

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

Abstract in English

# Kurzfassung

Kurzfassung auf Deutsch

# Contents

<b>Abstract</b>	<b>II</b>
<b>Kurzfassung</b>	<b>III</b>
<b>List of Tables</b>	<b>X</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Initial goal and contributions . . . . .	1
1.2 Thesis outline . . . . .	1
<b>2 Preliminaries</b>	<b>2</b>
2.1 Web applications . . . . .	2
2.1.1 Introduction . . . . .	2
2.1.2 JavaScript . . . . .	5
2.1.3 AJAX . . . . .	6
2.2 Semantic Web . . . . .	7
2.2.1 RDF . . . . .	7
2.2.2 RDF Schema . . . . .	9
2.2.3 OWL . . . . .	10
2.2.4 SPARQL . . . . .	12
2.3 VIVO Framework . . . . .	13
2.3.1 Class groups . . . . .	14
2.3.2 Profile Pages . . . . .	15
2.3.3 Default Data Entry Forms . . . . .	16
2.3.4 Custom Entry Forms . . . . .	17

<b>3</b>	<b>Problem Statement</b>	<b>19</b>
3.1	Challenges of the RDFS project . . . . .	19
3.1.1	Human skeleton . . . . .	19
3.1.2	Ontology Extensions . . . . .	20
3.1.3	Study Design Execution . . . . .	22
3.2	RDF Data input . . . . .	23
3.2.1	Multi dimensional form . . . . .	23
3.2.2	Existing instances and dependencies . . . . .	24
3.2.3	Instance browsing . . . . .	25
3.2.4	Sub form dependencies . . . . .	26
3.2.5	Validation . . . . .	26
3.2.6	Saving data . . . . .	26
3.2.7	Editing data on the forms . . . . .	26
<b>4</b>	<b>Implementation</b>	<b>27</b>
4.1	Web application ontology . . . . .	27
4.1.1	Introduction . . . . .	27
4.1.2	Form definition . . . . .	28
4.1.3	Data definition . . . . .	29
4.1.4	VIVO adoption . . . . .	32
4.2	Server-side implementation . . . . .	33
4.2.1	Overview . . . . .	33
4.2.2	Form representation in Java . . . . .	34
4.2.3	Data model in Java . . . . .	34
4.2.4	Data Dependencies . . . . .	35
4.2.5	Editing and deleting data . . . . .	36
4.2.6	Navigator . . . . .	36
4.3	Client-side implementation . . . . .	37
4.3.1	Object oriented JavaScript . . . . .	38
4.3.2	Handling data . . . . .	39
4.3.3	Sub form adders . . . . .	40
4.3.4	Data dependency . . . . .	41
4.3.5	Form validation . . . . .	41

<b>5</b>	<b>Evaluation</b>	<b>42</b>
5.1	Problems of the RDF Bones Project . . . . .	42
5.2	Self configurable web application . . . . .	42
5.3	Limitations of the system . . . . .	42
<b>A</b>	<b>Glossary</b>	<b>44</b>
<b>B</b>	<b>Appendix</b>	<b>49</b>
B.1	Something you need in the appendix . . . . .	49

## List of Figures

2.1	Client server communication . . . . .	2
2.2	HTML Link . . . . .	3
2.3	Servlet mapping definition in web.xml . . . . .	3
2.4	Data flow . . . . .	4
2.5	Template file . . . . .	4
2.6	SQL query with parameter . . . . .	4
2.7	Form layout and HTML document . . . . .	5
2.8	Request with parameters . . . . .	5
2.9	Example Java routine for data storage . . . . .	5
2.10	Simple JavaScript example . . . . .	6
2.11	Loading new element through AJAX . . . . .	6
2.12	Main structure of the RDFS vocabulary . . . . .	9
2.13	RDFS domain and range definition . . . . .	10
2.14	RDFS domain and range definition . . . . .	10
2.15	A subset of OWL vocabulary . . . . .	11
2.16	OWL object properties . . . . .	12
2.17	Properties for qualified cardinalities . . . . .	12
2.18	Document class group on the admin panel . . . . .	14
2.19	VIVO class group page for documents . . . . .	14
2.20	RDF configuration data in VIVO . . . . .	15
2.21	VIVO profile page layout . . . . .	15
2.22	Triples contributing to the displayed profile page layout . . . .	16
2.23	HTTP request for data entry form . . . . .	16



---

2.24	Object property entry form for bibo:author . . . . .	17
2.25	Data property entry form . . . . .	17
2.26	Data definition graphical (left) and lexical (right) . . . . .	18
2.27	Definition of custom entry form configuration class . . . . .	18
3.1	Ontology structure for skeleton . . . . .	20
3.2	FMA scheme for skull . . . . .	20
3.3	RDF Triple representation of a skull . . . . .	21
3.4	Skeletal Inventory Data Structure . . . . .	21
3.5	Ontology extension scheme . . . . .	22
3.6	Glabella and its expressions . . . . .	22
3.7	Study Design Execution Data Structure . . . . .	22
3.8	Ontology extension for Glabella . . . . .	23
3.9	Single vs. multi dimensional form . . . . .	24
3.10	JSON object for multi dimensional forms . . . . .	24
3.11	Data model . . . . .	25
3.12	Instance selection and dependency . . . . .	25
3.13	Navigator example . . . . .	25
4.1	Basic workflow . . . . .	27
4.2	Framework functionality . . . . .	28
4.3	Web application ontology for the form . . . . .	28
4.4	Difference between sub form adders . . . . .	29
4.5	Form descriptor RDF Data . . . . .	29
4.6	Statement in RDF Vocabulary . . . . .	30
4.7	Core ontology . . . . .	30
4.8	Variable types . . . . .	31
4.9	Statement types and attributes . . . . .	31
4.10	Statement configuration dataset I . . . . .	32
4.11	Statement configuration dataset II. . . . .	32
4.12	Form definition dataset . . . . .	33
4.13	Base definition of the data input process . . . . .	33
4.14	Overview of the mechanism on the server . . . . .	33
4.15	UML Diagram of the classes for the form . . . . .	34
4.16	Form descriptor JSON object . . . . .	34
4.17	Graph UML . . . . .	35

---

4.18 JSON vs graph model . . . . .	35
4.19 Example problem . . . . .	36
4.20 Elements of the simplified notation . . . . .	36
4.21 Two restriction directions . . . . .	36
4.22 Getting dependent data . . . . .	37
4.23 Difference between the submissions and edit data on the form	37
4.24 Selector VS form graph . . . . .	38
4.25 Navigator class diagram . . . . .	38

## List of Tables





# Chapter 1

## Introduction

Introduction.

You can reference the only entry in the .bib file like this: [1]

### 1.1 Initial goal and contributions

### 1.2 Thesis outline

# Chapter 2

## Preliminaries

### 2.1 Web applications

#### 2.1.1 Introduction

Usually web applications do not consist of one single page, but of several different pages. 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.

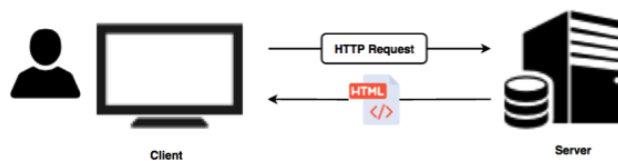


Figure 2.1: Client server communication

Usually web applications do not consist of one single page, but of several different pages. 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>
```

Figure 2.2: HTML Link

The link in Figure 2 shows the link for the subpage. Programming the server incorporates the task of assignment of the url-mappings to particular classes, which are responsible for the response preparation. In Java web application these responder classes are called servlets and the definition of the mapping-class assignment looks as follows:

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

Figure 2.3: Servlet mapping definition in web.xml

A modern web application do not just send static web pages to the client, that contains in this case the political articles, but the articles are stored in a database, and the pages are generated dynamically by substituting the retrieved data into so-called template files. 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.

In our example the news are stored in the NEWS table and the application has a class named News with the same attributes that the table has. Consequently one single row of the table can be stored in an instance of the News class. As the database returns a table with multiple news, thus the resulting Java data will have the type `List<News>`. Then this in this simple case the list is passed to the template engine together with the template file.



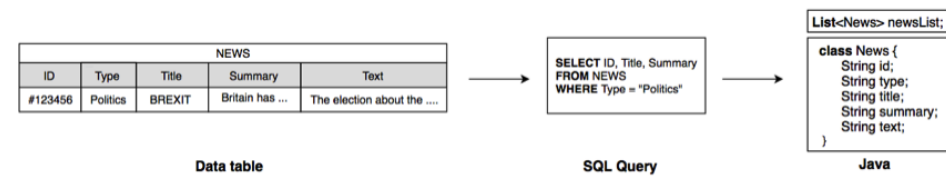


Figure 2.4: Data flow

```
<#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>
```

Figure 2.5: Template file

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.

The template file contains the following link

```
<a href="http://newsPortal.com/wholeNews?id=${news.id}" >Read more <a>
```

which redirects to the page where the whole article can be seen. To achieve this it is necessary to equip each link with the parameter `id` that holds the ID of `NEWS` item, order to inform the server about which article's whole text has to be displayed. Then the servlet class of the mapping `/wholeNews` has to perform the following query where the `id` is the input.

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

Figure 2.6: SQL query with parameter

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.

Form layout

Title

Type

Summary

Text

Submit

HTML Document

```
<html>
<form action="webApp/newArticleController">
  Title <input type = "text" name = "title"> <br>
  Type <input type = "text" name = "type"> <br>
  Summary <input type = "text" name = "summary"> <br>
  Text <input type = "text" name = "text"> <br>
  <input type = "submit" value = "Submit">
</form>
</html>
```

Figure 2.7: 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.

`"http://newsPortal.com/newArticleController?title=France won the EC&type=Sport&summary= ...."`

---

Figure 2.8: Request with parameters

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

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

Figure 2.9: Example Java routine for data storage

### 2.1.2 JavaScript

JavaScript (JS) is the programming language of the web browser. A JS code can be embedded into any HTML document between `<script></script>` tags. The most fundament capability of JS, is that it is capable of manipulating the elements of the web page. The following example illustrates a simple case, where clicking a button can change the page by adding a new div to an other div.

The HTML page contains with two divs with the id-s button and container. JavaScript handles each element on the page as objects. These objects can be referenced by `$("#id")` where the id is the id parameter of

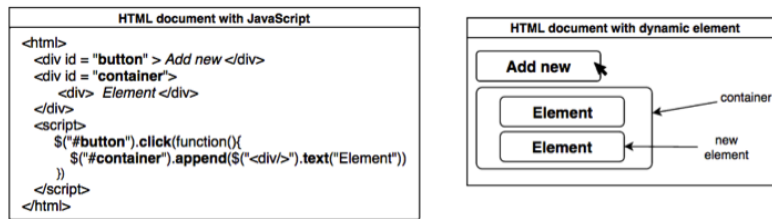


Figure 2.10: Simple JavaScript example

the html tag. Thus the definition of a click event to the first div is done by writing the following code:

`$("#button").click( ... the handler function ... )`

to the script. The added function defines only one single operation, which uses the append function on the `$("#container")` div object. The input parameter is a new div object, created by JS with the text value Element.

### 2.1.3 AJAX

AJAX is an abbreviation for Asynchronous JavaScript And XML. This is a technology that allows the web browser to exchange data with the server without reloading the whole page. AJAX calls are initiated from JavaScript and of course JS itself is responsible for handling the response. The following example shows an AJAX based solution for loading the whole text of an article.

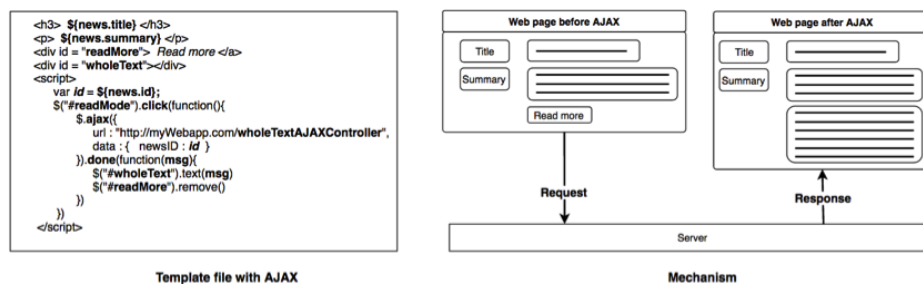


Figure 2.11: Loading new element through AJAX

The example from the image illustrates extends the previous case so that the click function contains the AJAX call. This call is practically the same as

basic request from the `<a/>` tag in HTML. It has a URL and a data object. The data object in this example consists of only one key-value pair, with the key `newsID`. The value is the JavaScript variable `id`, whose own value was set at the beginning of the script part by `$news.id` template variable. This is the way that Java variables can be passed the JavaScript variables. The `done` function of the AJAX routine defines what has to be done with the data that arrives. The response data coming from the server is accessible in the `msg` variable. In the example we assume the server return only the string of the whole text, which will be set as the text of the new div.

## 2.2 Semantic Web

### 2.2.1 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:

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

The actual data about a person:

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



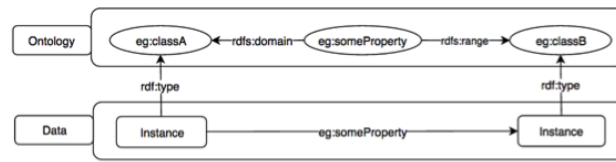


Figure 2.13: RDFS domain and range definition

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.

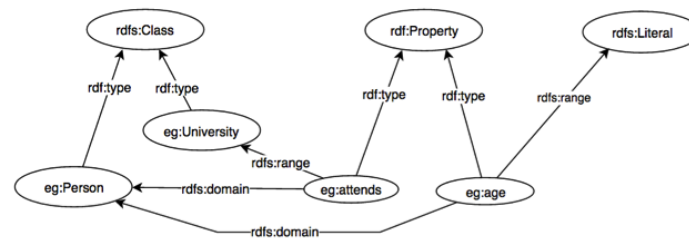


Figure 2.14: RDFS domain and range definition

### 2.2.3 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.

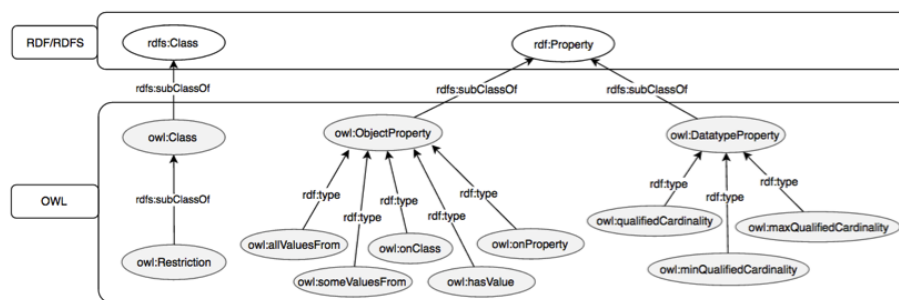


Figure 2.15: A subset of OWL vocabulary

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.

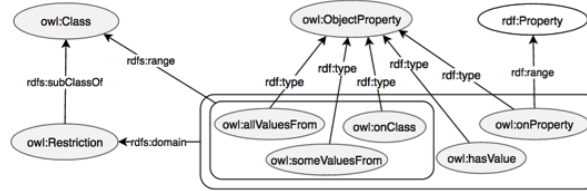


Figure 2.16: OWL object properties

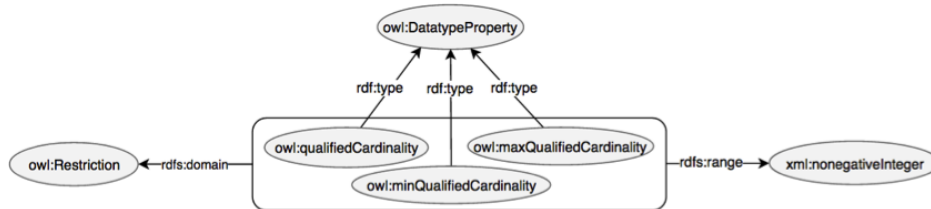


Figure 2.17: Properties for qualified cardinalities

### 2.2.4 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

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

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

## 2.3 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 too, and describes 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. Finally there is a possibility to import any RDF file to VIVO's triple store.

### 2.3.1 Class groups

One important feature of the VIVO framework is the possibility to order the classes of the ontology into so-called class groups. If a class is assigned to a class group then it appears in the list of the admin panel, which is used to select the type of the new instance.

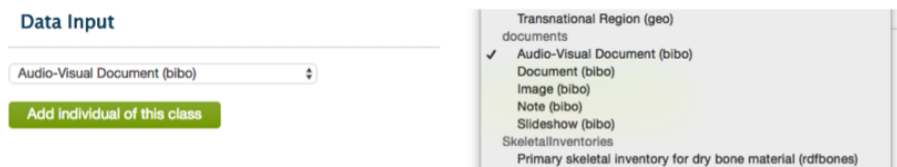


Figure 2.18: Document class group on the admin panel

Further possibility of class groups that it is possible to create links on the main menu (can be seen on the top of Figure 16), which redirects to a page where all the instances are listed that belong to one of the classes of the class group.

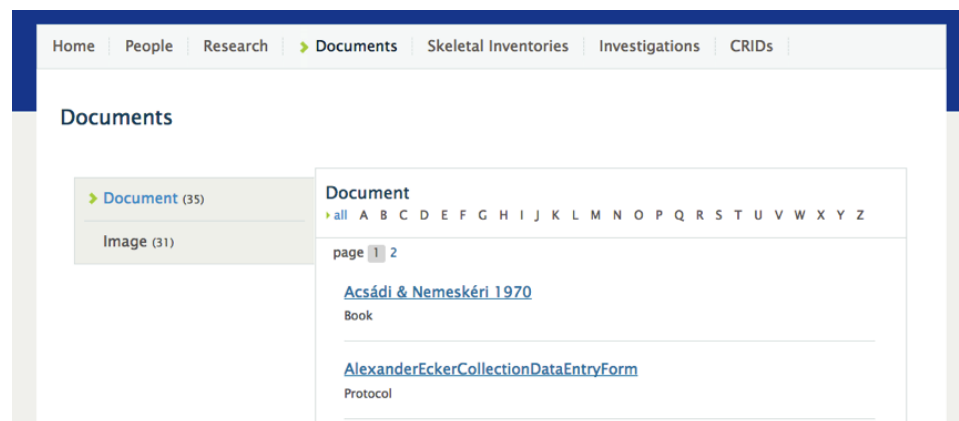


Figure 2.19: VIVO class group page for documents

The application configuration showed in the last two images is defined by the following set of configuration triples.

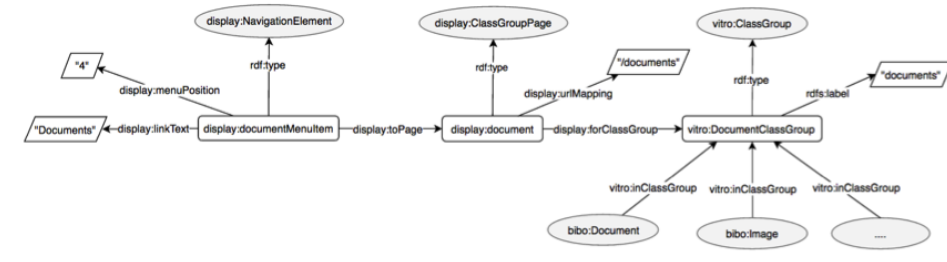


Figure 2.20: RDF configuration data in VIVO

There are three instances of the classes `display:NavigationElement`, `display:ClassGroupPage` and for `vitro:ClassGroup`. The triples itself are self-explanatory, but important to note that property `vitro:inClassGroup` is the one that connects the configuration dataset to the domain ontology.

### 2.3.2 Profile Pages

A profile pages in VIVO displays information about a particular RDF instance. These pages can be reached from the list on the class group pages. The profile page organizes the information into tabs. Each tab displays the properties of a specific property group. The next image shows a screenshot from the profile page under the Overview tab.

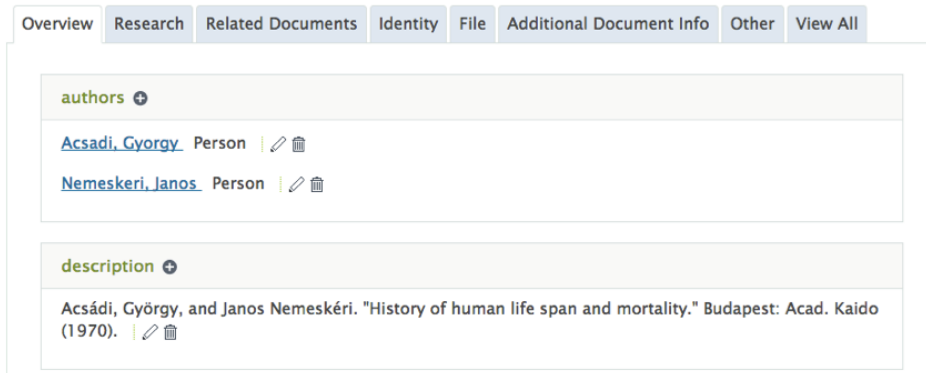


Figure 2.21: VIVO profile page layout

As it was already addressed this page adopts to the ontology by querying the properties whose domain or range is the type of the instance to display. The properties `bibo:author` and `vivo:description` are assigned to the property

group overview, and they appear on the page only because the both have the domain `bibo:Document` class.

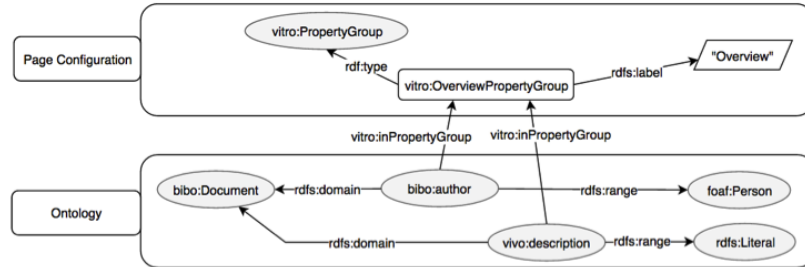


Figure 2.22: Triples contributing to the displayed profile page layout

### 2.3.3 Default Data Entry Forms

On Figure 21, next to the predicate labels (authors, description) there are plus image elements, which are a links. These links redirect the user to data entry forms where new triple can be added.

/editRequestDispatch	
<b>subject</b>	<a href="http://vivo.mydomain.edu/individual/n794">http://vivo.mydomain.edu/individual/n794</a>
<b>predicate</b>	<a href="http://purl.org/ontology/bibo/author">http://purl.org/ontology/bibo/author</a>

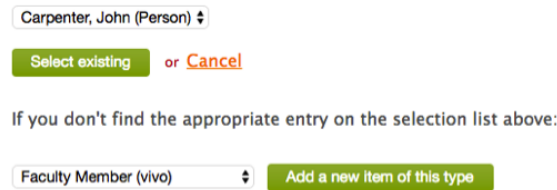
Figure 2.23: HTTP request for data entry form

They initiate the HTTP request depicted on Figure 23. The server gets with which subject and predicate the triple has to be created.

The subject of the triple is the instance; from whose the profile page the request has been initiated. The predicate is the property to which the link belongs. The data entry forms allows the user to set the object of this triple. By the property `bibo:author` the domain is the class `foaf:Person`, thus application offers each existing instance of this class to select, or allows to add a new instance as an object.

In the case of the property `vivo:description`, the domain is the class `rdfs:Literal` thus the entry form displays a text input field.

### Add an entry of type Person for Acsádi & Nemeskéri 1970



Carpenter, John (Person) ▾

Select existing or [Cancel](#)

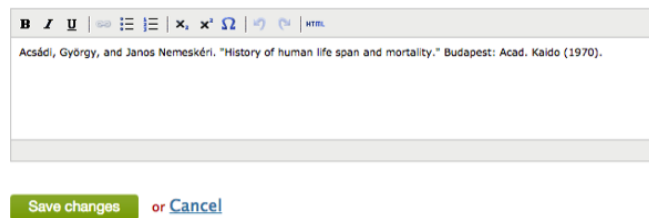
If you don't find the appropriate entry on the selection list above:

Faculty Member (vivo) ▾ Add a new item of this type

Figure 2.24: Object property entry form for bibo:author

### Change text for: description

An account of the resource.



**B** *I* U | | | | [HTML](#)

Acsádi, György, and Janos Nemeskéri. "History of human life span and mortality." Budapest: Acad. Kaido (1970).

Save changes or [Cancel](#)

Figure 2.25: Data property entry form

## 2.3.4 Custom Entry Forms

VIVO allows the editing of the triples through default entry forms only one by one. However it is often the case that it desired to add multiple triples, thus larger dataset by one entry form. This is as well possible in VIVO through custom entry form definition. Let assume an entry form, which let the user add new publications to person instance. About the publication its title, abstract and the date of publishing can be stored. The left part of following image shows dataset of the example. The red nodes denote the variables that are coming as input from the entry form; the green means that its value has to be an unused IRI, and the grey stands for constants.

The variable `?subject` is the instance from whose profile page entry form was called. To declare the information held by the graphical representation of the triples from Figure 25, three static Java variables are needed. Two arrays of string for the inputs and new resources (`literalsFromRequest`, `newResources`) and a string for the triples (`tripleString`). Moreover the con-

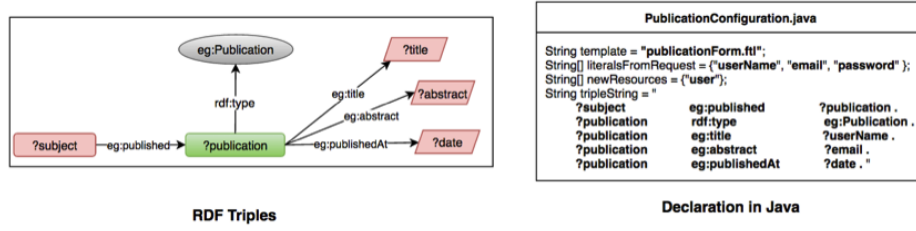


Figure 2.26: Data definition graphical (left) and lexical (right)

figuration class has an additional variable for defining the template file for the form layout (VIVO uses Freemarker template engine, and the .ftl extension stands for Freemarker Template File). The last step towards the definition of the custom entry form is to connect the property `eg:Published` with the predicate `vitro:customEntryFormAnnotation` to the literal value that holds the name of the entry form configuration class.



Figure 2.27: Definition of custom entry form configuration class

# Chapter 3

## Problem Statement

### 3.1 Challenges of the RDFBones project

#### 3.1.1 Human skeleton

The subject of the anthropological investigations is primarily human skeletal remains. To be able to create data about these remains, first of all an ontology is required. As the human skeleton is complex, the ontology is not developed by us, but an existing have been taken. The used ontology is the subset of FMA (Foundation Model of Anatomy) ontology. For us the two most important classes are the following:

- Subdivision of skeletal system - fma:85544
- Bone Organ – fma:5018

The class Bone Organ is the superclass of all bones in the human skeleton. Each bone belongs to a skeletal subdivision and a skeletal subdivision can be a part of another skeletal subdivision. This relationship in both case is expressed by the property fma:systemic\_part\_of. To define which bone organ belongs to which skeletal subdivision FMA uses OWL restrictions.

The most important skeletal subdivision for our project is the skull. Skull has the peculiarity that it consists not directly of bones but two of other subdivisions, which consists of the Bone Organ subclasses.

The following image illustrates then the data structure of skull.



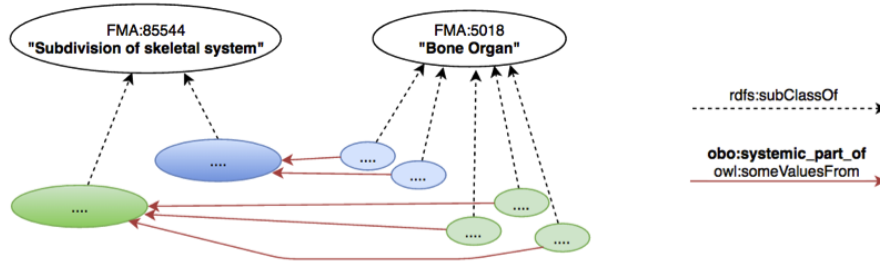


Figure 3.1: Ontology structure for skeleton

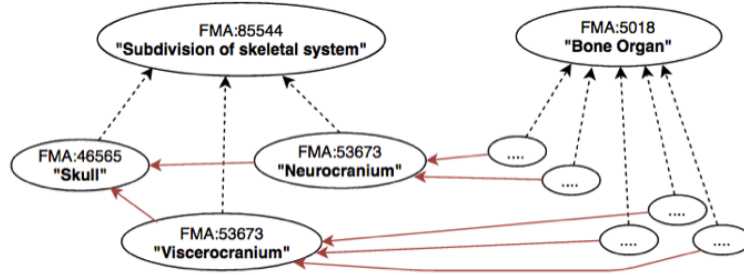


Figure 3.2: FMA scheme for skull

To implement an entry form that allows the user to create such triple set takes considerably more effort than the cases explained in the previous sections, because not only key value pairs have to be sent from the client to the server, but a multi dimensional dataset.

### 3.1.2 Ontology Extensions

It is often the case that in an investigation not only the bone itself, but also particular segments have to be addressed. However the bone segments of the bones are not standard, and they can differ according to researcher or research project. Therefore FMA do not contain any bone segment of the bone organs, and consequently we have to define it on our own. Important that the skeletal subdivision instances do not appear on the dataset on their own, but they are connected to Skeletal Inventories. Skeletal inventories are used to gather information about particular skeletal remains. The following

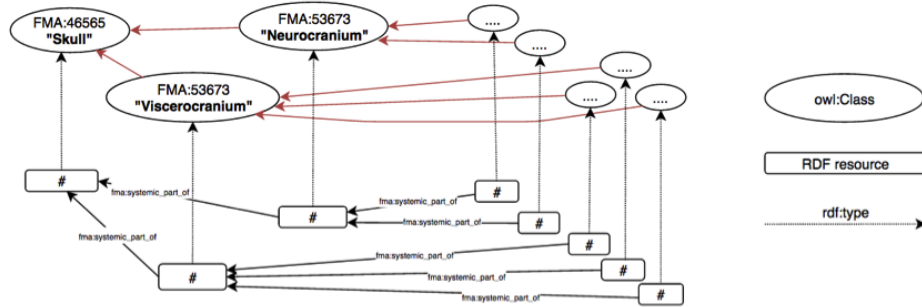


Figure 3.3: RDF Triple representation of a skull

image show the triple structure of skeletal inventories.



Figure 3.4: Skeletal Inventory Data Structure

The variable `?skeletalInventory` is the instance of the class `rdfskeleton:SkeletalInventory`, while the `?boneSegment` is from the class `rdfskeleton:SegmentOfSkeletalElement`. The core ontology of the project contains a subclass of the `rdfskeleton:SkeletalInventory`, the `rdf:PrimarySkeletalInventory`. This skeletal inventory type is for skeletal remain collections where only the whole bone organs have to be addressed. The way to define custom bone segments is always through a custom skeletal inventories, which contains restrictions on the property `obo:isAbout` and on the class of custom bone segments. Of course the custom bone segments has to be assigned to the bone organ class they belong to, via restrictions on property `obo:systemic_part_of`. The following image illustrates the extension definition.

As these extensions are expressed by OWL restriction the application can query the definitions. Consequently if the custom entry form is called from the profile of a skeletal inventory instance, then the entry form processor routine can ask, what bone segment are defined to the type of the subject variable coming as input, and can offer them on the interface.

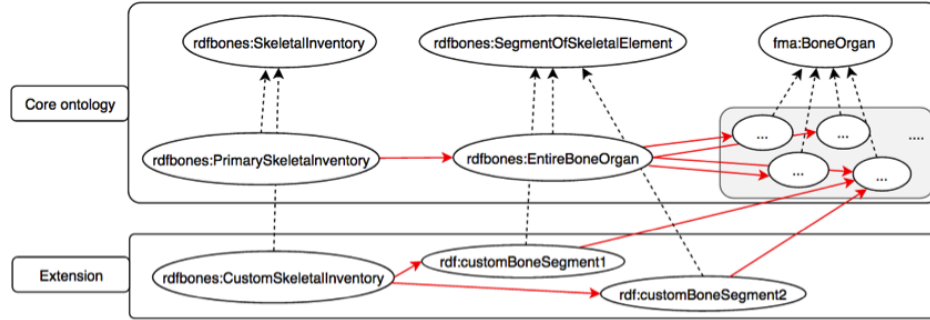


Figure 3.5: Ontology extension scheme

### 3.1.3 Study Design Execution

In most investigations the researcher take a set of bones belong to one individual and examine different tokens. Tokens refer to specific features of parts or regions of bones. These token have particular expressions.

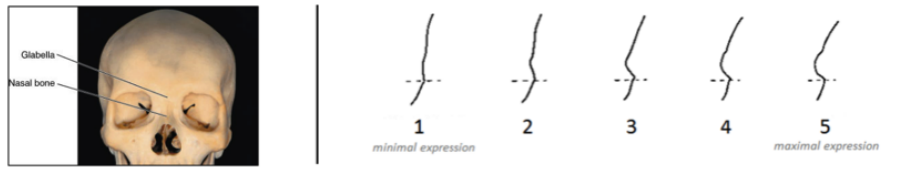


Figure 3.6: Glabella and its expressions

The previous images show the token called glabella, and its expressions. The task of the web application is let the researcher select one of the already added Nasal Bones (because on that bone is the glabella token), and set the expression of it. The following data structure models the process.

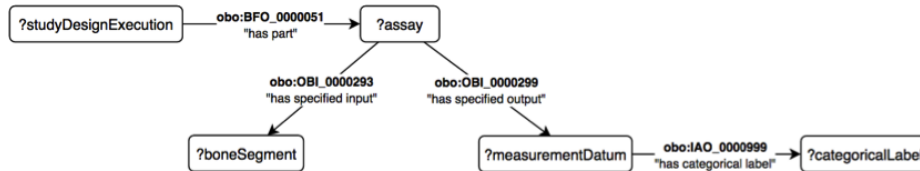


Figure 3.7: Study Design Execution Data Structure

Where the variable ?categoricalLabel represents the expression of the

token. The values this variable can take are defined in the ontology extension. The variable ?boneSegment is the bone on which the glabella can be found. This instance won't be as well newly created, but an already added bone has to be selected on the interface. The ?assay and the ?measurementDatum variables are new instances. To be able to generate an entry form for the problem, the following ontology extension has to be defined.

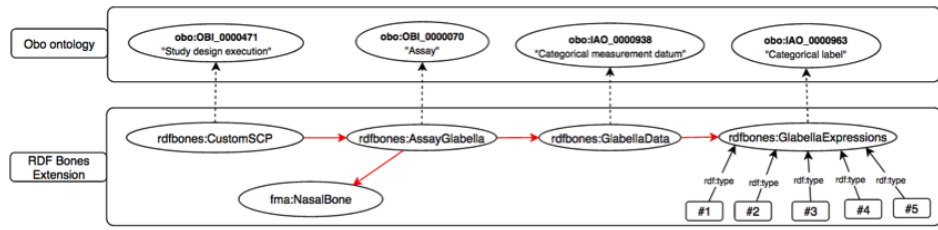


Figure 3.8: Ontology extension for Glabella

## 3.2 RDF Data input

### 3.2.1 Multi dimensional form

Imagine a problem of registering a new user to a web portal. The form offers the fields for name, e-mail address, birth, and address. The address consists of three fields, of the country city and the street-nr field. See figure 3.9 illustrates two types of user registry form. On the left (a) there is single form, which consists of the six fields. Each field holds a literal value added by the user. However on the right (b), there is a multi dimensional form, that allows the user to add multiple addresses. The implementation of such entry form requires JavaScript routines, that dynamically loads as many subform, as many times the user clicks the add button. Moreover thus the static HTML form is not sufficient for this task, JS is responsible for the reading of the data from the elements into a JSON object. On figure 3.10 the resulting data can be seen, where the blue box represents an array for the objects of the subform. Figure 3.11 illustrates the data model, by denoting with an additional box, that multiple addresses can be added, where the key of the array will be the string "addresses".

a)

b)

Figure 3.9: Single vs. multi dimensional form

a)

b)

Figure 3.10: JSON object for multi dimensional forms

### 3.2.2 Existing instances and dependencies

- It can be a reasonable step in order to improve the registration, to let the users not write as string, but select the country and the city of their addresses. Moreover the interface could refresh the list of cities if the country has been selected. To solve it in RDF based application, it is necessary to have instances of the class `Country` and `City`, and express their relationship with i.e the predicate `eq:isInCountry`.
- The task of the interface is to provide selectors, and refresh them according to user actions.
- Figure 3.12 shows the updated data model, with red triple which expresses a constraint

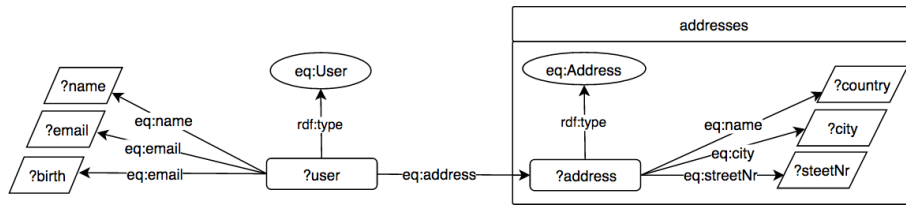


Figure 3.11: Data model

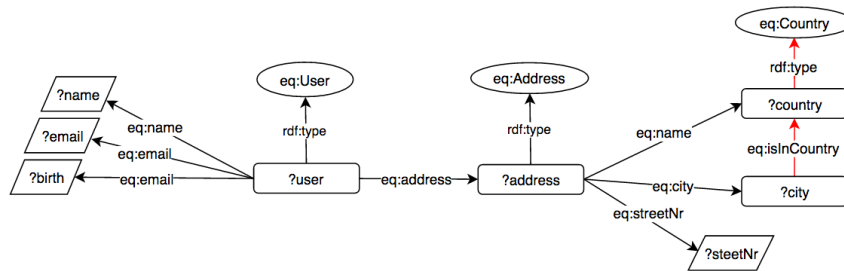


Figure 3.12: Instance selection and dependency

### 3.2.3 Instance browsing

- Figure 3.13 shows a further option for instance selection.

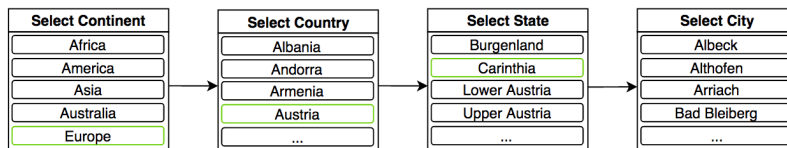


Figure 3.13: Navigator example

- The implementation requires on the server side query and the grouping of the result
- Client side - programming the navigator window

- 3.2.4 Sub form dependencies
- 3.2.5 Validation
- 3.2.6 Saving data
- 3.2.7 Editing data on the forms

# Chapter 4

## Implementation

### 4.1 Web application ontology

#### 4.1.1 Introduction

- Computer programs are able to overtake exercise from humans to accelerate work. In this case work we want speed up is the application development.

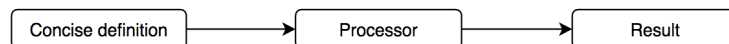


Figure 4.1: Basic workflow

- For example we have the form definition.

```
<form action="http://example.org/newUser">
  Username <input type="text" name="userName"></br>
  Password <input type="text" name="password"></br>
  <input type="submit">Submit</input>
</form>
```

- Can be modelled by a configuration dataset

```
{
  type : "form",
```



```

action : "http://example.org/newUser",
elements : [
  { type : "text", text : "Username", varName : "userName" },
  { type : "text", text : "Password", varName : "password" }
]
}

```

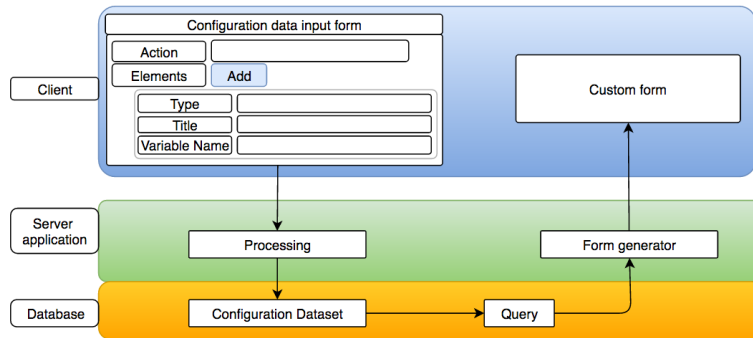


Figure 4.2: Framework functionality

- The goal is store the configuration of the web application persistently in a database. In our case the goal is to store the data in RDF data.

#### 4.1.2 Form definition

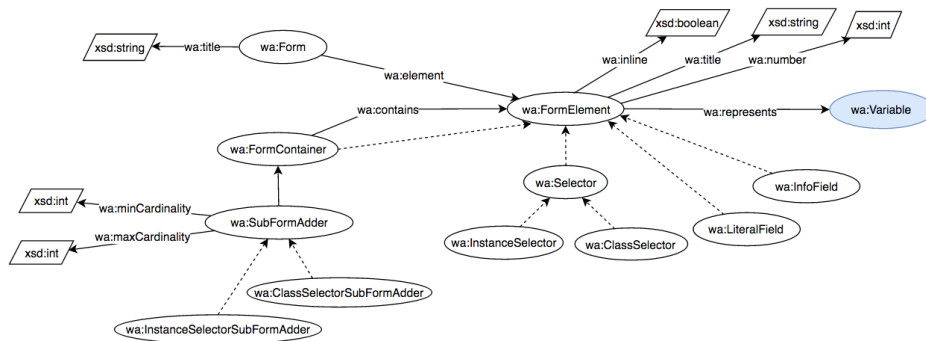


Figure 4.3: Web application ontology for the form

- Explanation of the main elements and their functionality

- Next that each form element has to represent a variable. This is denoted with blue the `wa:Variable`. And the variable name from Image X. is taken by variable instance, which is represented by the form
- Here is important to explain the difference between the `wa:SubFormAdder` and its two subclasses, `wa:ClassSelectorSubformAdder`, and `wa:InstanceSelectorSubformAdder`.

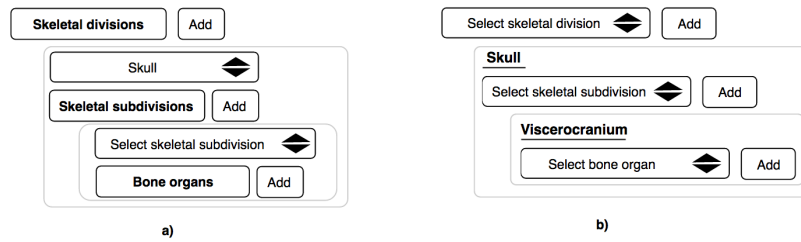


Figure 4.4: Difference between sub form adders

- min, max cardinality will be here addressed

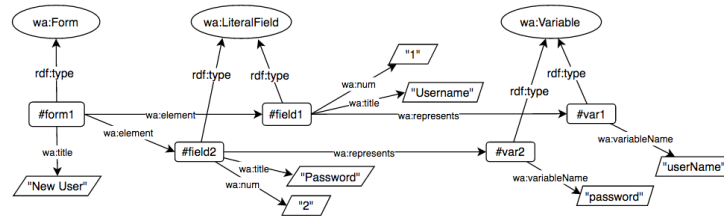


Figure 4.5: Form descriptor RDF Data

### 4.1.3 Data definition

- The goal of the data definition data scheme is to store what kind of triples has to be created
- The triples are denoted in the RDF vocabulary as statement
- There are two types of statements - the restriction and the data statement

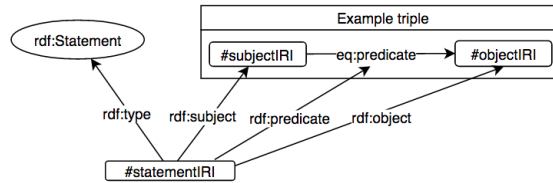


Figure 4.6: Statement in RDF Vocabulary

- And the nodes of the data graph to create is represented by the class `wa:Variable`
- These three classes represent the core of the ontology for data

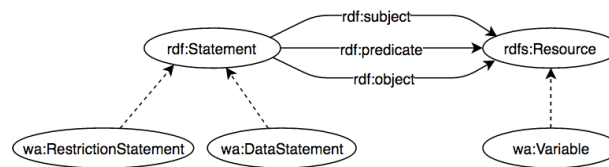


Figure 4.7: Core ontology

- The class `wa:Variable` is specified into Resource-, Class- and Literal-Variable
- And the LiteralVariable into further XML type variables
- Each variable have to have name, and they can hold a constant value or noted as main inputs if their value do not come from the form, but initially by the form loader HTTP request. In case of VIVO the main input variables are always the `subjectUri`, `predicateUri`, and `rangeUri`.
- There are two different types of statements, the class and the instance restriction statements.
- The class is used between classes, like which bone organ belongs to which skeletal division

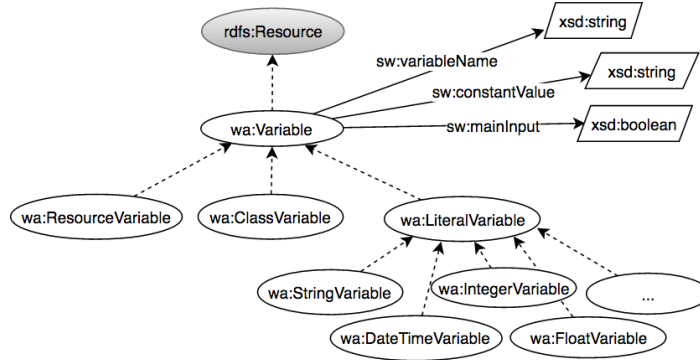


Figure 4.8: Variable types

- And the instance in turn expresses constraint between existing instances of the dataset to create, like the city-country relationship in the problem statement.
- The `wa:TypeStatement` is a statement, where the predicate is RDF type. It is interesting because it can act both as restriction and data statement as well. If the subject (so the instance) of the type statement appears on the interface, then the it is considered as restriction, but if the object (the class), then the it is data statement.
- Moreover the multi triple is really important because through them can be the multi dimensionality of the form expressed.

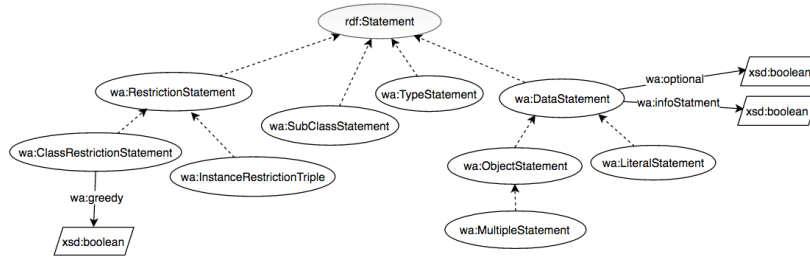


Figure 4.9: Statement types and attributes

- The following image show the configuration dataset for the problem in the previous chapter.

- In the middle there is a multi statement which expresses that more addresses of to a user can be assigned.

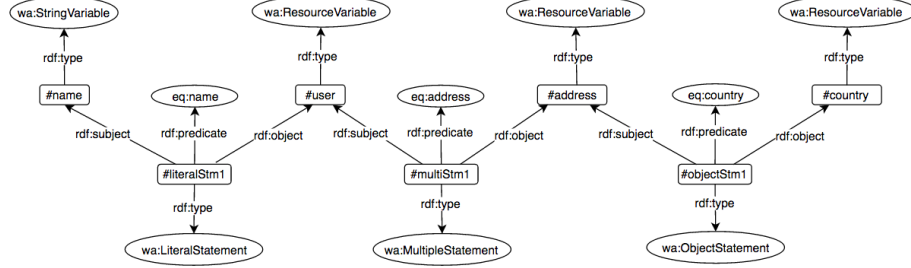


Figure 4.10: Statement configuration dataset I

- And here comes the type statement which is in this case acts as a restriction, and an instance restriction triple between city and country.

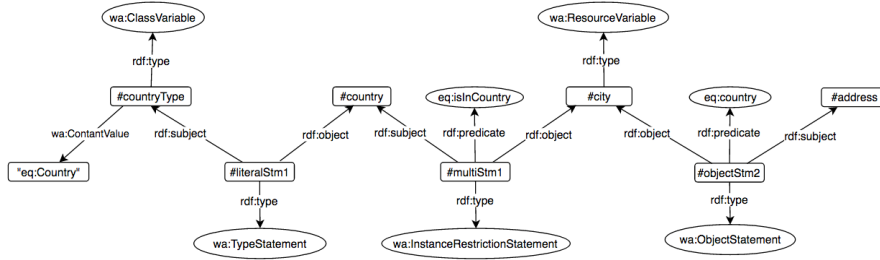


Figure 4.11: Statement configuration dataset II.

- The following image show the definition of the multi dimensional form.
- The literal field are not depicted for the sake of simplicity but they are connected to the `#form1` and `#subFormAdder` instances

#### 4.1.4 VIVO adoption

- As it was described in the previous section in VIVO each custom entry form is called by an HTTP request that contains three parameters: `subjectUri`, `predicateUri` and `objectUri`.

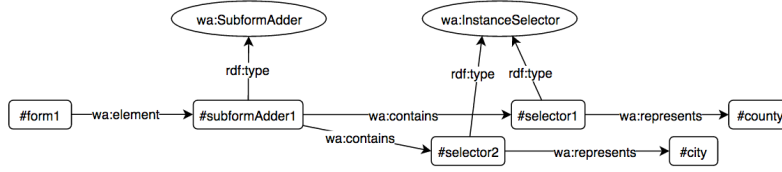


Figure 4.12: Form definition dataset

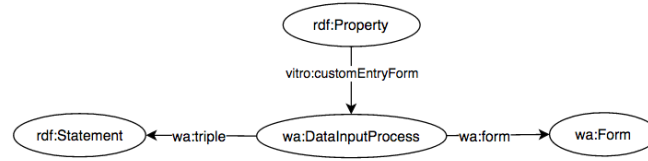


Figure 4.13: Base definition of the data input process

- So the key point is that the data input process instances can be queried by the property coming with the request.

## 4.2 Server-side implementation

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

### 4.2.1 Overview

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

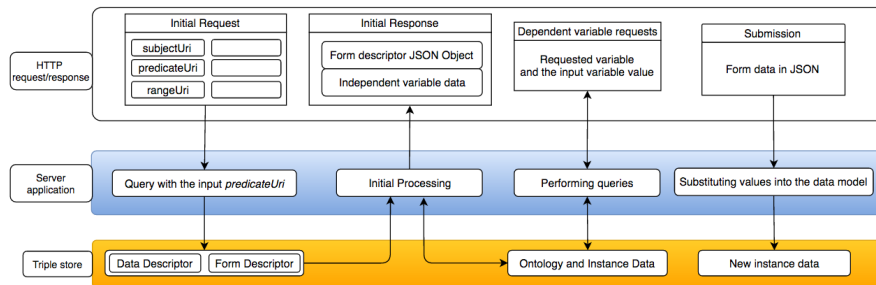


Figure 4.14: Overview of the mechanism on the server

### 4.2.2 Form representation in Java

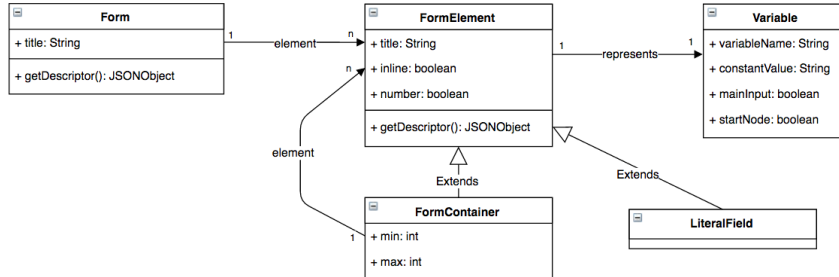


Figure 4.15: UML Diagram of the classes for the form

- 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.

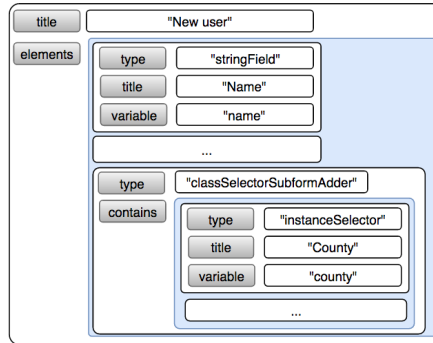


Figure 4.16: Form descriptor JSON object

### 4.2.3 Data model in Java

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

Validation:

- Data dependencies can be over graphs

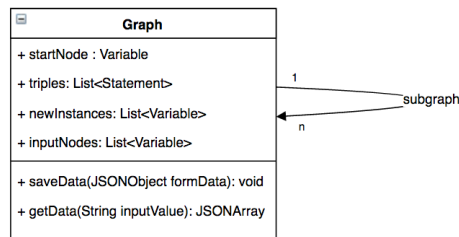


Figure 4.17: Graph UML

- 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

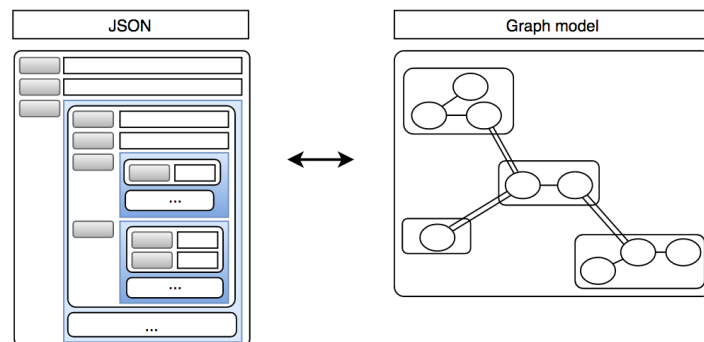


Figure 4.18: JSON vs graph model

- Data saving mechanism explanation
- Data retrieval mechanism explanation

#### 4.2.4 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....

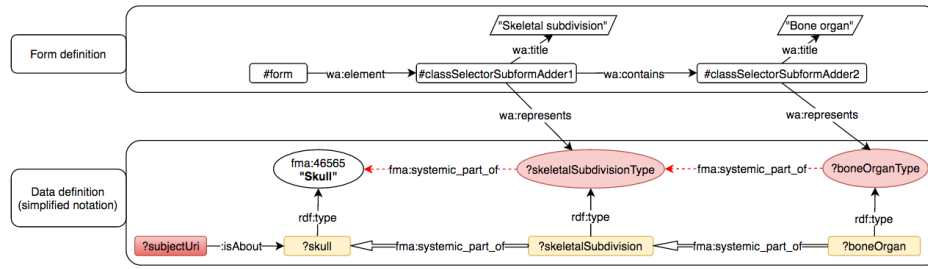


Figure 4.19: Example problem

where,

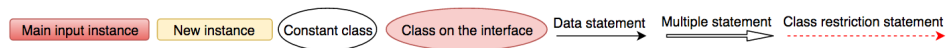


Figure 4.20: Elements of the simplified notation



Figure 4.21: Two restriction directions

- Post processing for different restriction types

#### 4.2.5 Editing and deleting data

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

#### 4.2.6 Navigator

- Goal is to display more information about instances to select in order to facilitate their finding

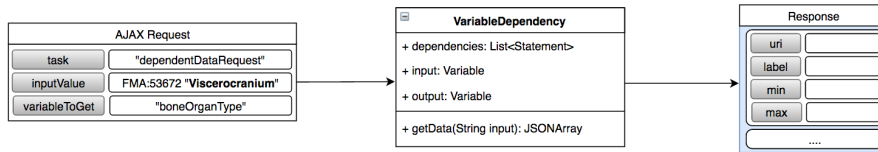


Figure 4.22: Getting dependent data

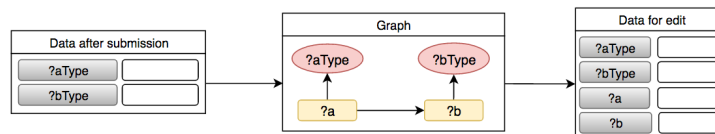


Figure 4.23: Difference between the submissions and edit data on the form

- By such advanced window it possible to offer filter options and loading the subset of the instances by introducing LIMIT on the SPARQL query. It could make the page more efficient
- But such element requires JavaScript code which could become complex
- Even if we need to offer some Navigator featured introduced in Section 3.2.3
- Example with RDF data configuration and routine for

## 4.3 Client-side implementation

- 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

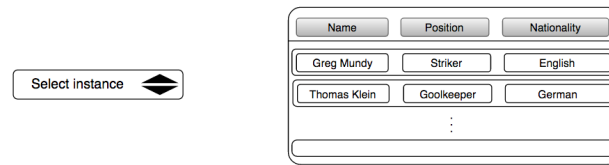


Figure 4.24: Selector VS form graph

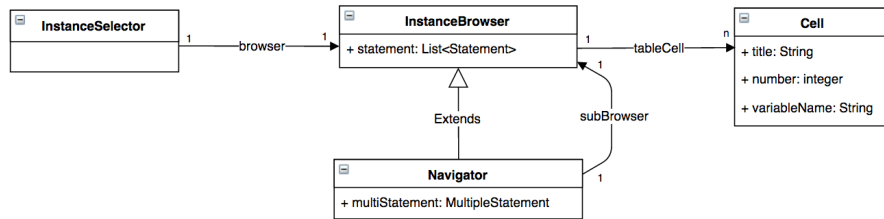


Figure 4.25: Navigator class diagram

### 4.3.1 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
- See an example for the class definition in JS

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

Listing 4.1: 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 4.2: Form generation based on configuration data

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

#### 4.3.2 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 = descriptor
        this.formData = formData
        this.inputField = $("<input/>").change(this.handler)
    }

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

Listing 4.3: 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 4.4: Data saving

### 4.3.3 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 4.5: 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

#### 4.3.4 Data dependency

#### 4.3.5 Form validation

## Evaluation

### 5.1 Problems of the RDF Bones Project

- 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.

### 5.2 Self configurable web application

- 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.

### 5.3 Limitations of the system





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