Figure 3.7, each assay must address a bone segment an input, and a measurement datum as an output. To define a custom study design execution (Figure 3.9) at first a subclass is needed (:SexExstimation), and the different custom assays (denoted with blue) that are connected to each other again with restrictions. Then the outputs of the assays are custom scalar measurement datum classes.

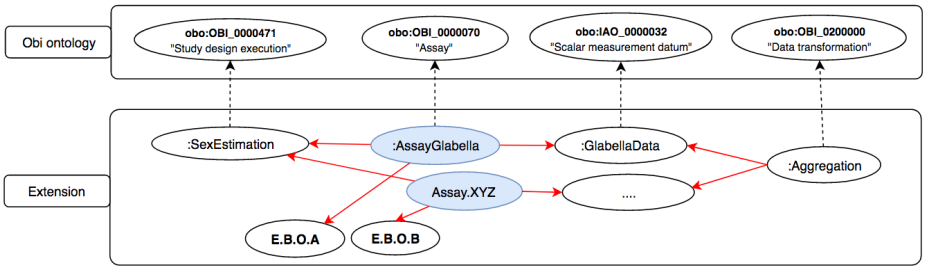


Figure 3.9: Ontology extension for sex estimation

## 3.2 RDF Data input

### 3.2.1 Dynamic data entry forms

In section **??** it was explained that the simplest data input process by RDF data is the substitution of the values coming from the client into a predefined set of RDF triples. In this case the interface is a static HTML form, the input data is a set of key-value pairs, while the RDF triples are defined on the server simply by a string. However in the previous section we have seen that the data models of the problems contain relationships with 1 to n cardinalities. This means that the data input process may contain, above a selection of the type, and setting some of the literal values of one new RDF instance, the dynamic adding of sub forms that represent further instances that are connected to the main instance with cardinality n.

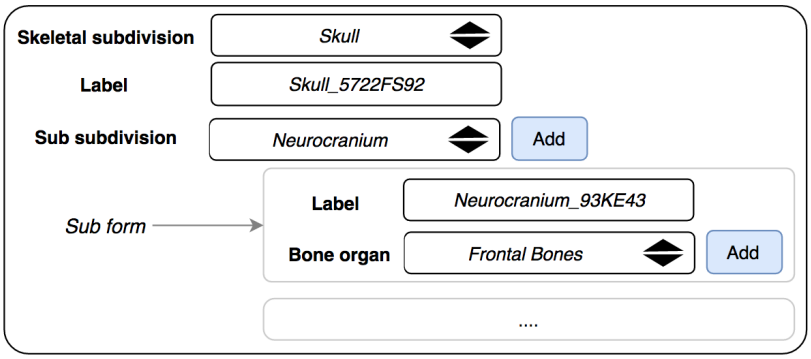


Figure 3.10: Multi dimensional form layout

Figure 3.10 depicts a form which allows to add new sub forms that represent the sud subdivisions of the selected skeletal subdivision. Such functionality can be implemented by means of JavaScript routines embedded into the HTML document. The idea is that there is a main form, which consist of the same type of elements like the static forms, and one of its selector field is equipped with a button that initiates the loading of a new sub form. Then the sub form consists of again the standard elements, and maybe of further sub form adders. In the example the sub sud subdivision form contains an other sub form adder field for the bone organs.

The task of the JavaScript code running on the client is handle these click events and add the new form elements to the HTML document dynamically.

Moreover it has to access the input elements and insert the values into the form data object with defined keys. As there can be multiple sub forms added to the forms their data object stored by means of arrays. Listing 3.1 shows the JSON data (JavaScript Object Notation) of the form in Figure 3.10.

```
var formData = {
  subdivisionType : "fma:5018",
  subDivisionLabel : "Skull_5733FS2",
  systemic_parts : [
    {
      subsubdivisionType : "fma:45720",
      subSubdivisionLabel : "Neurocranium:_93KE43",
    }, { ... } ]
}
```

Listing 3.1: JSON object generated by the form

The server then in turn has to be prepared that a particular subgraph of the data model has to be created multiple times, and the values are arriving in arrays. The following image shows the data model of the entry form.

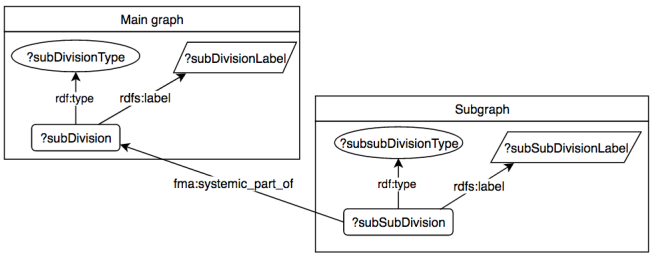


Figure 3.11: Multi dimensional RDF dataset

### 3.2.2 Adoption form elements to the ontology

The previous section showed the basics of forms for multi dimensional data input. An important issue which was just partly addressed in 2.3.3, is how the options of the particular selector fields are loaded. In order to load the options of the initial selector can template files be used, that gets the list of the classes coming from a SPARQL query on the ontology. In the example these are all skeletal subdivisions. But there are elements, for example the second selector field in Figure 3.17, whose values are dependent on the pre-

vious elements, which his means their options must be loaded dynamically only after selector of its ancestor element.

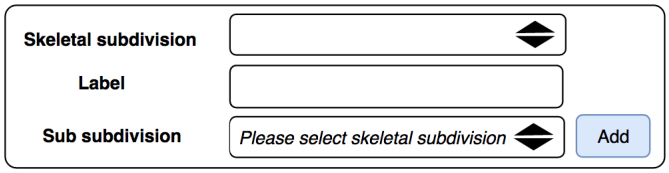


Figure 3.12: Initial state of the form

Therefore their values are not part of the initial HTML document like possibly in the case of the independent skeletal subdivision class selector, but they have to be set as values of JS variables, so that they can be loaded after the selection. This can be achieved the same way by template file routines, that gets the set of sub sudivisions grouped by skeletal division.

```
var subSubDivisions = {
  skull : [{ neurocranium } ... { .. }],
  vertebralColumn : [{ cervical vertebra } ... { .. }],
  ...
}
```

Listing 3.2: Form option data structure in JSON

The data set on the listing 3.2 stores the lists of the sub ubdivision with the corresponding keys (the keys are real IRIs in practice). Such data can be written to JS with template routines by means of two lists, like in listing 2.10. Then if the user selects the first field, JS can access based on the selected value the array to be loaded as options.

But as the example on Figure Figure 3.10 shows that after adding the sub subdivision the bone organs are supposed to be selected to. Therefore the client has to have the same data structure for bone organs as well grouped by the sub sudivsion. The problem is that this data on the form can easily became too large, which may lead to performance issues. Additionally if the whole dataset is queried by the page load then it can be as well inefficient. Therefore it is common a practice that data of the form is loading dynamically through AJAX (Asychronous JavaScript and XML). AJAX is technology that allows the client to communicate with the server without reloading the whole page. This makes the pages more interactive and as well

more efficient.

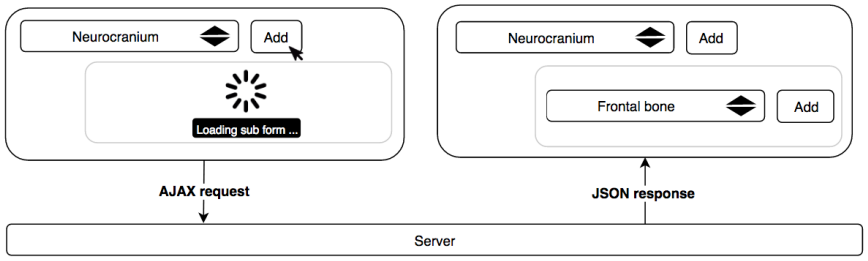


Figure 3.13: Loading form data through AJAX

The idea is that JS is programmed so that upon certain events, a particular request data is assembled and sent to the server. The response then in this case is not a whole HTML page but a dataset in XML or JSON format. After the response arrives, a JS routine continues its. This more elegant solution addresses further JS programming, and of course new server routines, that handles the requests, perform the queries with the incoming parameters, and respond the results.

### 3.2.3 Editing form data

The dataset created by the forms have be browsed and edited as well. This means that the application has to be able to restore the form to the same state as it submitted for initial data creation. This is the other direction of the data flow, bacause here the existing data will be retrieved by SPARQL, whose result comes into a JSON object, based on which the form can reset the its state.

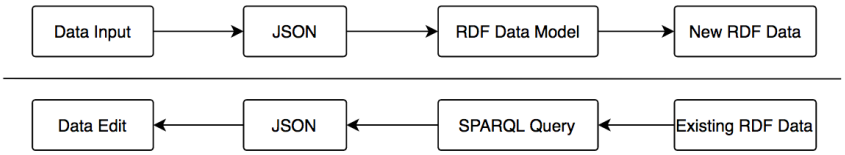


Figure 3.14: Two directions of the data flow

Important issue regarding the server implementation is that in such multi dimensional dataset, it is not sufficient to perform only one query for the whole form data. For example there can be more elaborate assays that have

| ?boneSegment | ?measurementDatum |
|---|---|
| BS1 | MD1 |
| BS1 | MD2 |
| BS2 | MD1 |
| BS2 | MD2 |

Table 3.1: SPARQL result table

multiple inputs and output (i.e Figure 3.15).

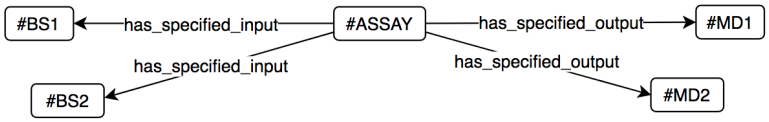


Figure 3.15: Multi dimensional data example

This means if the following query is executed on the dataset, results such a table ((Table 3.1)) that is hard to process.

```
SELECT ?boneSegment ?measurementDatum
WHERE {
  ?assay     :has_specified_input         ?boneSegment .
  ?assay     :has_specified_output        ?measurementDatum .
  FILTER ( ?assay = ${inputParameter})
}
```

Listing 3.3: SPARQL query for the form data

Therefore the data object of the form has to be retrieved gradually, by dividing the data model graph by the predicates, whose cardinality is larger than one. The process have to start with the main graph, and the results of the sub graph queries then has to be stored in arrays of the form data JSON object. The task of the client is to iterate on the arrays and reload the sub forms. Important part of the form reloading is that, not only the values of the literal fields have to be restored, but the selector field options as well. So that for example if the existing dataset contains a skeletal subdivision, it is necessary to load the possible bone organs into the selector, so that the user can add additional bone segments conveniently. Finally if a value of a

literal field has been edited by the user, or new sub forms has to be added or removed, the saving operation should be done immediately through AJAX. To achieve this JS has to assign events to each literal fields and sub form container, and be able to send the appropriate request to the server.

## 3.3 Solution Scheme

### 3.3.1 Model for RDF data input processes

The task of the implementation of a data input forms addresses two main questions. Firstly is what dataset has to be created, and secondly how this dataset is represented on the form. The vocabulary which is capable of defining the whole input process, has to be able to answer both these question. Therefore the model is divided into two sections, to the data definition and the form definition. The data definition consist of a set of RDF nodes and triples, while the interface definition consist of the forms and the data input fields.

Nevertheless we have the previous chapters we have seen that the scheme of the ontology plays an important role of the data entry form process. Namely if a set restriction statement assign multiple classes to one, like i.e. bone organs to a skeletal subdivisions, then the web user interface has to be adapted to this scheme, and it has to of the possibility to add sub forms dynamically. So to make the vocabulary complete is it inevitable to expresses what triples may be added multiple times and of which form elements adds further sub forms.

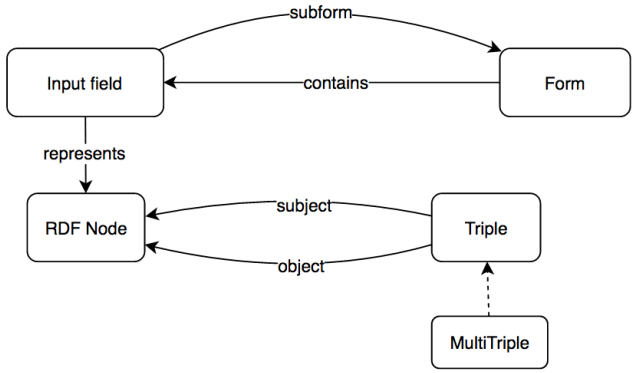


Figure 3.16: Vocabulary for RDF data input processes

Figure Figure 3.16 shows the main elements and relationships of the vocabulary. It can be seen that the connection between the data and the form definition is established through the represents predicate that connects form elements to RDF nodes. Moreover the MultiTriple is a subclass of the

triple, and it that allows to expresses the scheme of the ontology subset, for which the RDF data will be created.

### 3.3.2 Software framework outline

The above introduced vocabulary is represented currently by means of Java classes. Thus the definition of the problems is the creation of Java objects from the classes.

```
//Data definition
List<Triple> triples = new ArrayList<Triple>();
RDFNode subdivision = new RDFNode("subdivision");
RDFNode boneOrgan = new RDFNode("boneOrgan");
triples.add(new MultiTriple(boneOrgan, "fma:systemic_part_of",
    subdivision));
...
//Form definition
Form form = new Form();
form.addElement(new LiteralField("label"));
...
//FormConfiguration
FormConfig skeletalInventory = new FormConfig(form, triples);
```

Listing 3.4: Input process definition in Java

The idea is that framework operates completely upon this definition. The first task is to process the definition and generate JSON objects that describes the forms layout, and further Java objects for the data operations.

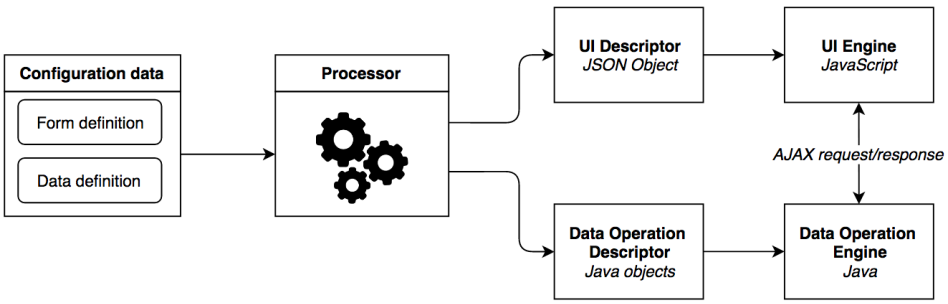


Figure 3.17: Implementation workflow

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-

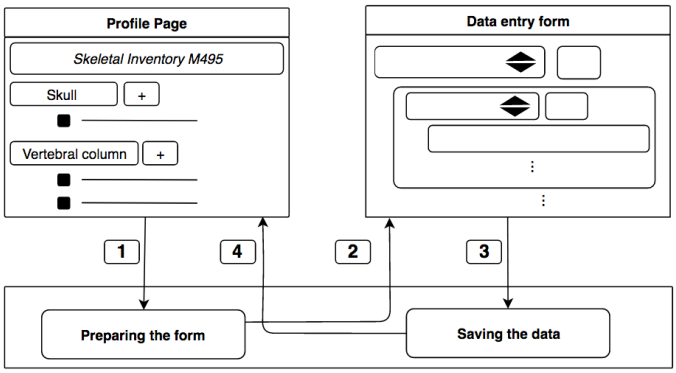


Figure 4.1: Complete workflow of data input process

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.

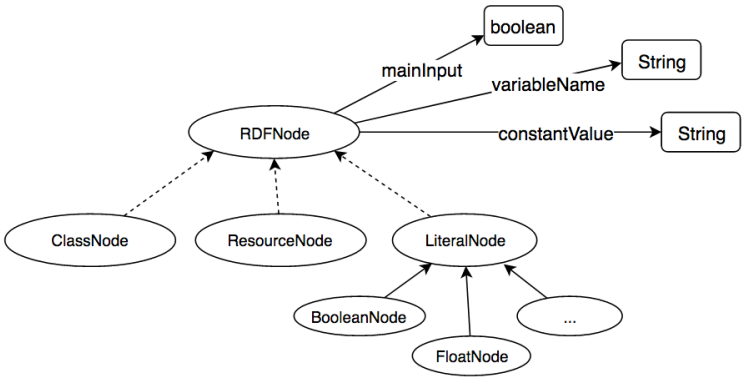


Figure 4.2: Variable types and their attributes

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 re-

striction 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.

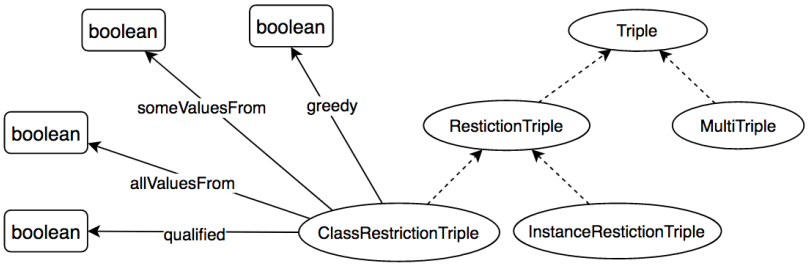


Figure 4.3: Triple types

### 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.

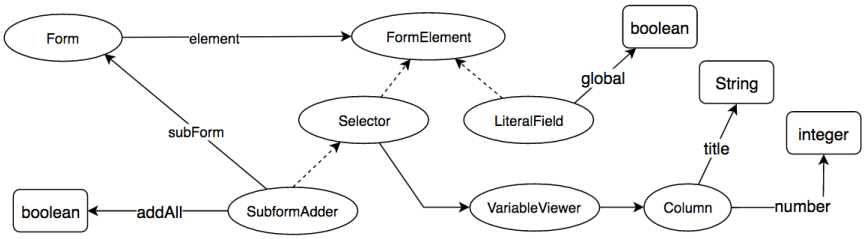


Figure 4.4: Form definition

## 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 3.3.1. 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.

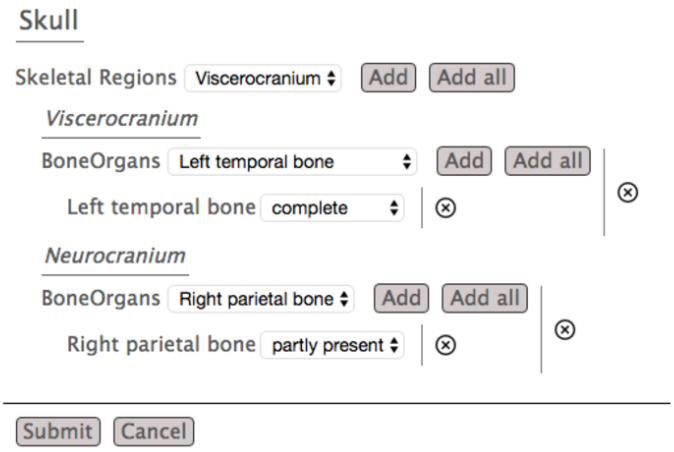


Figure 4.5: Input form for skeletal inventories

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

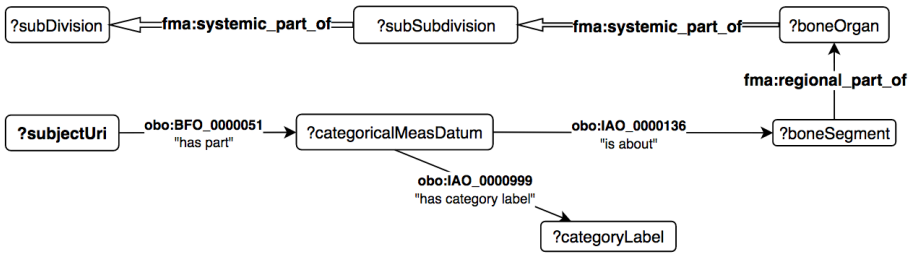


Figure 4.6: Skeletal inventory data triples

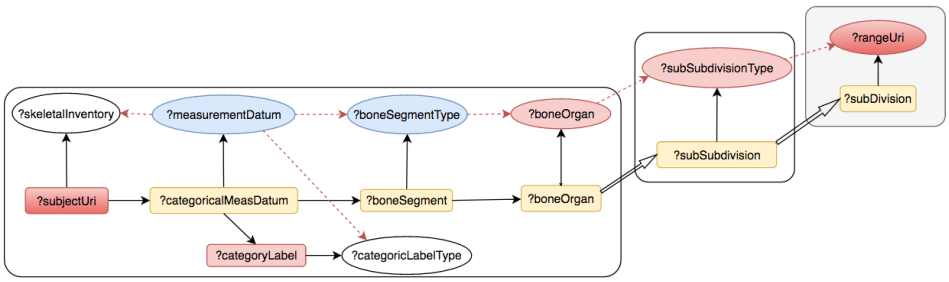


Figure 4.7: Complete data definition

the structure which is followed by the form as well.

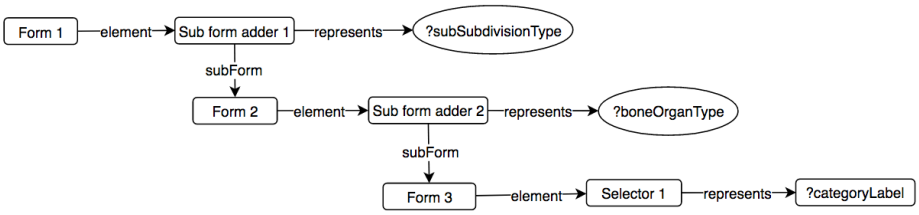


Figure 4.8: Form layout definition

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.

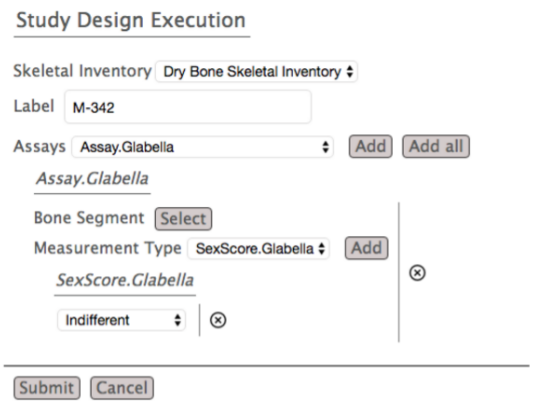


Figure 4.9: Input form for study design execution

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.

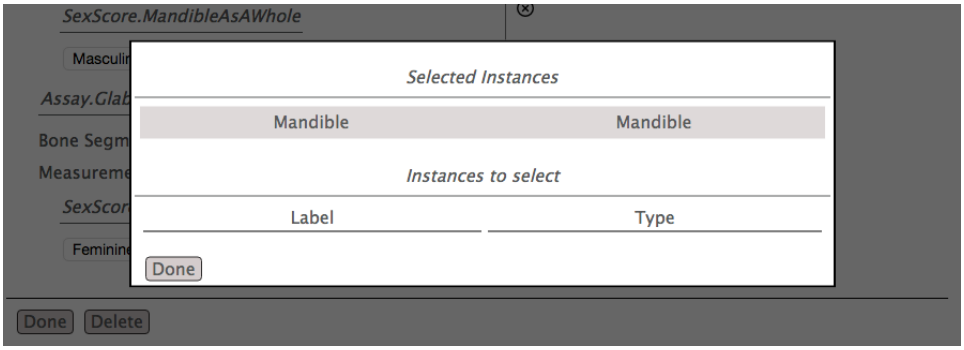


Figure 4.10: Instance selector for existing bone segment

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.

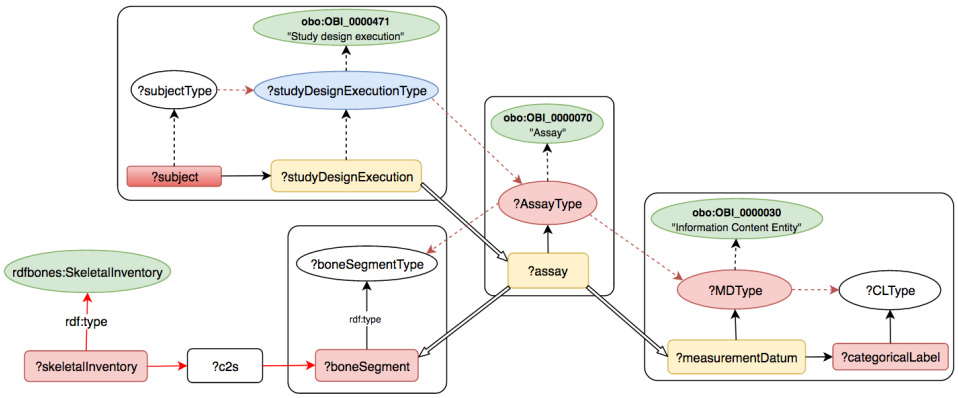


Figure 4.11: Complete data model

- 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 3.17 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.

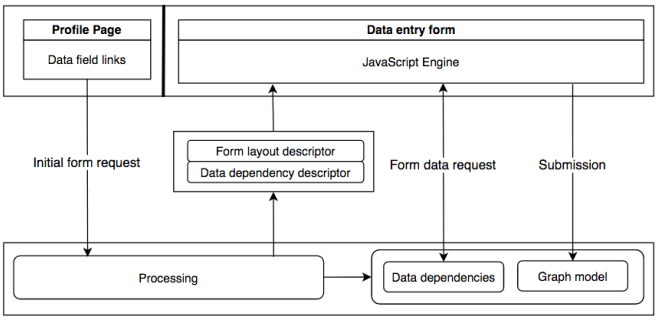


Figure 5.1: More detailed scheme

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.

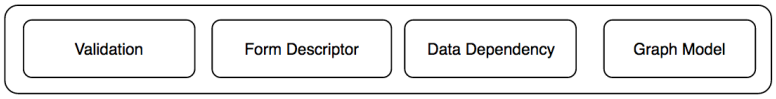


Figure 5.2: Processor tasks

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.

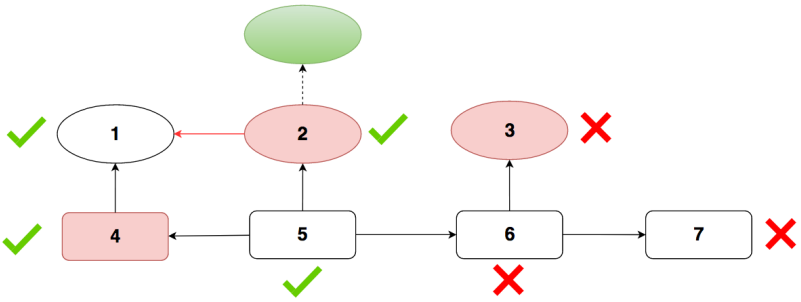


Figure 5.3: Valid and invalid nodes

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.

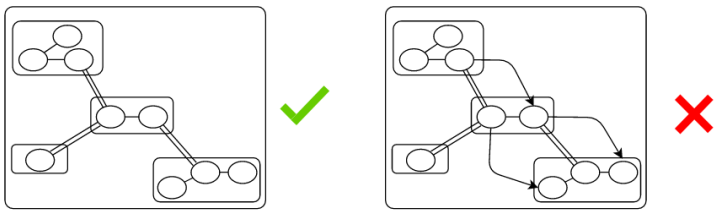

Figure 5.4: Valid and invalid graph

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 descriptor generation

The previous section outlined the rules of the data model. This and the following section although discusses the scheme of the object and the main principles of how the client and the server operates upon it. The use-cases of the previous chapter showed that input nodes of the form are dependent from the main input nodes and from each other as well. The task is to infer from the graph model the variable dependencies, that is build by the triples of the participant. Each variable dependency consists of the set of input nodes and triples, and one output node.

Important that form elements on the form has a specific order, and the rule is that a node cannot be dependent on an other node which comes in the form after it, because its value is not available at the moment when the options for the node before it has to be loaded, thus it cannot contribute to the SPARQL query with its value. However each node is dependent on the main input nodes, because their value is given before the form loading.

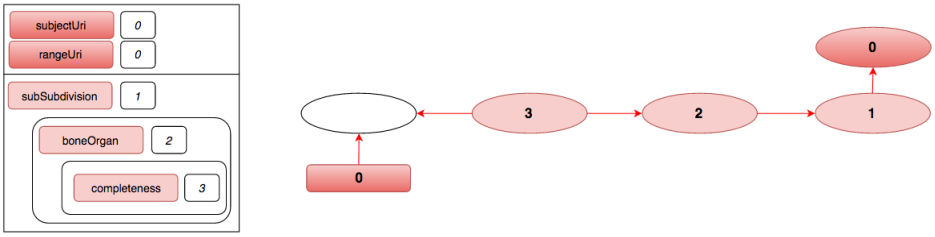


Figure 5.5: Dependency

### 5.1.3 Graph model generation

- Graph model
- New instances
- Delete
- Existing data loading

## 5.2 Implementation

This request arrives to the server and based on the parameter *predicateUri* the application finds the configuration triples. More details about how this parameter value is assigned to the configuration setting will be provided in the next chapter.

As it was described in the previous chapter the whole process starts from the pages for data display with the HTTP requests initiated by the links of the data fields.

### 5.2.1 Server side

- This section introduces how application on the server is able to operate based on data queried from the configuration dataset

#### Overview

- The following image shows the most important steps of the data input process

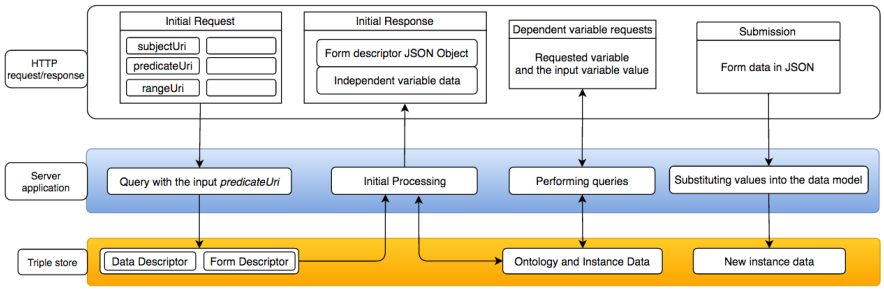


Figure 5.6: Overview of the mechanism on the server

#### Form representation in Java

- The literal fields asks for the type of the variable it represents and the type of the descriptor will be based on this literal field.

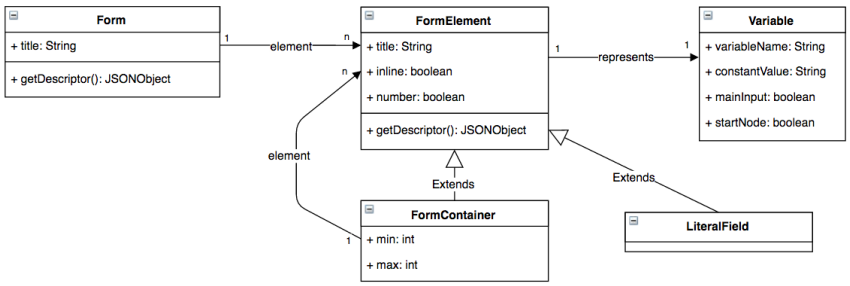


Figure 5.7: UML Diagram of the classes for the form

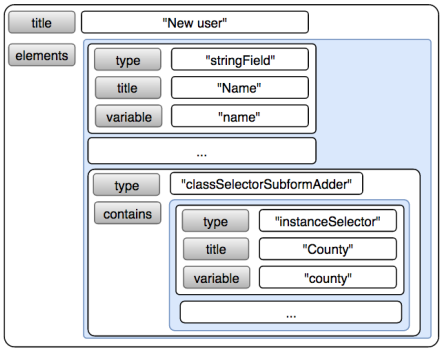


Figure 5.8: Form descriptor JSON object

### Data model in Java

- Querying the configuration triples regarding statements
- Processing into the tree structured graph model

Validation:

- Data dependencies can be over graphs
- But graphs can be connected only through multi statements
- The graph has to be a tree. There are no use cases right now where any loop would be required

- Data saving mechanism explanation
- Data retrieval mechanism explanation

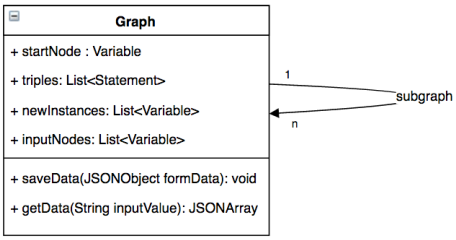


Figure 5.9: Graph UML

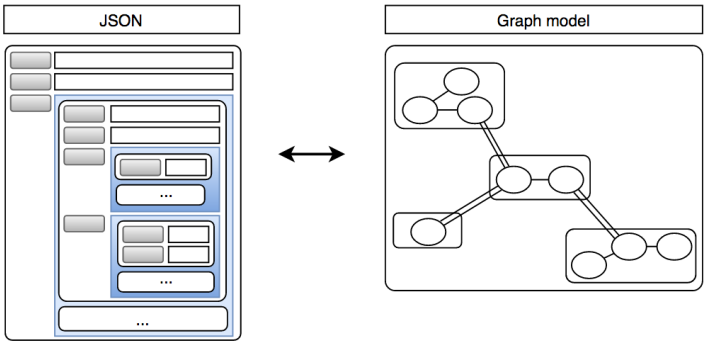


Figure 5.10: JSON vs graph model

### Data Dependencies

- The data dependencies are important only for the form.
- Everything starts with the form descriptor.
- There are cases where from the main form no element appears on the form.
- A have to bring examples to some problems that illustrate the problem.
- Here comes at first the ontology awareness into question....

where,

- Post processing for different restriction types

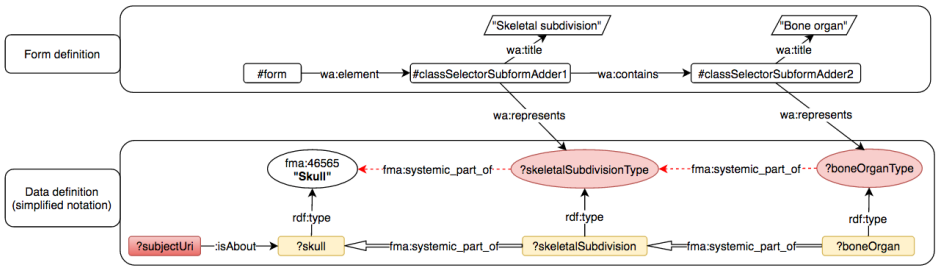


Figure 5.11: Example problem

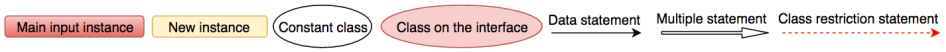


Figure 5.12: Elements of the simplified notation

#### Editing and deleting data

- This feature was not introduced on the overview image but it is really important.

### 5.2.2 Client side

- In this chapter it is discussed how the forms are implemented in JavaScript
- Each subsection contains code examples that gradually introduce the functionalities
- Codes are mostly simplified to facilitate the understanding

#### Object oriented JavaScript

- The task of code in on the interface is to operate based on a configuration data dynamically.
- There two subtasks, the generation of the UI elements (i.e input fields, buttons, etc.) and manage the data input and display on the form
- To solve these problem, an object oriented approach is applied
- This means that there are classes that handles both the UI and the data related tasks


Figure 5.13: Two restriction directions

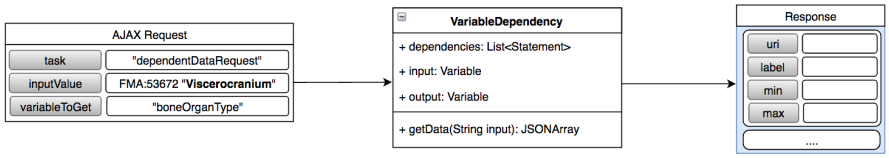


Figure 5.14: Getting dependent data

- See an example for the class definition in JS

```
class StringField {
constructor(...){
this.container = $("<div/>")
...
}
someMethod(){...}
}
```

Listing 5.1: JavaScript class

- Each form elements are represented by such objects, and the form loading based on the descriptor runs by the initialisation of these elements.

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

Listing 5.2: Form generation based on configuration data

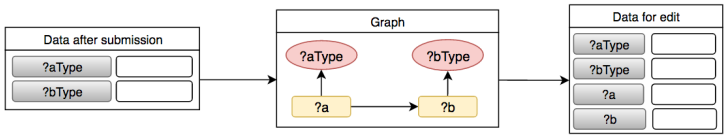


Figure 5.15: Difference between the submissions and edit data on the form

- Explanation of the code ...
- This is the way how it is possible to generate interfaces

### Handling data

- Above the element generation for the different forms it is necessary to handle of course the data based on the descriptor
- Each form element's descriptor contains a field called dataKey. The value of this field will be the key of data in the form data object.

```
class StringField {
constructor(descriptor, formData){
this.descriptor = descritor
this.formData = formData
this.inputField = $("<input/>").change(this.handler)
}

handler(){
this.formData[this.descriptor.dataKey]=this.inputField.val()
}
}
```

Listing 5.3: Data saving

- The previous code illustrates how the form element object set the global form data field based on configuration data
- Further details about the code...
- Handling existing data

- It can be the case that form is loaded for editing. Then in this case the formData variable coming as input to the constructor contains the value for the dataKey?

```
class StringField {
constructor(descriptor, formData){
if(formData[descriptor.dataKey] != undefined){
this.editMode = true
}
}

handler(){
if(this.editMode){
var oldValue = this.formData[this.descriptor.dataKey]
var newValue = this.inputField.val()
}
AJAX.updateField(oldValue, newValue)
}
}
```

Listing 5.4: Data saving

### Sub form adders

- Descriptor of the sub form adder contains a field called subform
- Then the initialisation of the form happend through the Form class.
- This is used by the initial form loading as well

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

add(){
var subformDataObject = new Object()
this.formData[this.descriptor.dataKey].
push(subFormDataObject)
```

```
this.subFormContainer.append(
new Form(this.subFormDescriptor, subFormDataObject))
}
}
```

Listing 5.5: Sub form adder routine

- Important for each subform a new JSON object is generated which is pushed to the array
- The same way the subform adder checks if the this.formData[this.descriptor.dataKey] contains already existing data and adds them if they are there

### 5.2.3 Data dependency

### 5.2.4 Form validation

Chapter 6

# Evaluation

## 6.1 Expressivity of the framework

- Most of the problems of the project were solved by the higher level language
- Thus the code of the application could be significantly reduced
- This enables a more concise and readable documentation, which is a really important issue by such open-source projects.

## 6.2 Reduced amount of code

- As the configuration data is in the form of RDF triples as well, they could be added through web interfaces to accelerate the development.
- The most important advantage of the framework that the web application ontology can describe the input forms for itself.
- Another utility that in such case not only the data can be exchanged between institutes using RDF data, but the application descriptor as well.

## 6.3 Limitations of the system

Chapter 7

# Conclusion and future work

## 7.1 Summary of achievements

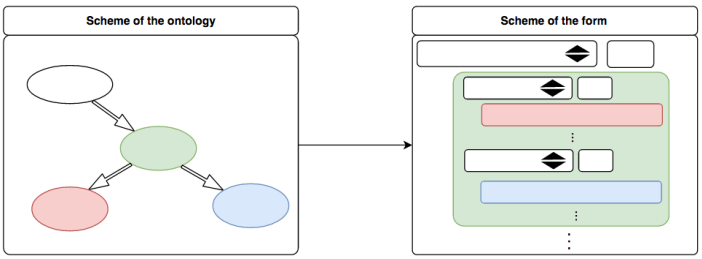


Figure 7.1: Scheme of scheme

## 7.2 Future work

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] OWL 2 web ontology language document overview. W3C recommendation, W3C, October 2009. http://www.w3.org/TR/2009/REC-owl2-overview-20091027/.

[2] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. On the semantics of sparql. In *Semantic Web Information Management*, pages 281–307. 2009.

[3] 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(4):1–19, 04 2016.

[4] 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.