STI – SEMANTIC TECHNOLOGY INSTITUTE

University of Innsbruck
Semantic Technology Institute

OMS: Ontology Mapping Store

Bachelor Thesis

Adrian Marte, Christoph Fuchs

Supervised by Dr. Elena Simperl
and co-supervised by Francois Scharffe

Innsbruck, January 18, 2008
Abstract

Nowadays a lot of ontologies and schemes are available on the Web, which express the same domain (semantic) but in different forms (syntax). Francois Scharffe, Antoine Zimmermann and Jerome Euzenat introduced a language-independent expressive mapping language [6] to link entities and relate equipollent ontologies. In our thesis we want to develop a Web service, which represents a storage to efficiently store and query for such ontology mapping documents available in different formats (XML, human readable format, ...) and a Web application providing access to the Web service.
Contents

1 Introduction 1
  1.1 Introduction ............................................. 1
     1.1.1 Semantic Web ....................................... 1
     1.1.2 Ontology ............................................. 2
     1.1.3 Ontology Mapping .................................... 3
  1.2 Global Architecture ................................. 6
     1.2.1 Key Features ......................................... 6
     1.2.2 Web Service .......................................... 8
     1.2.3 Web Application ..................................... 8
  1.3 REST Interface ........................................... 9
     1.3.1 How REST Works .................................... 9
     1.3.2 Specification ......................................... 9

2 Implementation 15
  2.1 Web Service ............................................. 15
     2.1.1 Layered Architecture ............................... 15
     2.1.2 The Connection Layer ............................... 17
     2.1.3 The Repository Layer ............................... 23
  2.2 Web Application ......................................... 27
     2.2.1 Ruby on Rails ....................................... 28
     2.2.2 User Interface ..................................... 32
     2.2.3 Basic Functionality ............................... 33
     2.2.4 Advanced Functionality ............................ 35
     2.2.5 Query Language .................................... 36
     2.2.6 Connection to Web Service ....................... 38

3 Conclusion 43
  3.1 Problems ............................................... 43
  3.2 Possible Improvements ............................... 45
Chapter 1

Introduction

1.1 Introduction

In the first chapter we will have a look at some basic terms and technologies that are essential for the Semantic Web and therefore for our thesis. In section one we will give a short introduction to the matter and have a look at the history of the Semantic Web. In the second and third section we will discuss the global architecture of the Ontology Mapping Store, i.e. the Web service and the Web application, and the interface through which it is accessed. Later we will describe both the Web service and the Web application in more detail, followed by a chapter mentioning some improvements that could be applied.

Throughout this paper we will illustrate the functionality of the Ontology Mapping Store by using an Ontology Mapping between two simple ontologies describing the domain “Computer Games”.

We assume that the reader is familiar with Internet technologies, so non-semantic technologies won’t be explained in detail.

1.1.1 Semantic Web

It all began back in 1980 when Sir Tim Berners-Lee started a project named ENQUIRE while working in the CERN\(^1\) laboratories. The purpose of ENQUIRE was to provide an environment that let him keep “track of all the random associations one comes across in real life and brains are supposed to be so good at remembering but sometimes mine wouldn’t.” [13] One of the biggest disadvantages of this program was the restriction to a single computer. Later on

\(^1\)Conseil Europeen pour la Recherche Nucleaire (European Council for Nuclear Research)
in 1989 Berners-Lee created a technique that allowed not only himself but also his colleagues to share recent results and thoughts via multiple computers. His invention did not stay in CERN - soon scientists all over the world were able to cooperate and communicate via a system of interlinked, hypertext documents which we know now as the World Wide Web. Since then the number of users grew steadily, reaching one billion users in 2005.

Due to it’s massive, sometimes exponential growth, the WWW contains an immense amount of information. According to a study by Maurice de Kunder at the Tilburg University [3] the indexed web contains at least 22.6 billion pages (as of September 18, 2007). The problem with this information load is that machines do not understand it’s content. The Web as we know it is made by human beings for human beings, who are able to book a trip from Innsbruck to London, find a cheap book and buy it, or filter out relevant information on a webpage. Today’s computers are not able to perform these kind of tasks, because they are unable to understand natural languages.

However, the vision of Sir Tim Berners-Lee is a World Wide Web which is also readable by computers - the Semantic Web. The following quote is from his book Weaving the Web [14] which was published in 1999.

“I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web - the content, links, and transactions between people and computers. A 'Semantic Web', which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The 'intelligent agents' people have touted for ages will finally materialize.”

– Sir Tim Berners-Lee, 1999

As of today, the Semantic Web is still a vision. Most of the data on the Web is still reserved for humans, thus only available in human-readable formats.

One approach of realizing this vision are ontologies, which treat data in a way that computers are able to understand.

### 1.1.2 Ontology

So what is an ontology? An ontology (in the context of the Semantic Web) is a data model representing a domain. This domain contains different objects that can be classified and put in relation with each other. Furthermore, ontologies are used to express these objects, classes and relations.

To describe ontologies many different languages have been introduced. Beside human-readable formats, there are several machine-readable languages (e.g.
1.1. INTRODUCTION

RDFS [19], OWL [17]) which are of major importance for the Semantic Web [16].

Two example ontologies describing the domain “Computer Games” are represented in figure 1.1. In the first ontology there are two Genres Strategy and First Person Shooter. Round Based Strategy and Real Time Strategy are two sub-genres of Strategy. The second ontology consists of the genres Real Time Strategy, FPS and Simulation and its sub-genres Racing and Flight Simulation. As you can see, these two ontologies describe the same domain, but the contained objects are different.

1.1.3 Ontology Mapping

The idea of Ontology Mapping is to develop independent languages and formats to represent the relations between two ontologies of the same or of overlapping domains. Ontology Mapping is the process of transforming the source ontology into the target ontology (and vice versa) according to those relations. By using ontology mappings machines are able to gain a wider range of information about a specific entity.

Figure 1.2 shows a mapping between the two ontologies introduced in section 1.1.2. First, there is a mapping between the two classes First Person Shooter and FPS. These two classes represent the same thing, but in a different way. The relation between the two classes Real Time Strategy and RTS is similar.

Ontology Mapping Language

When it comes down to describing things, some kind of a language is required. In computer science, a well-defined programming language is needed to describe a program or an algorithm. In the Semantic Web we need a mapping language to realize Ontology Mapping. For this a language-independent expressive mapping
language has been developed by Francois Scharffe, Antoine Zimmermann and Jerome Euzenat. With creating this mapping language the following aspects have been covered:

**Web compatibility** Based on the web standards URI [20], XML [22] and RDF [21] the RDF/XML format provides a machine-readable language which is extensible and capable of referencing particular correspondences.

**Loose coupling with ontologies** Some ontology languages provide simple constructs to map different ontologies, e.g. `owl:imports` in the Web Ontology Language OWL. But since these mappings are embedded in the ontology definition itself, this method of mapping results in a tight coupling of the two ontologies. The mapping language however tries to keep interoperability as flexible as possible by sourcing out the mapping into an independent system.

**Expressiveness** Sometimes two ontologies are not exactly equal, therefore applying equivalence is simply not enough. To solve this problem expressive rule languages like SWRL [18] or WSML [4] are needed to create more complex relations.

**Heterogeneous schema languages** The problem in linking two or more ontologies together are their languages. Since there exist different languages (SQL, XML) and formats (RDF, OWL) for describing ontologies, the mapping has to be language-independent and capable of expressing the complexity within the ontologies. For this the Ontology Alignment Format and the Ontology Mapping Language were introduced.

**Simplicity** Even if more complex mappings have to be created, simpler relations should not suffer. Using a well structured format is essential to avoid unnecessary overhead.

**Extensibility** The problem that comes with extending any format is to keep reverse compatibility, i.e. tools which use this format must not be dis-
1.1. INTRODUCTION

turbed by the extensions. Languages like RDF provide structures which fulfill these requirements.

Purpose independence Different purposes require different formats. For this, format conversion is often necessary.

Executability As mentioned before, there are different languages for describing mappings. Each of them has to be machine-readable and -executable, i.e., there should exist tools for interpreting and parsing such formats.

The ontology mapping language allows to specify correspondences between two ontologies. What makes it specific is the possibility to represent complex correspondences independently from the language the ontologies are modeled in. In other words the mapping language gives a way to represent all kinds of schema mappings.

Mapping API

To manipulate the mapping language and to build applications upon it, a mapping API has been developed by François Scharffe, Antoine Zimmermann and Jerome Euzenat. The Mapping API has been implemented in Java and therefore can be used platform-independently. The API offers the following functionality:

- Parsing mapping documents. The parser populates an object model representing the different expressions of the mapping language. It supports all syntaxes of the mapping language available at the moment. The object model can be easily accessed using a set of classical methods.

- Serializing mapping documents. The mapping documents (represented by the object model) can currently be serialized in the mapping language abstract/human-readable syntax, and in the Web Services Modeling Language abstract/human-readable syntax. In the future there will also be support for a RDF/XML syntax, as well as export methods to other ontology representation such as OWL and Flora.

The API is freely available under the license of the Distributed Ontology Management Environment and can be downloaded at the tools section of the Ontology Management Working Group\(^2\). In the approach presented in this paper the version 0.1.1 of the mapping API was used.

However, we had to edit some piece of code in the parser class, the alignment.XpathParser in the org.omwg.mediation.parser packet to be more precise, to enable parsing from an input stream. We had to do this because we have to parse documents referenced by an URI object. Listing 1.1 shows the additional method that has been added to the parser.

1.2 Global Architecture

In terms of the Semantic Web we provide a Web service representing a repository to store and query mapping documents. Based on this Web service we develop a Web application, which enables the user to make use of the Web service on a well structured and easy-to-use interface.

We developed a global architecture of our system using current technologies and web standards, as shown in figure 1.3.

The Web service is written in Java and the application is implemented in Ruby on Rails. Both use a RESTful interface (described in section 1.3), whereas the Connection interface of the Web service can easily be exchanged with another interface, e.g. SOAP\(^3\), see section 2.1. With this architecture it was possible to first define the interfaces on how the Web service works and how it provides it’s functionality, and then start implementing both the Web service and the Web application independently and synchronously.

1.2.1 Key Features

As mentioned before, all the features of the Ontology Mapping Store have to be offered by the Web service and have to be accessible via the Web application. Thus, there will be no difference between the basic features that are offered by the service and those that are presented in the application.

\(^3\)Simple Object Access Protocol
The main purpose of the *Ontology Mapping Store* is to manage ontology mapping documents in different formats (Human Readable Format, XML, RDF, ...). At the beginning of this project only the basic CRUD [25] operations have been taken into account, which are the following:

- Create (store) ontology mapping documents in different formats
- Read (retrieve) ontology mapping documents in different formats
- Update (edit) ontology mapping documents in different formats
- Delete (remove) ontology mapping documents in different formats

Advanced features like the management of users, search and sorting the results were not relevant at first. During development, we thought that it would be nice to have at least some kind of user management, so that not everyone can edit and delete any mapping he or she wishes to. This task has to be achieved through the Web service, because we need a centralized user database. If the application itself would handle the user management, there would be no possibility to separate this application from the Web service or to write other applications for it. For this the following features have been introduced:
Unfortunately, this approach led to some problems. Due to the fact that the Web service stores the user data, every time a method of the Web service is called where user authentication is required, the user data, i.e. user name and password, have to be passed to check for validity. In our implementation this data is sent in plain text. However, when authenticating through the Web application the password is encrypted using the $SHA1$ encryption algorithm before it is sent. This increases the security a little bit, because an intruder is unable to sniff the password as is.

Apparently this is still insecure, since one could sniff the packets sent to the Web service and would be able to see the authentication data. But we thought this problem is negligible since security is not a big issue in this kind of project. To solve this problem one could use a secure or encrypted connection like $HTTPS$. For more details see section 3.2.

### 1.2.2 Web Service

The Web service provides methods to manage both mapping documents and users associated with the mapping documents. For both we have implemented the operations described in section 1.2.1. A layered architecture allows to keep the Web service as flexible as possible, e.g. one can add or exchange connection interfaces to provide alternatives to access the Web service, apart from our RESTful interface.

A main feature of the Web service is that it can handle mapping documents in different formats (unfortunately, only $OMWG$ is supported at the moment), i.e. the web service offers the possibility of storing, respectively querying, mapping documents and selecting different output and input formats. For this the Web service uses, amongst others, the $Mapping API$, a Java framework for processing mapping documents [6]. Additionally we integrated a superuser, which has the abilities to manage all users and mapping documents regardless of their owner.

### 1.2.3 Web Application

The Web application should provide an easy-to-use front-end for the Web service. The core functionality, given by the Web service itself, has to be covered.
However, additional functionality (like sorting, filtering, etc.) can be added by the application. As far as the Web application is concerned we have decided to write it in Ruby on Rails (see section 2.2.1). For further details on the Web application, see chapter 2.2.

1.3 REST Interface

REST, which is short for **Representational State Transfer**, is an architectural style for distributed hypermedia systems like the WWW. REST is a hybrid style deduced from several network-based architectural styles, e.g. the client-server architecture. The term REST was first introduced by Roy Thomas Fielding in his dissertation *Architectural Styles and the Design of Network-based Software Architectures* [5] in the year 2000. Today, the Representational State Transfer style is well established and is widely used throughout different types of networks, mainly the WWW.

1.3.1 How REST Works

The basic principles of REST are identifiers which represent resources on the server side, and states that are transferred on the client side. When a client in the network sends a request to a server, only the identifier and a basic operation (mostly CRUD operations) is passed on. The server responds with the resource itself and the state of the client changes. The most common example of a RESTful architecture is the Web, since much of it conforms to the REST principle.

**Example: Browsing the Web**

A Wikipedia article about Roy Fielding (the resource) is represented by the following URI (identifier): http://en.wikipedia.org/wiki/Roy_Fielding. The user (client) is on a Homepage that has a link to this article (the user is in some state). Once the user clicks on this link, the browser sends a HTTP request to the server, like GET http://en.wikipedia.org/wiki/Roy_Fielding. The server responds with the HTML code of the article (resource) and the browser (client) changes its state.

1.3.2 Specification

The Web service can be accessed in a RESTful way, that means the functionality is provided through URIs. Basically, the REST architecture suggests that one
should use the given HTTP methods to perform the appropriate operation, as shown in Table 1.1. Since PUT and DELETE are hardly implemented we additionally provide an interface, which can be used with only the GET method, which every HTTP server is supposed to have implemented.

The main objects that are manipulated by the Web service are users and mappings. The Web service provides at least two URIs for every operation that can be performed. Additionally to the basic CRUD operations, another method which retrieves mapping metadata has been added. This allows a faster retrieval of basic mapping information, resulting in a higher performance. This circumstance leads to 18 different operations, split into ten mapping and eight user operations.

### Handling of Mappings

The basic mapping operations are

- Add Mapping Document (listing 1.2)
- Retrieve one or more Mapping Documents (listing 1.3)
- Retrieve Mapping Metadata (listing 1.4)
- Edit Mapping Document (listing 1.7)
- Delete Mapping Document (listing 1.8)

The following listings contain the specification of each operation. Parameters \((\text{param } 1, \text{param } 2, \ldots)\) may be added if necessary. For a detailed specification of all allowed parameters see table 1.2. For every operation there is an alternative operation, using only HTTP GET. Those alternatives may be used if the HTTP functions DELETE or PUT are not implemented properly. Parts that are surrounded by square brackets are optional.

<table>
<thead>
<tr>
<th>Action</th>
<th>HTTP</th>
<th>CRUD</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Mapping or User</td>
<td>POST</td>
<td>CREATE</td>
<td>INSERT</td>
</tr>
<tr>
<td>Retrieve Mapping, Mapping Metadata or User Information</td>
<td>GET</td>
<td>READ</td>
<td>SELECT</td>
</tr>
<tr>
<td>Edit Mapping or User Information</td>
<td>PUT</td>
<td>UPDATE</td>
<td>UPDATE</td>
</tr>
<tr>
<td>Delete Mapping or User</td>
<td>DELETE</td>
<td>DELETE</td>
<td>DELETE</td>
</tr>
</tbody>
</table>

Table 1.1: Analogies
1.3. REST INTERFACE

Adding a mapping implies three mandatory parameters. The parameters userName and userPassword must be set, as well as mapping or URL. The format parameter is optional and will automatically be set to the default format (OMWG) if not set. If the mapping was added successfully, the mapping itself is returned. If not, an empty document will be returned.

By retrieving one or multiple mappings or mapping metadata, all applicable parameters become optional. When there are no parameters given, an XML document containing all mappings in the database will be returned. The basic structure of the document is shown in listing 1.5. When more than one mappings found to match the criteria given by various parameters, a similar XML document containing those mappings is returned. If there is exactly one mapping matching the criteria, this mapping is returned. If there is no mapping found, an empty set of mappings will be returned (see listing 1.6).
The update and delete methods are very straightforward. Update and delete require user authentication, thus userName and userPassword are mandatory parameters. Because updating an existing mapping is similar to adding a new mapping, there are the same four parameters needed as in the Add Mapping Document operation (listing 1.2). Deleting a mapping, in addition to user authentication, requires a clear identification of the mapping, given by either the ID or mappingID parameter.

Listing 1.7: Update Mapping Document

```
PUT /mapping/?param1=...&param2=...&...

# Alternative
GET /mapping/update?param1=...&param2=...&...
```

Listing 1.8: Delete Mapping Document

```
DELETE /mapping/?param1=...&param2=...&...

# Alternative
GET /mapping/delete?param1=...&param2=...&...
```

Actions that require user authentication may return an error message when the user authentication failed, i.e. the user data is empty or invalid. The Web service returns an XML document containing a descriptive error message (listing 1.9). This kind of error is also returned when another user tries to add a mapping that has the same mappingID than an existing mapping, because the system assumes that the user wants to update the existing mapping.

Listing 1.9: XML document returned if validation fails

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<error>
  <message>Invalid user data</message>
</error>
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Applicable for</th>
</tr>
</thead>
<tbody>
<tr>
<td>userName</td>
<td>Name of the User</td>
<td>Add, Edit, Delete</td>
</tr>
<tr>
<td>userPassword</td>
<td>Password of the User</td>
<td>Add, Edit, Delete</td>
</tr>
<tr>
<td>mapping</td>
<td>The Mapping Document</td>
<td>Add, Edit</td>
</tr>
<tr>
<td>URL</td>
<td>URL to the Mapping Document</td>
<td>Add, Edit, Retrieve</td>
</tr>
<tr>
<td>format</td>
<td>Format of the Mapping Document</td>
<td>Add, Edit, Retrieve</td>
</tr>
<tr>
<td>source</td>
<td>Source of the Mapping Document</td>
<td>Retrieve</td>
</tr>
<tr>
<td>target</td>
<td>Target of the Mapping Document</td>
<td>Retrieve</td>
</tr>
<tr>
<td>ID</td>
<td>Internal ID of the Mapping</td>
<td>Retrieve, Delete</td>
</tr>
<tr>
<td>mappingID</td>
<td>Mapping ID of the Mapping Document</td>
<td>Retrieve, Delete</td>
</tr>
<tr>
<td>userID</td>
<td>ID of the User owning the Mapping</td>
<td>Retrieve</td>
</tr>
</tbody>
</table>

Table 1.2: Parameters used by mapping operations
1.3. REST INTERFACE

Handling of Users

The basic user operations are

- Create User Account (listing 1.10)
- Retrieve User Information (listing 1.12)
- Edit User Information (listing 1.14)
- Delete User Account (listing 1.15)

Creating a user account requires three mandatory parameters. The Web service expects `userName`, `userEmail` and `userPassword`. For a complete list of parameters see table 1.3.

Listing 1.10: Create User Account

```plaintext
POST /user?param1=...&param2=...

# Alternative
GET /user/add?param1=...&param2=...
```

Before a user is added to the database various checks are performed. At first username and e-mail are checked on availability. If either the desired username, or the e-mail provided is already in use, the response of the Web service will be an XML document containing an error (listing 1.11).

Listing 1.11: Erroneous response while creating a user account

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<error>
  <message>User already exists</message>
</error>
```

Checking for validity of e-mail and username is intended to be done by the Web application. The Web service itself accepts any kind of string as username or e-mail address.

Information about a specific user account is retrieved as in listing 1.12.

Listing 1.12: Retrieve User Information

```plaintext
GET /user?param1=...&param2=...

# Alternative
GET /user/get?param1=...&param2=...
```

For privacy reasons there is no possibility to list multiple users. However, if the username is correct, an XML document containing the user information is returned. An example of such a document is shown in listing 1.13. By returning
the e-mail address it is possible that someone acquires private information (the e-mail address of a registered user) by guessing a correct username. This is a known security issue and can be solved by adding an option if the user wants his e-mail address displayed or not. For more details see section 3.2.

Listing 1.13: User information

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<user>
  <id>2</id>
  <name>Bob</name>
  <email>bob@example.com</email>
</user>
```

A user that is logged in should be able to edit his account information. To achieve that, the operation shown in listing 1.14 has been implemented. The user is able to change his e-mail address and his password. The username can not be changed.

Listing 1.14: Edit User Information

```plaintext
PUT /user?param1=...&param2=...&...

# Alternative
GET /user/update?param1=...&param2=...&...
```

Deleting an account can only be done by a superuser. This may be necessary if a user is flooding the database with senseless mappings.

Listing 1.15: Delete User Account

```plaintext
DELETE /user?param1=...&param2=...&...

# Alternative
GET /user/delete?param1=...&param2=...&...
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Applicable for</th>
</tr>
</thead>
<tbody>
<tr>
<td>userID</td>
<td>Internal ID of the Mapping</td>
<td>Retrieve, Delete</td>
</tr>
<tr>
<td>userName</td>
<td>Name of the User</td>
<td>Create, Retrieve, Edit, Delete</td>
</tr>
<tr>
<td>userEmail</td>
<td>E-mail of the User</td>
<td>Create, Retrieve, Edit, Delete</td>
</tr>
<tr>
<td>userPassword</td>
<td>Password of the User</td>
<td>Create, Edit, Delete</td>
</tr>
<tr>
<td>userNewPassword</td>
<td>New password of the User</td>
<td>Edit</td>
</tr>
<tr>
<td>userNewEmail</td>
<td>New e-mail of the User</td>
<td>Edit</td>
</tr>
<tr>
<td>adminName</td>
<td>Name of the superuser</td>
<td>Delete</td>
</tr>
<tr>
<td>adminPassword</td>
<td>Password of the superuser</td>
<td>Delete</td>
</tr>
</tbody>
</table>

Table 1.3: Parameters used by user operations
Chapter 2

Implementation

2.1 Web Service

This section describes how the Web service is implemented. First a short overview of how the layered architecture looks like is given. Then each layer is described in more detail. The ‘specification’ of the Connection layer - the REST interface to be more precise - is the most significant one, since it describes how the Web service can be accessed.

2.1.1 Layered Architecture

For reusability and interexchangeability we developed a two-layered architecture for the Web service consisting of the Connection and Repository layer (see figure 2.1).

**Connection** Receives, decodes and forwards the incoming requests to the Repository layer and returns the processed data retrieved from the Repository layer. In other words, this layer gives an interface to access the functionality of the Web service from the “outside”.

**Repository** Provides the repository functionality, i.e. manages the mapping documents and users by providing both a mapping store and a user store. In other words, this layer implements the functionality or business logic of the Web service.

In our solution the Connection layer is implemented using the free Java Framework Restlet, which provides a simple and structured way to implement RESTful
Web services (and Web applications). The Repository layer is realized using Hibernate, a powerful, high performance object/relational persistence and query service, on top of Apache Derby, an open source relational database implemented entirely in Java.

The process of adding a mapping document to the Ontology Mapping Store looks like this: The Connection layer receives a request to store a mapping document in the repository. It first extracts and decodes the received data and then forwards this request to the repository. The repository then checks if all required data provided is valid and the given user has the required access rights. If everything is fine, the mapping document is parsed using the mapping API and then stored in the data structure. The repository sends a response to the Connection layer that the mapping document has been stored, which then gets forwarded to the request source.

After some experiments it turned out that a lot of validation checks have to be performed before any request is executed. Since we wanted the repository to assume that incoming requests are valid and that every required data is given, we introduced another layer within the Repository layer, the Facade layer. Instead of sending requests directly to the repository, the Connection layer sends them to the Facade layer. The Facade then checks if everything is fine, parses the mapping document and forwards the requests to the repository and so on. Figure 2.2 shows the final architecture of the Web service.
For the Facade layer and the Repository layer a Java interface has been developed, which has to be implemented when exchanging these layers. There exist two interfaces for both the Mapping Store - the repository for mapping documents - and the User Store - the repository for users. These interfaces define the methods mentioned in section 1.2.1. See section 2.1.3 for more information.

As far as the Connection layer is concerned, it is very difficult to define interfaces, since the various Web interfaces such as SOAP or RPC have hardly any similarities.

2.1.2 The Connection Layer

The Connection layer represents the connection point to the outside, i.e. it gives access to the functionality of the Web service. Introducing this layer gives the possibility to offer various interfaces to access the Web service at the same time and to easily exchange them without effecting the functionality of the Web service.

The Connection layer receives, decodes and forwards incoming requests to the Repository layer, the Facade layer to be more precise (see section 2.1.1), and returns the processed data, which was retrieved, to the request source.

The Restlet Framework

In our solution the Connection layer has been realized using the Restlet framework, a free Java framework, which provides a simple and structured way to implement both RESTful Web services and Web clients. It is therefore a client and a server framework.

The Restlet framework tries to comply with the REST architectural style as much as possible. It provides classes and interfaces that represent the REST concepts such as resource, representation, component, etc. This makes it easy to think RESTfully and to translate a solution into code more naturally.

The Restlet framework is composed of two main parts. First, there is the “Restlet API”, a neutral API supporting the concepts of REST and facilitating the handling of calls for both client-side and server-side applications. This API must be supported by a Restlet implementation before it can effectively be used. Currently, the “Noelios Restlet Engine” (NRE) is available and acts as the reference implementation. From now on we will refer to this combination simply as the Restlet API.

Important components and therefore classes of the Restlet framework are:
CHAPTER 2. IMPLEMENTATION

Restlet The Restlet class is a base component of the Restlet API, it offers methods to handle and process incoming requests. All classes listed here are subclasses of the Restlet class.

Application The Application class is the component which manages a set of related Restlets. It represents, as its names suggest, an application providing some functionality.

Router Routers are specialized Restlets that can have other Restlets attached to them. They can automatically delegate calls to the attached Restlets based on a URI template.

Connector A Connector in REST is a software element, that enables the communication between components, typically by implementing one side of a network protocol. There are various connectors for both the client and the server side. For example, one can easily setup a server, which provides various resources, using the HTTP server connector. The Restlet framework provides several implementations of client and server connectors.

Client A Client can work with remote resources using various connectors.

We started using the Restlet framework when it was still in beta stage. Unfortunately, the API experienced major changes when changing from beta to stable. Since we wanted the Web service to be considered as stable, we changed to the stable release of the Restlet API, although we had to make some major changes to the existing source code (see section 2.1.2).

The Restlet framework can be freely downloaded under either the CDDL license or the GPL license at the Restlet Web site\(^1\). The Web site also provides a lot of information about the framework, such as tutorials of how to use the framework, the JavaDocs of the API, a short introduction to the framework itself and many other things. We used the Restlet framework in the version 1.0.1.

Realization Using Restlets

The root component of the Web service is the Application class. In our case, this is the RestApplication class which inherits from the Application class. To fully implement this class, the RestApplication has to implement the method createRoot, which returns an instance of the Restlet class. In our case, this is a Router object (we call it the root router).

To this router two other Router objects, the mapping router and the user router respectively are attached based on the URIs /mapping and /user respectively. This means that the mapping router delegates calls to the URI with the prefix /mapping and the user router delegates calls to the URI with the prefix /user.

\(^1\)Restlet Web site: http://www.restlet.org/
2.1. WEB SERVICE

Basically we have implemented one class for every feature described in section 1.2.1, each of which inherits from the Restlet class. Each class, or in other words each Restlet, overrides the specific handle method, through which the Restlet listens for incoming requests. For example one has to override the handleGet method to handle all requests sent using the HTTP GET method.

We can divide these Restlets into two types: the ones regarding the mapping store and the ones regarding the user store. The Restlets of the first type are registered at the mapping router, e.g. the MappingGetRestlet is registered at the mapping router at the address /get and therefore handles requests sent to the address /mapping/get. The Restlets of the second type are registered at the user router.

An example of the first type is the MappingAddRestlet class, which will be described in section 2.1.2 in more detail. As far as the user Restlets are concerned, we will not describe them in detail, because their implementation is very similar to the one of the mapping Restlets.

The Add-Mapping-Document Restlet

In this section we want to give a short overview on how the Connection layer has been realized on the basis of the Restlet implementation of the “Create Mapping Document” feature. Listing 2.1 shows the MappingAddRestlet class, which handles requests to store mapping documents in the repository layer.

```
public class MappingAddRestlet extends Restlet {
    private MappingRestlet end;

    public MappingAddRestlet(MappingRestlet end) {
        super(end.getContext());
        this.end = end;
    }

    public void handlePost(Request request, Response response) {
        Query query = Decoder.decode(request);
        MappingStoreQuery storeQuery = new MappingStoreQuery(query);
        UserStoreQuery userQuery = new UserStoreQuery(query);

        this.end.doAdd(storeQuery, userQuery, response);
    }
}
```

As you can see in Listing 2.1 line 4, the constructor of MappingAddRestlet expects an instance of the MappingRestlet class which can be thought of the end object of a processing chain. The MappingRestlet class processes every request regarding mapping documents such as “Add Mapping Document”, “Get Mapping Document” and so on. This way we manage the processing of all requests at a central place, but we will discuss that in section 2.1.2.
As shown in section 1.3.2, we want the MappingAddRestlet class to handle both POST and GET requests. For this we have to extend the Restlet class and override the appropriate handle methods, i.e. the handleGet and handlePost methods.

The main components of the Restlet class are the handle methods (see listing 2.2). For each HTTP method there is a corresponding handle method, that means, handleDelete, handleGet, handlePost and handlePut. Each handle method has two parameters, a Request and a Response object. The Request object represents the data of the request as an object model. The Response object is used to return the response to the request source.

Listing 2.2: Handle methods of the Restlet class

```
public void handleDelete ( Request request , Response response );
public void handleGet ( Request request , Response response );
public void handlePost ( Request request , Response response );
public void handleUpdate ( Request request , Response response );
```

Line 9 shows the handlePost method, which handles every request sent using the HTTP POST method. The decode method in the Decoder class called on line 10 extracts every data contained in the POST header and stores it in a HashMap-like class represented by the Query class. A Query object represents a request to the Repository layer. This can be an “Add” or a “Get” request for example, and so on. The Query class also offers some helper functions to access the values of the parameters represented by an instance of this class more easily.

Lines 11 and 12 show the process of extracting relevant data stored in the Query instance, which holds every data sent by the POST request. Similar to the Query class, which is in fact the superclass of the MappingStoreQuery and the UserStoreQuery class, the MappingStoreQuery object extracts only relevant data for the mapping store, whereas the UserStoreQuery object contains only relevant data for the user store. Both queries are passed to the MappingRestlet.

You may wonder why we also pass the Response object to the MappingRestlet. This is because the MappingRestlet also takes care of returning the processed data to the request source and for this the Response object is needed.

Finally we have to register the Restlet at the router to receive requests sent to a specific URI. As already mentioned in section 2.1.2, at the beginning of the delegation process there is a root router, which forwards incoming requests either to the mapping router or the user router, depending on the URI they were sent to. The mapping router has to be registered at the root router, whereas the MappingAddRestlet has to be registered at the mapping router. Listing 2.3 makes this more clear.

Listing 2.3: Routers

```
// Attach the Mapping router at the root router .
rootRouter . attach ( "/ mapping ", mappingRouter );
```
2.1. WEB SERVICE

Figure 2.3 shows the architecture of the mapping Restlets.

The MappingRestlet Class

As already mentioned the MappingRestlet class is the central class, which handles all requests regarding Mapping Documents. It processes requests received by the existing Restlets, e.g., the MappingAddRestlet, and forwards them to the Facade layer. This class also returns the results given by the Facade layer to the request source.

To show how this is done, we will carry on with the example from the previous section and explain how the doAdd method looks like. See listing 2.4 for the Java code of this method.

Listing 2.4: doAdd method of the MappingRestlet

```java
public void doAdd(MappingStoreQuery mappingQuery, UserStoreQuery userQuery,
```
```java
Response response) {
    try {
        // Get format of the mapping document.
        Format format = mappingQuery.getFormat();

        if (format == null) {
            // Set default format to OMWG.
            format = Format.OMWG;
        }

        // Check login data.
        User user = userFacade.login(userQuery);

        // Forward mapping document to facade.
        Mapping mapping = mappingFacade.add(mappingQuery, user);

        // Set response.
        response.setEntity(marshaller.marshalMapping(mapping, format));
    } catch (Exception e) {
        response.setEntity(marshaller.marshalError("There was an error storing the Mapping Document"));
    }
}
```

The parameters of this method are a `MappingStoreQuery` object, a `UserStoreQuery` object and a `Response` object. The `mappingQuery`, and respectively the `userQuery`, contain relevant data for the mapping store, respectively the user store. The `Response` object is used to return the response to the request source.

On line 5 the format parameter is extracted from the mapping store query. As shown in table 1.2 the format parameter gives the syntax of the mapping document. This parameter is required to parse and serialize the mapping document (see line 19). If no format parameter is given, the default format is set to `OMWG`, in other words, we assume at this point that `OMWG` is the syntax of the mapping document.

What we do next is to check if the user data is valid. For this we call the `login` method of the user facade and pass the user store query object (the facades will be explained in section 2.1.3). If the user data is valid an instance of the `User` class is returned, otherwise an exception will get thrown, which will be caught on line 20.

After the user has been validated, the mapping document is sent to the mapping facade. Line 16 shows how the `add` method of the mapping facade is called. Both the mapping query and the user object are passed. This is necessary for associating the mapping document with the specific user. If everything went well, that means no exception has been thrown, an instance of the `Mapping` class is returned.
Before a response is send to the request source, the mapping document is marshalled to fit the response model of the Restlet framework. This is done through the `Marshaller` class. Finally, we set the response object by calling the method `setEntity` of the response object, which then gets sent to the request source (this is done by the Restlet framework inside this method).

You might have already noticed that we do error handling by simply throwing exceptions whenever something is wrong. At a central point the exception is then caught and processed accordingly. In this case some type of error message is returned to the request source.

As you can see in listing 2.4, it is pretty easy to access the `Repository` layer, since the facades do almost all the work.

### Changes in the Restlet API

When we first used the Restlet API, the `Restlet` class offered `handle` methods for each type of HTTP request (see listing 2.2), so when we started implementing we used these methods. However, in the final release the Restlet framework reduced all these methods to a single `handle` method that handles every request regardless of the HTTP method used, as shown in Listing 2.5.

```
public void handle(Request request, Response response);
```

So when we switched to the stable release we somehow had to fix this problem. For this we have introduced the `RestletAdapter` class, an adapter class, that extends the Restlet class and overrides the `handle` method to forward the request to the specific `handle` method of the old class model. What we had to do then is to change the superclasses of each Restlet from `Restlet` to `RestletAdapter`.

### 2.1.3 The Repository Layer

The `Repository` layer provides the repository functionality, i.e. it manages mapping documents and users by providing both a `Mapping store` and a `User store`. In other words, this layer implements the functionality or business logic of the Web service.

As already mentioned in section 2.1.1, there is another layer inside the `Repository` layer, namely the `Facade` layer. The `Facade` layer consists of two components, the mapping facade, which simplifies working with the mapping store, and the user facade which simplifies working with the user store.

For the exchangeability of the repository we provide Java interfaces for all four components. Note that (normally) a separate facade implementation has to be
provided for each repository implementation, since the facades heavily depend on the repository used.

Additionally, we provide for both stores Query classes, which give access to important parameters such as 'user name' or 'mapping document'. The Mapping Store Query, respectively the User Store Query, contain relevant data for the mapping store, respectively the user store. Both classes inherit from the Query class.

A simplified UML diagram representing these classes is shown in figure 2.4.

Figure 2.4: Simplified UML of the classes of the Repository layer

As you’ve maybe already noticed, the store interfaces handle User and Mapping objects, whereas the facades cope with Query objects. This can be explained by the fact that the facades require all data to validate if everything is correct and all required data is given. The facades then pack these pieces of information in more abstract objects, namely instances of the User respectively the Mapping class, which are then forwarded to the store.

It is important to mention here that the Mapping class is an abstract class. There are two implementations of this class, the StringMapping and the URLMapping class, see figure 2.5. The StringMapping class represents a mapping document in String representation and the URLMapping class references to a mapping document via a URL. Both classes implement the abstract method getDocument,
which returns an instance of the mapping API class `MappingDocument`, by parsing the mapping document using the mapping API.

![Diagram of mapping and user classes](image)

Figure 2.5: The mapping and user classes

As you can see in figure 2.4, each mapping object has an associated user. This is, for example, required to determine if a specific user has the rights to delete or update an existing mapping document.

To give an overview of all classes mentioned so far, we want to describe them in the following list:

- **Mapping** is an abstract class representing a mapping document. There are two different implementations of the Mapping class. The StringMapping represents a mapping document in String representation, whereas the URLMapping refers to a mapping document using an URL, i.e. the URLMapping can point to mapping documents that are stored somewhere on the web.

- **User** represents a user, which can be associated with mapping documents.

- **Mapping Store** is an interface describing a store that manages mapping documents.

- **User Store** is an interface describing a store that manages users.

- **Mapping Facade** is an interface that describes a facade to manage mapping documents. This facade simplifies working with a mapping store. For example, it checks if a user has rights to perform some action on a mapping document.

- **User Facade** is an interface that describes a facade to manage users. This facade simplifies working with a user store.
Mapping Store Query contains relevant data for the mapping store.

User Store Query contains relevant data for the mapping store.

Hibernate and Apache Derby

To store and manage the mapping documents, we use the object/relational persistence framework Hibernate on top of the relational database Apache Derby [12]. Both components are implemented in Java and are therefore easy to embed in Java applications.

A main advantage of using Apache Derby is the fact, that it is not necessary to run a server to access the database. The database is stored in a directory/file structure inside the application directory. Apache Derby also offers a JDBC driver to access the database. Since we use Hibernate, it does not matter which relational database we use, as long as there is a JDBC driver for the specific database. This is also a reason, why we have decided in favour for Hibernate.

Hibernate can be downloaded under the FSF Lesser Gnu Public License at the Hibernate Web site\(^2\). Apache Derby can be downloaded under the Apache license at the Apache DB Project Web site\(^3\). We used Hibernate in the version 3.2.2 and Apache Derby in the version 10.2.2.0.

Realization using Hibernate

This sections shows how the Repository layer has been implemented using the Hibernate object/relational persistence framework. However, we will not describe how Hibernate exactly works, since this goes beyond the scope of this thesis.

We will show this by explaining how a mapping document gets stored in the mapping store. For this we will introduce the MappingManager. This class manages mapping documents stored using Hibernate on top of a relational database. Listing 2.6 shows the MappingManager with its add method.

```
public class MappingManager implements MappingStore {
    ...
    public Mapping add(Mapping mapping) throws Exception {
        // Get the Hibernate Session.
        Session session = HibernateUtil.currentSession();
        Transaction transaction = null;
```

\(^2\)Hibernate [http://www.hibernate.org/](http://www.hibernate.org/)

\(^3\)Apache Derby [http://db.apache.org/derby/](http://db.apache.org/derby/)
try {
    // Start a transaction.
    transaction = session.beginTransaction();

    // Save the Mapping.
    session.saveOrUpdate(mapping);

    // Commit the transaction.
    transaction.commit();

    return mapping;
} catch (HibernateException he) {
    // Rollback transaction.
    HibernateUtil.rollback(transaction);

    // Throw exception.
    throw he;
}
The ability of replacing the Web application is one advantage of this architecture. The REST interface (see chapter 1.3), which is provided by the Web service, makes this an easy task. Anyone who is familiar with the specification of the interface can write a Web application that makes use of our Web service. Another advantage is that you are not bound to any programming language. You can write your Web application in almost any language you want: PHP, Java, Ruby, Python, to name just a few. Further, you are not bound to Web applications. If you are more familiar with writing desktop applications in C++, C#, .NET or Java that’s fine, too.

Before we could start with the implementation of our Web application, we had to decide what programming language we wanted to use. We first thought of PHP, because both of us were familiar and had some experience with it. After some research on possible alternatives, we decided to use some other language than PHP. We wanted to try something new. That’s why our choice fell on the Web development framework Ruby on Rails.

### 2.2.1 Ruby on Rails

This section will give a quick overview of the Web application framework Ruby on Rails (often referred to as Rails), which is our framework of choice for the OMS web application. As Rails was written in Ruby, it is essential to first have a look at the Ruby language.

#### The programming language Ruby

Ruby is a reflective, dynamic, object-oriented programming language. It combines syntax inspired by Perl with Smalltalk-like object-oriented features, and also shares some features with Python, Lisp, Dylan, and CLU. Ruby is a single-pass interpreted language. Its official implementation is free software written in C. \(^6\)

The language was developed by the Japanese computer scientist *Yukihiro “Matz” Matsumoto*. He started working on Ruby in 1993 and published his work in 1995.

\(^6\)From Wikipedia, the free encyclopedia [29]
2.2. WEB APPLICATION

Matsumoto’s vision was a programming language that is easy and fun to write, and that is optimized on human aspects like readability and understanding:

“Often people, especially computer engineers, focus on the machines. They think, ‘By doing this, the machine will run faster. By doing this, the machine will run more effectively. By doing this, the machine will something something something.’ They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves.”

–Yukihiro “Matz” Matsumoto[2]

Since Ruby is published under the GPL license [7], various developers have worked and still work on it, and it has experienced continued improvements up to now. The current version (as of December 2007) is Ruby 1.8.6 which was released on March 12, 2007.

The main downside of Ruby is its performance. It is currently slower than other dynamic scripting languages like Perl, PHP or Python. However, the development team is working on that and a new, faster interpreter (YARV\(^7\)) will be implemented in future Ruby versions. Matsumoto plans to fully merge YARV into the stable version of Ruby 1.9.1, which will be released by the end of 2007.

Ruby is highly object oriented. Almost everything in Ruby is an object, even primitives and literals. Because of this, the code in Listing 2.7 works:

```
Listing 2.7: Objects in Ruby

1       -1337. abs # 1337
2       "some string". length # 11
3       "Test". index("e") # 2
4       "Nice Day Isn ’ t It?". downcase . split (/). uniq . sort . join # " 'acdeinsty"
```

Ruby tries to make things simple for a programmer. If you want to create an array, you don’t have to allocate any memory for example. Listing 2.8 shows a more complex array creation, containing integers, a string, a floating point number and another array in an array.

```
Listing 2.8: Array creation

1       a = [0 , 1, 2, 'three ', 4.44 , [5 , 6]]
```

Before we move on to Ruby on Rails, we will have a look at a last example. The code in Listing 2.9 creates a hash table and performs some basic operations with it. Notice the use of the if modifier, which says “Only delete the email if the name is Carl”. Other modifiers also exist, like unless, until, or while.

\(^7\)YARV - Yet Another Ruby VM [32]
Listing 2.9: Example showing hash creation/modification and the use of modifiers

```ruby
hash = {:id => '4', :name => 'Carl '}
hash.update({:email => 'carl@somedomain.com'})
# {:email => "carl@somedomain.com", :name => "Carl", :id => "4"}
hash.delete(:email) if hash[:name] == "Carl"
# {:name => "Carl", :id => "4"}
```

Because of Ruby’s strict object orientation, there are some oddities of the language, compared to other languages like C or Perl. As noted before, every data type is an object, even classes and primitives like integer or boolean. Even “nil” is an object. Thus, every variable is a reference to an object and every function is treated as a method. There are also some conventions every Ruby programmer has to be aware of. For example, every variable that begins with a capital letter is treated as a constant.

As of today, the popularity of Ruby is still growing. The main reason for its current success is the Ruby on Rails framework.

**Ruby on Rails**

Rails was written by the Danish programmer David Heinemeier Hansson and was presented to the public for the first time in July 2004. In 2005 he won the Google/O’Reilly "Hacker of the year" award [26] and until today he is working on several projects, including Ruby on Rails.

“Rails (Ruby on Rails) is an open source web application framework written in Ruby that closely follows the Model-View-Controller (MVC) architecture. It strives for simplicity and allowing real-world applications to be developed in less code than other frameworks and with a minimum of configuration.” From Wikipedia, the free encyclopedia [28]

Rails is based on two fundamental principles that are implemented throughout the entire framework: **Convention over Configuration** (CoC) and **Don’t repeat yourself** (DRY).

**Convention over Configuration**, as the name implies, minimizes the configurational effort by introducing all kinds of conventions. For example: If your model has a class named `Game`, Rails implies that the corresponding database table is named `Games`. This particular convention which ties an object to a table - respectively an instance of this object to a table row - is known as the **active record pattern**. Another example for naming conventions is the typical directory structure that every Rails project has. Application specific files are stored in the `app` folder, which is further split into `model`, `view` and `controller` (see figure 2.7).
Don’t repeat yourself drives the programmer to write every piece of code only once and put it in a single, unambiguous place. Strictly followed, this principle gives you a single point of failure behaviour. If your application produces an error or an uncaught exception, there is only one segment of code that has to be changed.

Figure 2.7: The typical Rails directory structure.

The OMS was the first application we developed using Rails. Due to our lack of previous experience with Rails we first had to learn Ruby as a language and become familiar with the Rails framework. During the development we got more and more into Rails and learned many new techniques and best practices.

With the release of Rails 1.2 a feature named ActionWebService caught our attention. We soon realized that this is one of many things the Rails framework offers that would perfectly suit our application. Unluckily we already made it through the early stages of development and we decided to stick to our Java web service. Now, having better knowledge of the whole subject, the option of using a different approach seems much more attractive (see Chapter 3.2). The current Rails version (as of December 2007) is 2.0.1, released on December 6, 2007.

It was clear from the beginning that our Web application had to be easy to use, clean (to avoid an overloaded look) and in touch with the user, in order to keep the Web service hidden from him/her. The user interface was therefore the starting point of our approach.
2.2.2 User Interface

Every application needs a proper UI - so does a Web application. This section will give a very short overview of the design process the ontology mapping store went through - from the first outlines to the actual realization in graphic programs and the implementation in proper technologies like CSS 2.0.

Prototypes

At the very beginning we needed to know what the application has to be capable of. Beside basic mapping-CRUD operations the Web application should offer a search and some basic user management. First sketches were drawn, realized in Photoshop, and later on implemented in the alpha development stage. After all, we were not happy with the look of the interface - a redesign of the application was necessary.

The second attempt of a proper user interface was pretty good. The look&feel of the UI was unchanged during development and provided access to all features that made it to the final version. During the process of development the interface became very overloaded and did show lots of irrelevant information. We decided to redesign the application again.

![Figure 2.8: Evolution of the OMS Web application](image)

Final User Interface

We wanted the latest version of the user interface to be as clear and structured as possible. After analysing the application over and over again we came to the conclusion that searching for mappings is the most important feature - so the front page was designed to be a clear search-interface à la Google.
2.2. WEB APPLICATION

2.2.3 Basic Functionality

The following pages will give an overview of the functionality of the web application. The functions of the OMS can be split up into two parts: Basic functions that can be used by everybody and advanced functionality for users that have an account and are logged in. The following sections (2.2.3 and 2.2.4) will show the possible actions that can be done with the Web application. Because the Web application relies on the Web service, the business logic behind those actions will be presented in section 2.2.6.

The basic functions are functions that every user has access to, regardless if he/she is logged in or not. When visiting the home page of the application, the front page is shown (see Figure 2.9). Besides information regarding the user status (logged in or logged out) the user has the ability to access all basic functions and information pages, which display contact information and a short introduction to the OMS. The front page also does include an interface to perform search operations.

Search

There are two types of search interfaces. They differ look- and handling-wise, but are equally powerful.

Basic Search is accessible via the front page or directly via the basic search page. A simple query containing only strings will search for mappings that contain the given string(s) in their ontology URIs. However, applying various options can extend the basic search behaviour, so that an advanced search can be performed via a single-line search interface. For further details on search parameters and how to use them, see section 2.2.5. Mappings matching the search criteria will be listed in the results view.
Advanced Search is only accessible via the advanced search page and should provide the user with an easier way to perform more complex queries than using the basic search. It is more accessible and easier to understand for the not so experienced users.

It is also possible to include wildcards [*] in your search, so that computerg* will match any mapping that has the term computer followed by any character - for example computergame, computergames or even computergeek. Again, mappings matching the search criteria will be listed in the results view.

Unless there are no mappings that fit the query, the results view will show a list of mappings that are returned by the Web service. The mappings that are returned can further be filtered by sorting them using their different properties.

If the user is logged in and is owning one of those mappings, he/she has the ability to modify or delete his submissions. If the user is logged out or the mapping is not his own, he will only be able to open, i.e. read the mapping.

Browse

Another useful feature of the Web application is the browse page. The browse page will show the last mappings that were submitted to the store (New mappings). The user also has the option to view mappings that have been modified lately (Latest changes). Mappings can be retrieved alphabetically, for example by clicking on 'A' there are only mappings shown that start with an A (e.g. http://awesomedomain.com/some_mapping). The resulting list of mappings gets paginated automatically, the first page is shown by default. Further actions like sorting the results is - equally to the search view - also possible.

Account Creation

To add mappings to the database and manage them, authentication is needed. If anonymous access was possible, everyone would be able to edit or delete mappings. There are situations (like sites who build up on user generated content) where anonymous access is desirable. If this is the case, a CVS 8 is necessary. However, for our application the easiest thing to do is to introduce some kind of user management.

To avoid automated user creation, the user has to fill out a CAPTCHA 9 that is automatically generated (see figure 2.10). We use the free available service Captchator to generate the CAPTCHAs. The service provides randomly generated images via a REST interface and is also capable of checking whether the user input is correct or not. For more information see the Captchator Homepage [1].

---

8Concurrent Versions System
9Completely Automated Public Turing test to tell Computers and Humans Apart
2.2. WEB APPLICATION

Once the user has created his/her account, he/she has access to the advanced functionality of the OMS.

2.2.4 Advanced Functionality

As mentioned before, advanced actions are actions that alter the database, thus require user authentication. A user that is logged in to our system can create a mapping, manage his/her mappings and edit personal information.

Add Mapping

There are two possibilities of adding a mapping. The first possibility is to store a mapping that is already online. A link to the mapping (given by the user) is read by the application and given to the Web service. The Web service only stores this URI (i.e. the reference to the mapping) and parses the mapping every time this mapping is requested. If the linked mapping or the link itself is invalid, the Web service will return an error which is caught by the application. The second possibility is to upload a mapping. The user can choose between writing (or pasting) a mapping in a form or uploading it from the local machine.

Manage Mappings

Managing operations include editing and deleting mapping documents that are owned by the user who is performing the operation. If the user fails authentication, the Web application catches the error returned by the Web service (see listing 1.11) and shows the error message to the user. An example of such an error message is shown in figure 2.11.

Figure 2.10: A CAPTCHA that says mfsap
Before the mapping is submitted, the syntax of the mapping document is checked. As noted in our specification (section 1.3.2), there are different outcomes whether adding a mapping fails or not. An error like the one shown in figure 2.11 will be displayed if the syntax check did not pass.

Manage Account

To give the user the possibility to change the user password, some kind of account management had to be implemented. The Web service offers fitting functions for those actions (see section 1.3.2). The Web application uses these functions to alter account information. For actions like changing the user password or e-mail address, AJAX comes in handy, as this kind of manipulation is done “on the fly”. This means that there is no need of an external link to a “Change e-mail” page. The e-mail address can be changed directly, while viewing the account information. This results in a more dynamic user experience and an uninterrupted workflow.

This part of the thesis was intended to give a basic overview of the functionality. For a deeper view on how the application uses the Web service to alter the OMS see section 2.2.6.

2.2.5 Query Language

The Query Language came hand in hand with the Advanced search. The purpose of the OMS Query Language is to provide extra search options without using the advanced interface. Advanced search options include mapping format, mapping ID, username, and a specified search for the mapped ontologies (referred to Ontology 1 and Ontology 2).

The full syntax of the Query Language is as follows:

```
-o1:onto1 -o2:onto2 -f:omwg -id:http://www.example.com/example# -usr:user
```

The example above contains all options that are available for a query. The options for the mapping-format (-f:), the mapping-id (-id:) and the user who
provided the mapping (-usr:) are optional. The options for Ontology 1 and 2
are optional too, but have special treatment if not provided. If neither of the
two are supplied then the string that is provided at the beginning (and is not
an option) will be interpreted as Ontology 1 OR Ontology 2.

search_strings -f:omwg -id:http://www.example.com/example# -usr: user

Furthermore, if a “basic” search is performed, the search string will again be
interpreted as Ontology 1 OR Ontology 2.

search_strings

Note that the application will not search for any other mapping-properties be-
side Ontology 1 or 2. For example, if you want to search for mappings that
were uploaded by a specific user, you have to provide the related option -usr:
followed by the username.

Examples

Here are some examples to give a better feeling of how the query language works.
First of all, let’s define two mappings that are stored in our database.

Ontology 1

http://url/game/video# http://some_url/computergame#
Bob

Ontology 2

http://some_url/computergame# http://another_url/entertainment/videogames#

Figure 2.12: Two example mappings from our database.

In the following they will be referred to as mapping1 and mapping2. The
blue boxes are the ontologies (ontology1 on the left, ontology2 on the right side).
The pink box on each mapping refers to the user that created the mapping.

Let’s start with the most basic search you can perform.

entertainment

This query will return any mapping that is linked to an ontology that has the
term entertainment in its URI. In our case this will be mapping2, because its
ontology2 contains the search string (http://another_url/entertainment/videogames#).
Another basic example would be the following:

```
game
```

The term `game` is contained in all ontologies, thus `mapping1` and `mapping2` would be returned. Matches are again highlighted in red.


If too many results are returned, it is possible to start a more specific search. One possibility is to search for ontology1 only, for example:

```
-o1: entertainment
```

The result will be empty, because neither mapping1 nor mapping2 contains the string `entertainment` in ontology1.

Another search parameter is the `-usr` parameter. When provided, the mappings returned were submitted by the specified user.

```
game -usr: alice
```

In this example, only `mapping1` will be returned, because mapping2 was created by Bob.

### 2.2.6 Connection to Web Service

The Web application is 'connected' to the Web service via our REST interface. All information that is exchanged via the application and the service is transferred via the HTTP protocol. This section describes how the Web application uses the Web service, how HTTP requests are formed and how the results are parsed. The operations used are specified in section 1.3.2.

**Initial actions**

The first thing the application needs to do is to get data from the user, e.g. a string from a search field. This string is saved to the appropriate model in the Rails framework. If necessary, the string is checked for validity. For example, the user is forced to enter a valid username (no special characters and at least
three characters long) and a valid e-mail address. Those validity checks are done by regular expressions. An example for the regular expression that validates e-mail addresses is given in listing 2.10. The code is part of the create_check function in the user_controller.

Listing 2.10: Regular expression that validates an e-mail address

```ruby
elsif email[/^([-\@\s]+)@((?:[-a-z0-9]+\.)+[a-z\s]{2,})$/i] == nil
  render_text "Please enter a valid email address."
```

When the data passes all checks, a HTTP request can be formed.

Forming Requests

HTTP requests are formed by using the Net::HTTP library [10] which comes with Ruby. This library has various functions implemented, including methods that are designed to form requests, which are of interest for using our REST interface. It provides functions to access WWW documents via HTTP, Hyper Text Transfer Protocol version 1.1.

The HTTP request is formed by calling one of those methods. As a result there is a Net::HTTPResponse object returned, which wraps the HTTP response header. There are many response codes with different meaning. The most familiar one may be 404 HTTPNotFound, which indicates that the file was not found on the server. Our Web application checks the Net::HTTPResponse object and only succeeds when the response is HTTPSuccess (2xx) or HTTPRedirection (3xx).

Since the PUT method is not implemented properly in the Net::HTTP library, we decided to use the alternative specification of the REST interface when necessary. So there are only GET and POST requests used in our application. GET requests are used when user information or mapping documents are retrieved from the Web service. POST requests are used when mapping documents are stored or edited, or when a user account is created or user information is edited.

A GET request is formed using the OpenURI Module in open-uri.rb which is included in the Ruby core. OpenURI is a wrapper for HTTP, HTTPS and FTP. A usage example is given in listing 2.11.

Listing 2.11: Forming a GET request

```ruby
url_to_invoke = 'http://www.example.com/example_mapping';
open(url_to_invoke) {
  |page| page_content = page.read()
  # Parse or print the response that is stored in 'page_content'
  ...
}
A **POST request** is formed using the `Net::HTTP.post_form` method which is included in the Ruby `Net::HTTP` library. The method does basically the same things as a simple HTML form that uses POST as the submit action. It takes named parameters that have values represented as strings and passes them via a POST request to the given URI. A usage example is given in listing 2.12

```ruby
Listing 2.12: Forming a POST request

```url_to_invoke = 'http://www.example.com/search.cgi';
res = HTTP.post_form URI.parse(url_to_invoke),
  { "query" => "http specification",
    "language" => "en" }
#
```

Beside the `Net::HTTP` library, the `URI` library [11] has to be loaded in order to use the `post_form` method. The `URI` library is capable of converting strings to `URI` objects. It also provides further functionality like splitting the URI into various parts (scheme, userinfo, host, port, etc.).

To see how these requests are actually formed in our application see the following example, which will give insight to all necessary steps to submit a mapping.

**Example: Submit a mapping**

Listing 2.13: The first step is to read the parameters that the user has entered. We have already checked that the user is logged in and has the privileges to submit a mapping.

```ruby
Listing 2.13: Store parameters

```submit = params[:submit] unless params[:submit].nil?

```ruby
Listing 2.14: After the parameters are stored, we are able to process the data. Since there are two possibilities of adding a mapping (see section 2.2.4) we have to choose where the mapping file is located.

```ruby
Listing 2.14: Read the mapping file

```
2.2. WEB APPLICATION

Listing 2.15: Now the POST request can be formed. \texttt{Net::HTTP.post_form} expects a URI and a hash of parameters as its arguments. The parameters used are those specified in table 1.2. As you can see in lines 3 and 4, the user information is also sent to the Web service.

```ruby
# Build the POST Request
res = Net::HTTP.post_form(URI.parse("#{ $HOST_ADDR } mapping"),
{ 'userName' => @session['username'],
  'userPassword' => @session['password'],
  'format' => @mapping_format,
  'mapping' => @mapping_text, 'URL' => @mapping_url })
```

Listing 2.16: The last thing to do is to check the \texttt{Net::HTTPResponse} that was given by the Web service. Erroneous responses are caught and an error message is shown which tells the user what went wrong. If the response shows that the request was successful, the user will be notified that everything went fine.

```ruby
# Was the POST Request successful?
case res
  when Net::HTTPSuccess, Net::HTTPRedirection
    # ... Handle error and success scenarios
```

Parsing the results

Unless the response is a single mapping document, the Web service always returns an XML document as a result. The XML document is for example a list of mappings or a list of mapping metadata, which has to be parsed (see listing 1.5). We are using \texttt{hpricot} to parse the XML documents. Hpricot is a vendor plugin for \texttt{Rails} and is published under an open source license. Hpricot is a very flexible HTML parser written by \textit{why the lucky stiff}. \textit{Why the lucky stiff} (often called \texttt{why} or simply \texttt{why}) is along Yukihiro Matsumoto and David Heinemeier Hansson a key figure in the Ruby community.\footnote{\texttt{why} \url{http://whytheluckystiff.net/}}. The parser is based on Tanaka Akira’s HTree\footnote{HTree - HTML/XML tree library \url{http://www.a-k-r.org/htree/}} and John Resig’s JQuery\footnote{JQuery is a new type of JavaScript library \url{http://jquery.com/}}, but with the scanner recoded in C (using Ragel\footnote{Ragel State Machine Compiler \url{http://www.cs.queensu.ca/~thurston/ragel/}} for scanning).\footnote{[23]}
Hpricot first loads an XML page. After that, it can search for specific elements or fetch the content of an element. For example, if we want to retrieve the mapping metadata we need to loop through every mapping and scan for the metadata tags. If the metadata tags are present, we parse the metadata and save it (see listing 2.17).

Listing 2.17: Parse result with hpricot

```ruby
1  doc = Hpricot(open(url))
2  
3  # for each mapping - get the METADATA
4  (doc/"mapping-meta-data").each do |mapping|
5    id = (mapping/"mappingid").first.inner_html  
6      unless (mapping/"mappingid").first.nil?
7    source = (mapping/"source").first.inner_html
8      unless (mapping/"source").first.nil?
9    target = (mapping/"target").first.inner_html
10   unless (mapping/"target").first.nil?
11    username = (mapping/"name").first.inner_html
12      unless (mapping/"name").first.nil?
13    modified = (mapping/"lastupdate").first.inner_html
14      unless (mapping/"lastupdate").first.nil?
15  
16  # Save the result
17  ...
18 end
```

The `doc` object represents the document itself, and can be interpreted as a root tag that embraces the whole XML document. The code in listing 2.17 takes this root tag and looks for 'mapping-meta-data' tags. If there are such tags, the content of the metadata tags `id`, `source`, `target`, `username` and `modified` is retrieved for each set of mapping metadata.
Chapter 3

Conclusion

In this thesis we have presented the Ontology Mapping Store (OMS), a repository for ontology mapping documents, consisting of a Web service and a Web application. The Web service provides the functionality (e.g. query for mapping documents) of the repository through a RESTful interface. Based on this Web service, the Web application enables the user to make use of the Web service on a well structured and easy-to-use graphical interface.

In the next section we want to discuss some problems and flaws which have occurred during and after implementation of the Ontology Mapping Store. In section 3.2 we present a restructured global architecture that would solve some of these issues.

3.1 Problems

During and especially after development we have found some flaws in both the Web service and the Web application.

As already mentioned in section 2.1.2 the Restlet framework experienced major changes when switching from beta to final, which led to major changes in the Web service implementation as well. Additionally, some concepts of the Restlet framework were not easily applicable for our implementation of a RESTful Web service. In a RESTful architecture and therefore in the Restlet framework, resources play an important role. However, in our solution we are working with methods or operations rather than resources. Therefore, we did not use the Restlet framework as it is designed to be used. In other words, we should have used the class Resource instead of the Restlet class.
It was hard to work with the mapping API in the version that was available when we implemented our program. For instance, the parser that was used could only parse documents in one specific syntax. Additionally, the parser could not handle concurrent calls. For this we had to implement an adapter class, that used a mutex for synchronisation of the parser.

The biggest problem regarding the Web application was the global architecture we defined at the beginning of our thesis. One requirement of our work was the separation of the Web service and the Web application. With this precondition given we began to implement both parts of our application independently. After a while we found out that it would have been better to implement both the Web application and the Web service in Ruby on Rails or a similar framework like Django, Zope or TurboGears.

The current architecture forces us to rip Rails in half. The Rails architecture consists of the Action Pack which handles routing, manages the application controllers and the views. The Active Record part of Rails does an object relational mapping between the model and the database. Figure 3.1 shows how strict Rails realizes the MVC architecture.

As mentioned before, our current architecture does not allow us to use the whole functionality of Rails. We have implemented the Web service in Java, mainly because the mapping API was written in Java too. This circumstance forces us to use a different approach for data manipulation functions as the Rails one. Listing 2.15 and listing 2.16 show how a typical request is formed to create or update an entry in our database. A typical approach as shown in listing 3.1 would be much shorter and simpler. There are plenty of other functions in the ActiveRecord::Base class which allow us to manipulate database entries.

Figure 3.1: The Rails MVC structure

\footnote{See the Rails Framework Documentation for further details http://api.rubyonrails.org/}
3.2. POSSIBLE IMPROVEMENTS

Listing 3.1: Create an object and save it the Rails way

```ruby
# Create mapping object and store it in the database
mapping = Mapping.new(...)

mapping.save() # store the object in the database
```

Hence, retrieving a page from the Java Web service and parsing the response with *hpricot* is redundant, because the object can be directly accessed via the *Active Record* pattern. Using our current architecture we lose the whole Web service and Active Record part of *Rails*, including direct Web service and database integration (see figure 3.2).

![Diagram](image.png)

Figure 3.2: Limitations to Rails by our application architecture

### 3.2 Possible Improvements

In this section we want to briefly present a restructured global architecture of the OMS, which tries to fix the flaws we have pointed out in the previous section. See figure 3.3 for a graphical representation of the new architecture.

The idea of the new global architecture is that we split the Ontology Mapping Store in two applications:

**Web application and Web service** One single *Ruby on Rails* application which provides the functionality of both the Web service and the Web application of the OMS. This application manages the users and mapping documents in the database using the features given by *Ruby on Rails*. However, it does not directly use the mapping API to process the mapping documents but interacts with the Mapping Parser Web service.
Mapping Parser Web service  A Web service written in Java, which gives access to functionality of the mapping API, e.g. serialize mapping documents in different formats. This Web service can be accessed over various interfaces such as a restful HTTP interface.

With this approach we can use the whole functionality of Ruby on Rails including its features to create both Web applications and Web services. Since in Ruby on Rails it is straightforward to integrate a Web service in an existing workflow or application, we could easily exchange data with the Mapping Parser Web service.
List of Figures

1.1 Ontology Examples ............................................. 3
1.2 Mapping Example: Computer Games ......................... 4
1.3 Global architecture ............................................ 7

2.1 Web Service architecture ................................. 16
2.2 Final Web service architecture .......................... 16
2.3 Mapping Restlets .......................................... 21
2.4 Simplified UML of the classes of the Repository layer .... 24
2.5 The mapping and user classes ............................. 25
2.6 OMS Web application architecture ....................... 28
2.7 The typical Rails directory structure ....................... 31
2.8 Evolution of the OMS Web application .................... 32
2.9 OMS Web Application: Final User Interface .............. 33
2.10 A CAPTCHA that says mfsap ................................. 35
2.11 Error shown by the Web application ...................... 36
2.12 Two example mappings from our database .............. 37

3.1 The Rails MVC structure .................................. 44
3.2 Limitations to Rails by our application architecture .... 45
3.3 Restructured global architecture .......................... 46
LIST OF FIGURES
List of Tables

1.1 Analogies .......................... 10
1.2 Parameters used by mapping operations .................. 12
1.3 Parameters used by user operations ..................... 14
Bibliography


