

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

Abstract	II
Kurzfassung	III
List of Tables	VIII
1 Introduction	1
1.1 Initial goal and contributions	1
1.2 Thesis outline	1
2 Preliminaries	2
2.1 Web applications	2
2.1.1 Introduction	2
2.1.2 JavaScript basics	5
2.1.3 Data in JS	6
2.1.4 AJAX	7
2.2 Semantic Web	8
2.2.1 RDF	8
2.2.2 RDF Schema	9
2.2.3 OWL	11
2.2.4 SPARQL	12
2.3 VIVO Framework	14
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	18
A Glossary	21
B Appendix	26
B.1 Something you need in the appendix	26

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	7
2.12	Main structure of the RDFS vocabulary	10
2.13	RDFS domain and range definition	10
2.14	RDFS domain and range definition	11
2.15	A subset of OWL vocabulary	12
2.16	OWL object properties	13
2.17	Properties for qualified cardinalities	13
2.18	Document class group on the admin panel	15
2.19	VIVO class group page for documents	15
2.20	RDF configuration data in VIVO	16
2.21	VIVO profile page layout	16
2.22	Triples contributing to the displayed profile page layout	17
2.23	HTTP request for data entry form	17

2.24	Object property entry form for bibo:author	17
2.25	Data property entry form	18
2.26	Data definition graphical (left) and lexical (right)	18
2.27	Definition of custom entry form configuration class	19

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.

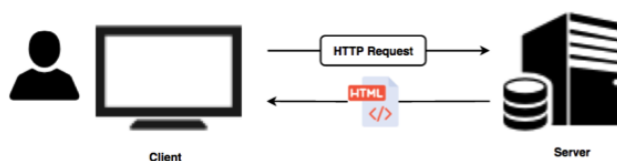


Figure 2.1: Client server communication

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.

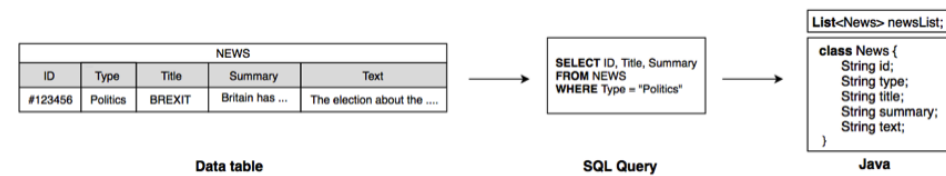


Figure 2.4: Data flow

```
<#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	<input type="text"/>
Type	<input type="text"/>
Summary	<input type="text"/>
Text	<input type="text"/>
<input type="submit" value="Submit"/>	

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

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 basics

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

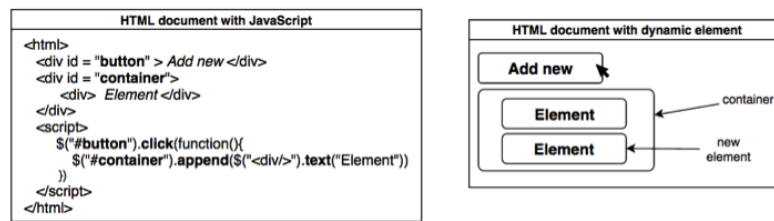


Figure 2.10: Simple JavaScript example

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 Data in JS

In JavaScript there are not typed variable, each type variable is defined by the keyword `var`. In the simplest case a JavaScript variable holds a single value of a primitive type like boolean, string or number (a). Or it be an array (b) or an object (c), which is a set of key-value pairs. The following figure shows the definition of three main JavaScript variable types.

```
var a = "someText"           //Data field
var b = ["element", 3.45, false] //Array
var c = {                    //JSON Object
    name : "Bob",
    friends : ["Jack", "John"],
    studies : {
        university : "University_of_Freiburg",
        course : "Computer_Science"
    }
}
```

Listing 2.1: JavaScript data types

Important to note the values of JavaScript object (c) can be single values, arrays or even further objects. The access of these variables are quite simple.

```
b[2]    —>    3.45
c.studies.course —> "Computer Science"
```

Listing 2.2: JavaScript data access

These variables can be set either through the template variables like any value in the HTML document. Or the event handler routines can assign values to the variables based on user actions.

2.1.4 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. This mechanism is where the JavaScript data plays a really important role. The following example shows an AJAX based solution for loading the whole text of an article.

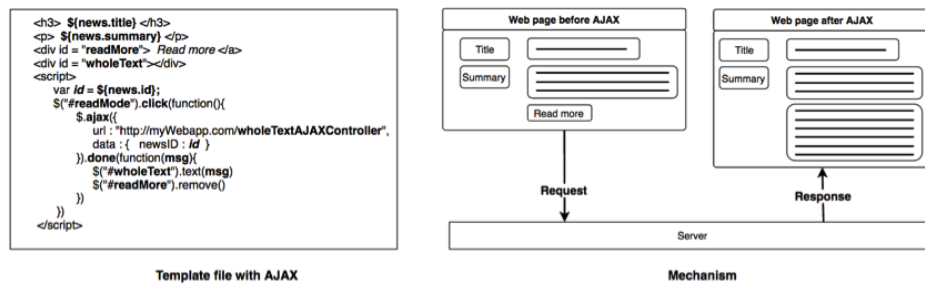


Figure 2.11: Loading new element through AJAX

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

2.2.2 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

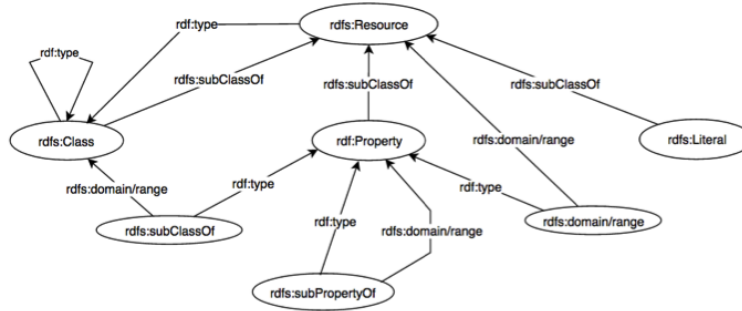


Figure 2.12: Main structure of the RDFS vocabulary

are they are related to each other. RDFS provides a mechanism to define such ontologies using RDF triples. The most important elements of the RDFS vocabulary can be seen on the following image.

The two most important classes in the RDFS vocabulary is the `rdfs:Class` and the `rdfs:Resource`. The `rdfs:Class` is class, because it is the instance of itself, and the same way the `rdfs:Resource` is a class. The `rdf:Property` 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:

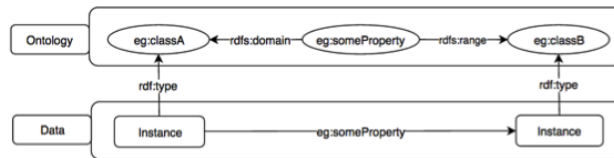


Figure 2.13: RDFS domain and range definition

Since both the `rdfs:domain` and `rdfs:range` are properties themselves, they have as well their domain and range, which is the class `rdfs:Resource`. The property `rdfs:subClassOf` expresses subclass relationship between classes. It means if a class B is a subclass of class A, and resource R is the instance

of class B, then resource R is the instance of class A as well. Since it describes the relationship between two classes its both domain and range is the class `rdfs:Class`. The property `rdf:subPropertyOf` expresses the relationship between two properties. If property P2 is sub property of P1 and two instances are related by P2 then they are related by P1 as well. Its domain and range is the class `rdf:Property`. Now everything is given to define the ontology for the example of the previous section.

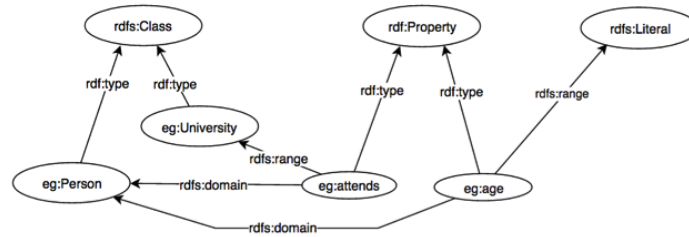


Figure 2.14: RDFS domain and range definition

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 2.3: 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 2.4: OWL restriction in N3 format

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

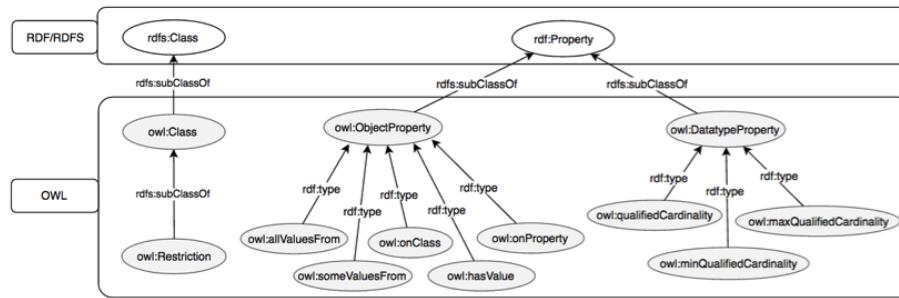


Figure 2.15: A subset of OWL vocabulary

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.

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:

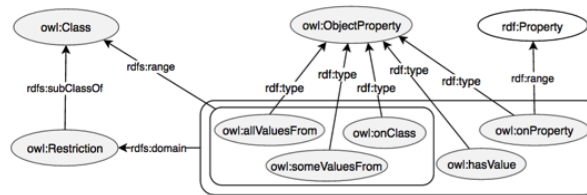


Figure 2.16: OWL object properties

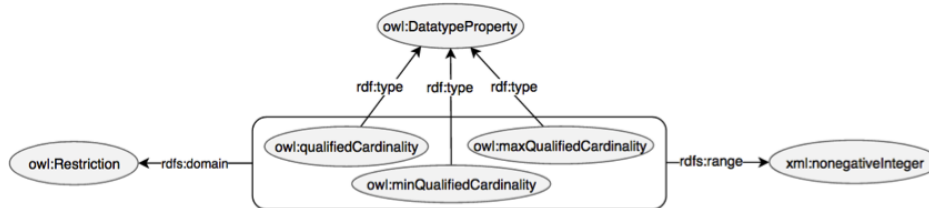


Figure 2.17: Properties for qualified cardinalities

```

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

```

Listing 2.5: 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 2.6: 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 2.7: SPARQL Query III.

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 its 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 can be 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 any 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.

Further possibility of class groups that it is possible to create links on the main menu (can be seen on the top of Figure 2.19), which redirects to a page where all the instances are listed that belong to one of the classes of

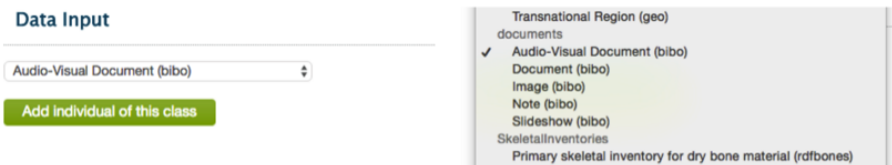


Figure 2.18: Document class group on the admin panel

the class group.

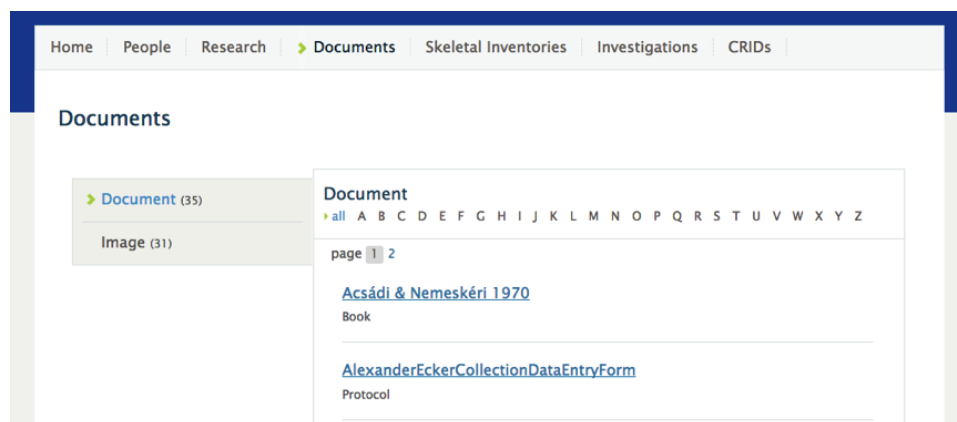


Figure 2.19: VIVO class group page for documents

The application configuration showed in the last two images is defined by the set of configuration triples on Figure 2.20.

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 by clicking one of the entries of the list on the class group pages. The profile page organizes data connected to the individual into tabs. Each tab displays the properties of a specific property group. Figure 2.21 shows a screenshot from the profile page under the Overview tab.

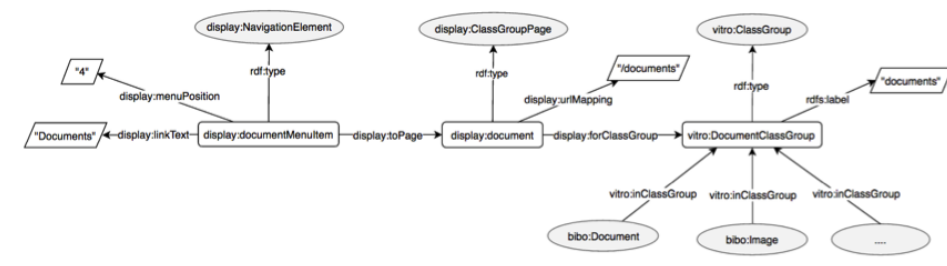


Figure 2.20: RDF configuration data in VIVO

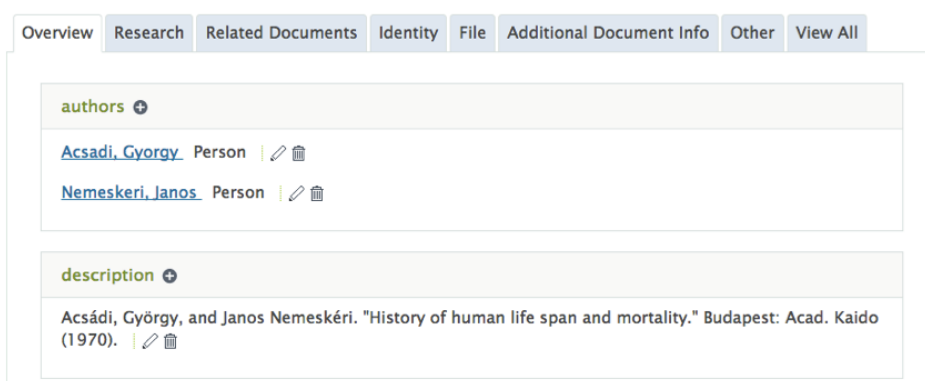


Figure 2.21: VIVO profile page layout

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.

An additional important feature of VIVO is the definition of so-called faux properties. They are really similar to the `rdfs:subProperty`, but they are considered by the profile pages.

2.3.3 Default Data Entry Forms

On Figure 2.22, 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.

They initiate the HTTP request depicted on Figure 2.23. The server gets with which subject and predicate the triple has to be created.

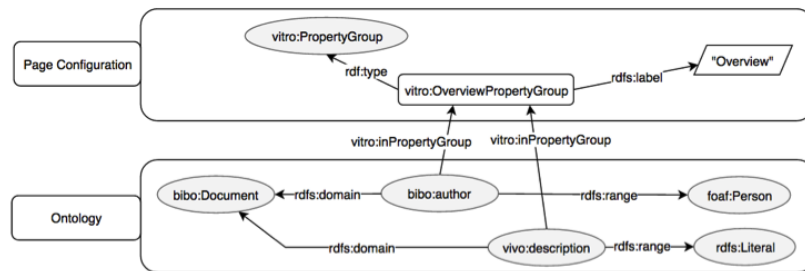


Figure 2.22: Triples contributing to the displayed profile page layout

/editRequestDispatch	
subject	http://vivo.mydomain.edu/individual/n794
predicate	http://purl.org/ontology/bibo/author

Figure 2.23: HTTP request for data entry form

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.

Add an entry of type Person for Acsádi & Nemeskéri 1970

⌵

or

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

⌵

Figure 2.24: Object property entry form for `bibo:author`

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

Change text for: description

An account of the resource.

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.

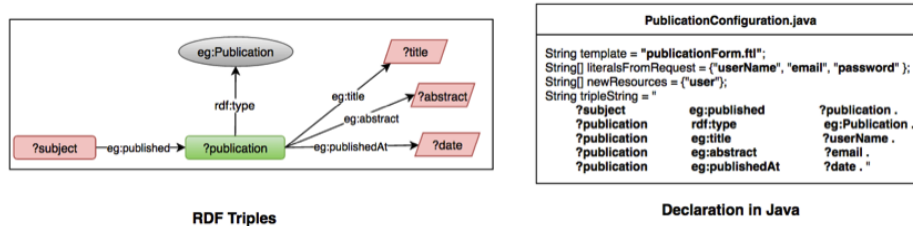


Figure 2.26: Data definition graphical (left) and lexical (right)

dataDef

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 ??, 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 configuration 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

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.