FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Easy RDF for PHP (ERP) API

A PHP API for processing RDF Resources

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Alexander Aigner
Matrikelnummer 0625287

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Hannes Werthner
Mitwirkung: Projektass. Mag.rer.soc.oec. Birgit Dippelreiter

Wien, 25.01.2012

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Easy RDF for PHP (ERP) API

A PHP API for processing RDF Resources

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Alexander Aigner
Registration Number 0625287

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ. Prof. Dr. Hannes Werthner
Assistance: Projektass. Mag.rer.soc.oec. Birgit Dippelreiter

Vienna, 25.01.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Alexander Aigner
Ospelgasse 12-14/6/8, 1200 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

I would like to thank a number of people who have supported me while writing this masters thesis. First of all, I want to thank Prof, Dr. Hannes Werthner who gave me the possibility to write this work. Further, I want to thank Mag. Birgit Dippelreiter for her continuing support and all the valuable suggestions during writing my thesis. At last, I would like to thank to my family and my girlfriend for their support and for repedently proofreading this work.

Alexander Aigner

Abstract

The vision of the Semantic Web and its concepts represent the future of the Web. Application programming interfaces (APIs), which simplify the integration of semantic technologies, are an important part in realizing this vision. The Resource Description Framework (RDF) defines one of these semantic technologies, which is used to describe the resources of the Web to make them interpretable by machines. While RDF APIs are getting more popular amongst programming languages like Java or .Net, there is still a lack of efficient and satisfying RDF interfaces for PHP. Considering that PHP is the fourth most used programming language for Web development, it is necessary to provide usable interfaces. Therefore, we decided to create an easy to use RDF API for PHP, namely, the "*Easy RDF for PHP (ERP)*" API. By evaluating and comparing popular RDF APIs from various programming languages, we were able to extract strengths and weaknesses, which were considered during the development of our API. This enabled us to introduce valuable, but previously unused concepts to PHP, creating an efficient and easy to use API. Further, the comparison illustrates that most APIs do not provide efficient and satisfying interfaces for RDF. In a final comparison, it is illustrated that the ERP API achieves high ratings in the characteristics efficiency, effectiveness and satisfaction and, therefore, it presents opportunities for the development of other APIs.

Kurzfassung

Die Vision des Semantischen Webs und dessen Konzepten repräsentiert die Zukunft des Webs. Application Programming Interfaces (APIs), welche die Integration semantischen Technologien vereinfachen, repräsentieren einen wichtigen Teil zur Realisierung dieser Vision. Das Resource Description Framework (RDF) ist eine von diesen semantischen Technologien, welche es ermöglicht Ressourcen aus dem Web zu beschreiben und dadurch für Maschinen interpretierbar zu machen. Während RDF APIs bereits vermehrt für Programmiersprachen wie Java oder dotNet existieren, besteht immer noch ein Mangel an effizienten und zufriedenstellenden Schnittstellen für PHP. Wenn man bedenkt, dass PHP die viert häufigst verwendete Programmiersprache im Bereich Webentwicklung ist, ist es notwendig, auch für diese Programmiersprache brauchbare Schnittstellen zur Verfügung zu stellen. Deswegen haben wir uns entschieden, eine leicht zu verwendende RDF API für PHP zu entwickeln. Diese API nennt sich "*Easy RDF for PHP (ERP)*" API. Durch evaluieren und vergleichen von bereits existierenden und weitverbreiteten RDF APIs, basierend auf verschiedenen Programmiersprachen, haben wir es geschafft, einige Stärken und Schwächen zu definieren, welche während der Entwicklung unserer API beachtet wurden. Dies ermöglichte uns wertvolle Konzepte in die ERP API einzubringen, welche zuvor noch nicht in PHP verwendet wurden. Weiters zeigt der Vergleich, dass die meisten APIs keine effizienten und zufriedenstellenden Schnittstellen für RDF zur Verfügung stellen. Eine abschließende Gegenüberstellung illustriert, dass die ERP API eine hohe Bewertung in den Kategorien Effektivität, Effizienz und Zufriedenheit erreicht und deshalb einige Verbesserungsmöglichkeiten für die Entwicklung anderer APIs präsentiert.

Acronyms

API	Application Programming Interface
ARC	Appmosphere RDF Classes
CSV	Comma-Separated Values
ERP	Easy RDF for PHP
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
JavaDoc	Java Documentor
JSON	JavaScript Object Notation
Nt	N-Triple
OOP	Object-Oriented Programming
OWL	Web Ontology Language
PHP	PHP: Hypertext Preprocessor
PHPDoc	PHP Documentor
RAP	RDF API for PHP
RDF	Resource Description Framework
RDFS	Resource Description Framework Shema
RIF	Rule Interchange Format
RSS	Really Simple Syndication
SGML	Standard Generalized Markup Language
SPARQL	SPARQL Protocol and RDF Query Language
SWRL	Semantic Web Rule Language
UCS	Universal Character Set
URI	Uniform Resource Identifier

URL	Uniform Resource Locator
URN	Uniform Resource Name
UTF	Unicode Transformation Format
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	Extensible Markup Language
XMLS	Extensible Markup Language Shema

Contents

CHAPTER 1

Introduction

Most of the services we use daily, heavily depend on technologies and information stored on systems around the world. Therefore, the Internet, which connects companies and people around the world, is one of the most important technologies of our times. By allowing us to exchange, publish or search for all kinds of information, it has great impact in our lives.

Even though, there are lot of different uses for the Internet, the Web may be its most popular one. While the Internet can be seen as the physical connection between different computer networks around the world, the Web defines the concepts which allows users to access information, which is stored on computers in these networks. Therefore, the web contains a vast amount of information, which is accessible from all around the world.

Unfortunately, the Web, as it is known these days, does not allow us to use its information efficiently. Most of the information on the Web is unstructured and thus "inaccessible" for machines. To enable machines to access and interpret information on the Web, the vision of the Semantic Web was born.

The Semantic Web represents the future of the Web and aims to extend it rather than to replace it. It can be seen as a collection of technologies that aim to describe the accessible resources on the Web, therefore, making them processable by machines. Semantic technologies encode meanings separately from data. Thus, it enables machines as well as humans to understand, share and process them at execution time.

Application Programming Interfaces (APIs) that support semantic technologies are an important part in realizing this vision. It is necessary that Web developers are provided by APIs, which simplify the integration of these technologies within applications. Unfortunately, we argue as long as such APIs don't provide efficient and easy to use interfaces, developers will continue avoiding these technologies. The majority of Web sites are created by a small group of people, often just one person. Nowadays, the integration of sematic technologies into Web applications may not bring any real benefit for such developers. Further, if the APIs are slow or hard to include, developers will not spend the extra efforts to use them.

The *Resource Description Framework (RDF)* is one of such semantic technologies. It is a framework for describing the Web's accessible resources and an important part of the Semantic Web. While RDF APIs are already more popular amongst programming languages like Java or .Net, there is still a lack of efficient and satisfying APIs for PHP (*PHP: Hypertext Preprocessor*) . Considering that PHP is the fourth most used programming language for Web development, it is necessary to provide usable interfaces to developers.

Aim of this work is to analyze the current situation of RDF APIs for PHP and create an easy to use API (for the average users). This API is called *"Easy RDF for PHP (ERP)"* API. To achieve high usability, we perform a comparison of all PHP APIs and popular APIs of other programming languages. This comparison points out opportunities and requirements for the development of the ERP API. Furthermore, it allowed us to extract valuable concepts for the proposed API, which were not implemented before, but improve the usability significantly.

The API supports all basic manipulation methods such as adding, deleting and editing RDF resources. Besides a simple parameter search the API also supports the popular *SPARQL (SPARQL Protocol and RDF Query Language)* language. Additionally it is possible to to use concepts of *RDFS (Resource Description Framework Shema)* and *OWL (Web Ontology Language)*.

By using object-oriented programming paradigms, we were able to develop an API that provides an easy to use interface without resigning on powerful functionality. Therefore, inexperienced as well as experienced are able to use this API.

The work is structured as follows: First, we define basic principals and concepts

that are necessary for understanding the work. Next, we introduce APIs from PHP and other programming languages. After that, we evaluate these APIs using the ISO/IEC 25000 quality in use model and use this evaluation for comparing these APIs. Then we highlight the ERP API as well as the objectives and motivations for its development. We also illustrate and describe the usage of the ERP API by presenting a few examples. Finally, the ERP API is evaluated using the ISO/IEC 25000 quality in use model and compared to the ARC API. For winding up the work we conclude our research and describe ways of improving the API in the future.

CHAPTER 2

Basic principles

Target of this chapter is to specify the basic principles of necessary technologies and concepts that are important for understanding the practical part of this work. Besides specifying technologies and concepts, it is also an aim to delineate their relations and dependencies. Further, we mention their role in nowadays society and information technology.

First of all, we explain the Internet and the Web as well as their relationship. Next, we introduce the idea of Semantic Web and highlight its concepts and technologies by introducing and describing the Semantic Web stack. Last, we outline the most important concepts regarding our prototype. These are RDF, PHP and API.

2.1 The Internet

The *Internet* (from interconnected network) is a global system of interconnected computer networks. Using electronic, wireless and optical networking technologies it connects billions of users around the world. The Internet allows users to interchange a vast range of information and other resources. Further, it enables a huge number of services used regularly by its users, such as downloading and reading the inter-linked hypertext documents of the Web or enabling the infrastructure of electronic mail. The Internet can be considered as the physical foundation of the *Web* [50].

2.2 The Web

The Web (W3, WWW or World-Wide-Web) was developed in 1989 by physicians and engineers at CERN (an European Particle Physics Laboratory in Geneva, Switzerland). The early Web enabled employees and collaborators to exchange ideas and information about common projects. Besides exchanging information, it aimed to enable linkage between related information. This allowed its users to extract combined or even new knowledge. Linkage between information is defined by the Web's property called scaling. Since scaling wasn't an issue, soon developers started developing new features for the Web. Scaling allowed the Web to expand rapidly from its origins at CERN across the whole world. In 1991 the Web was introduced to the public [4].

Another aim of the development of the Web was to end incompatibilities between the different computer systems that were available in that time. The Web aims to share, for example, interlinked pages of text, images, animations, sounds or videos, everything independent of server or client architecture. Before the Web access to network information was dominated by system requirements and different command languages[1] [4, 6].

Nowadays, information on the Web is mainly stored in so called *hypertext documents*. The term hypertext (already mentioned 1945 in the article 'As We May Think' [11]) defines an easy-to-use and flexible format for sharing and interlinking information. Hypertext documents define the structure of the Web and are considered the underlying concept of the Web as we know it [4, 44].

The success of the Web is based on three core concepts [4, 7] that are still important in the present. These technologies are called *Uniform Resource Identifier (URI)*, *Hypertext Transfer Protocol (HTTP)* and *Hypertext Markup Language (HTML)*.

URI describes the concept of a special string used to completely identify a resource on the Web. Further explanation of this concept is provided in section 2.3.

HTTP is an alternative to the *File Transfer Protocol (FTP)*. It defines a networking protocol for distributed, collaborative and hypermediated information systems for

[1] A command language is a domain-specific interpreted language. Common examples of command languages are shell or batch programming languages. These languages can be used directly at the command line, but can also automate tasks that would normally be performed manually at the command line.

transferring hypertext documents over a network. Since FTP's target is to provide a reliable protocol for transferring files, it does not provide the speed required for hypertext documents. HTTP slightly loosened the requirements of reliability in tradeoff with increased speed and thus, enabling the speed that is necessary for fast traversal of hypertext links [4, 44].

HTML is defined as a data format for interchanging hypertext by allowing it to be transmitted over a network. HTML is used to describe and structure information and can be seen as the building blocks of hypertext documents. Even though HTML was developed in 1990, it is still important in the present. Nowadays there exist a variety of server-side programming languages that present the backbone of Web sites. Anyway, most of the server-side results are still transferred using the HTML format [4, 44]. One such server-side programming language, used for more complex tasks is PHP (see section 2.5)

With the development of the Web new datatypes and protocols arise. This leaded to concerns of fragmentation of the Web. Fragmentation could have destroyed the universe of interlinked information that evolved during the usage of the Web. To counter this trend, the *World Wide Web Consortium (W3C)* was founded in 1994. These days, the W3C has about 150 members, including major Web technology developers and Web depended organizations. The W3C tries to introduce common and generally accepted standards of Web technologies and concepts [4]. The combination of these Web technologies will enable the *Semantic Web*.

2.3 The Semantic Web

While the Web is linking hypertext documents (Web of documents), the *Semantic Web* refers to W3C's vision of the Web of linked data [52].

Originally, the Web intended to provide information equally to human users and machines. Unfortunately, nowadays the majority of this information is only intended to be readable by human users. Humans are able to use different sources of information (for example, hypertext documents) and to put them in relation with each other. For machines this process is much harder or sometimes even impossible. For instance,

consider a hypertext document that contains the information that Vienna is the capital of Austria and another document that contains that Austria is a country of Europe. Human users understand the information and might even infer that Vienna is in Europe. Most machines can't extract or even process such information if it is not further described by using semantic technologies [52].

This leads to the fact that even though the Web carries a vast amount of information, it is not easy to find or even process it. Even search engines that use complex statistical methods are not able to find all relevant information on a certain topic. Most search engines use an approach of comparing query strings with strings contained in hypertext documents. Unfortunately, often information is stored in databases and dynamically included into hypertext documents. These dynamics combined with the usage of different technologies make it hard to enclose such information. Therefore, a need emerged of a more context-based and less text-based search: a semantic search [52].

The goal of the Semantic Web is to transform the Web from a linked document repository into a distributed knowledge base. That would improve the effectivity of exploiting the Web's vast amount of information and services, especially for machines [20, 52].

To achieve such transformation the W3C developed languages to describe the Web's accessible resources (the fundamental elements of the Web). These languages are designed to capture information and, therefore, enable applications to gain a better understanding of the resources. If machines are able to understand information on the Web, they can use them more intelligently. Semantic Web technologies aim to structure the meaningful content of Web pages, creating an environment where software agents can carry out sophisticated tasks for users [5, 20, 52].

To illustrate the relation of Semantic Web technologies Tim Berners-Lee introduced the Semantic Web stack presented in figure 2.1. It highlights that the vision of the Semantic Web is based on different concepts and technologies. Only if all these concepts and technologies are realized the vision of the Semantic Web can come reality. The Semantic Web stack can be divided into three categories [32]:

1. Hypertext Web technologies,

2. Standardized Semantic Web technologies and

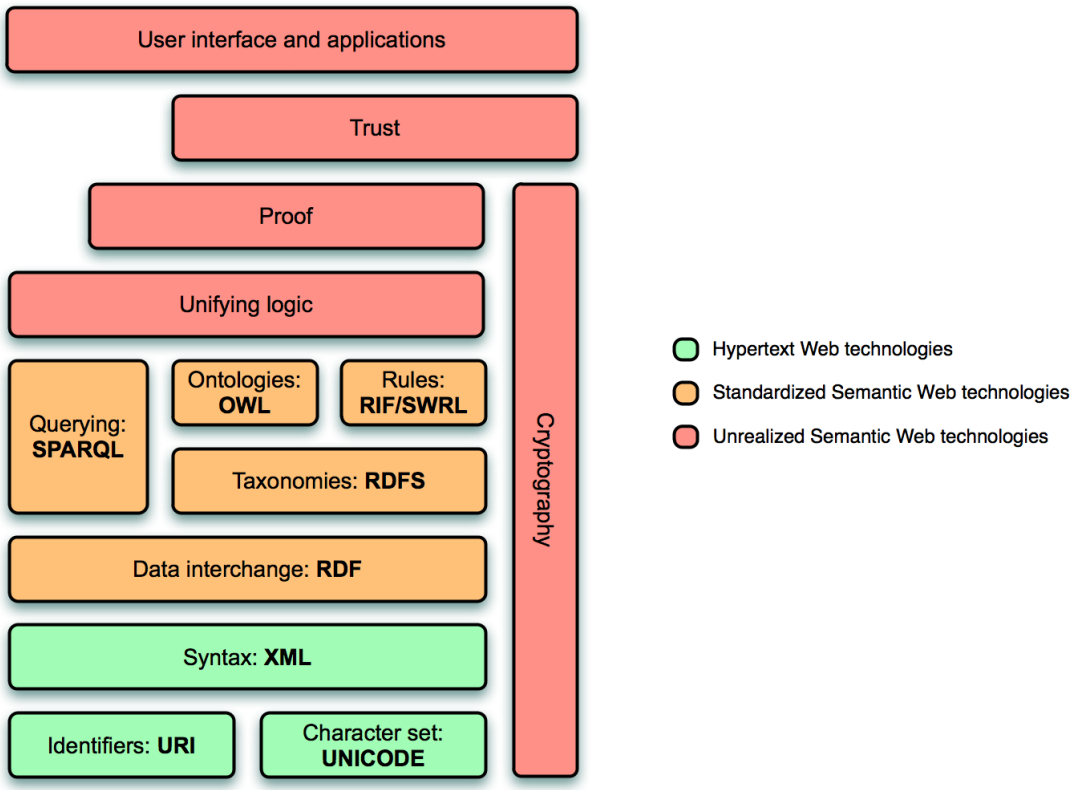


Figure 2.1: Semantic Web Stack (modified according to [32]).

3. Unrealized Semantic Web technologies

These categories and their corresponding technologies and concepts are described in more detail in the following.

2.3.1 Hypertext Web technologies

The two bottom layers of the Semantic Web stack describe technologies that are already well known from the original hypertext Web. The concepts of URI, Unicode and XML define the building blocks and, therefore, the foundation of the Semantic Web. This category also shows that the Semantic Web intends to be an extension of the existing Web rather than a replacement.

URI

As mentioned before, an Uniform Resource Identifier (URI) describes the concept of a string used for completely identifying a resource. URIs are commonly used in the Web, for example, for identifying hypertext documents, files or other resources. As the concepts of HTML, HTTP matured and the functionalities of Web browser increased, a need emerged to distinguish:

1. a string that provided a location of a resource or
2. a string that merely named a resource.

The term *Uniform Resource Locator (URL)* represents the first and the more contentious *Uniform Resource Name (URN)* the second option.

URLs describe a resource by a special string that represents the location of the resource. Originally, URLs were the only kinds of URIs. Therefore, the two terms are often used synonymical.

The concept of URNs was to identify a resource by a freely chosen name.

Originally, an URI was intended to be part of either an URL or an URN. Since URIs were developed that didn't fit in any of these classes, the strict classification was relinquished. Nowadays, URLs and URNs are considered a partition of URIs [4, 7] as illustrated in figure 2.2.

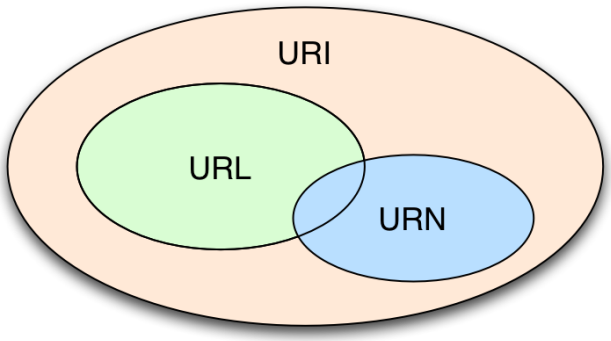


Figure 2.2: Venn-Diagram of URI, URL and URN.

URIs are one of the most important technologies for the Semantic Web. The central layers of the Semantic Web stack heavily rely on this concept.

Unicode

Unicode is a standard for the consistent encoding, representation and handling of text expressed (in most of the world's) writing systems. Developed in conjunction with the *Universal Character Set (UCS)*[2] standard, the latest version of Unicode (6.0) consists of a repertoire of more than 109,000 characters [45].

The development of Unicode is coordinated by the nonprofit organization Unicode Consortium. Their target is to replace existing character encoding schemes with Unicode. To use Unicode, the Unicode Consortium provides character encoding sets called *Unicode Transformation Format (UTF)*. UTF is a method for mapping Unicode-characters to bytes. Thus, it allows it to be implemented by computers [45].

UTF-8 (8-bit variable-width encoding) and UTF-16 (16-bit variable-width encoding) became the main method of encoding characters on most Unix-like operating systems or Windows. UTF-8 is the most common Unicode encoding method used within HTML documents [45].

XML

The *Extensible Markup Language (XML)* is a flexible text format for specifying semi or completely structured data. XML derived from SGML (Standard Generalized Markup Language [54]) and was originally designed to meet the challenges of large-scale electronic publishing [62].

XML is a tool for creating structured documents, but it doesn't impose semantic constraints on the meaning of these documents. The flexibility of XML didn't only influence hypertext Web technologies like HTML, but also defined an important fundament for the Standardized Semantic Web technologies [55, 62].

Nowadays, XML plays an important role for the exchange of data on the Web and in the Internet. Because XML documents sometimes need to be further characterized, the concept of *XML Schema (XMLS)* was introduced. XMLS is a language for restricting the structure of XML documents and also extends XML with datatypes. [61, 62]

[2]The Universal Character Set is a standard set of characters upon which many character encodings are based. The UCS contains nearly 100.000 abstract characters, each identified by an unambiguous name and an integer number called its code point [46].

2.3.2 Standardized Semantic Web technologies

The following technologies were introduced and already standardized by the W3C to extend the Web and enable developers to build Semantic Web applications.

RDF

The *Resource Description Framework (RDF)* is a datamodel for describing objects ("resources") and relations between them. It is a general-purpose language for representing all kinds of information on the Web. Since RDF is a model of metadata and only addresses many of the encoding issues that transportation and file storage require by reference, RDF relies on the support of XML [51].

RDF is an important concept for the practical part of this work and is described in more detail in section 2.4.

RDFS

RDF Schema (variously abbreviated as RDFS, RDF(S), RDF-S, or RDF/S) extends the basic vocabulary of RDF and has strong similarities to XML Shema. It is a W3C recommendation since 2004 [59].

RDF properties can be interpreted as attributes of resources. Further, they can represent relationships between resources. Unfortunately, RDF itself provides no mechanisms for describing such properties or relationships between properties and other resources. This is where RDF relies on RDFS. RDFS is used for, for example, creating hierarchies of classes or specifying own datatypes. RDFS elements are typically identified by a *rdfs* prefix and can also be described using RDF's concepts. [59].

SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is a query language and can be used to query any RDF-based document including, for example, statements involving RDFS and OWL.

SPARQL is an important part of the Semantic Web stack, since it allows Semantic Web applications to retrieve information from RDF documents and use them [37, 60].

OWL

If machines are expected to perform useful reasoning tasks on information, the language must go beyond the basic semantics of RDFS [61].

The *Web Ontology Language (OWL)* is an extension to RDF and RDFS. It formally describes the meaning of terminology used in Web documents. It is based on description logic and thus enables reasoning within the Semantic Web. OWL is designed to be used by applications, which need to process the content of information rather than such that just present it to humans [61].

In practice, the concrete syntax to store OWL ontologies and to exchange them among tools and applications is the RDF/XML syntax. This is the only syntax supported by all OWL tools. As a result, also OWL syntax can be described using the RDF language. Therefore, it is possible to use SPARQL within OWL [61].

While the RDF/XML format already ensures interoperability among OWL tools, other syntaxes are also supported. These include alternative RDF serializations such as Turtle or N-Triple. Further, there exists an extra OWL XML serialization syntax [61].

2.3.3 Unrealized Semantic Web technologies

The top layers of the Semantic Web stack contain technologies that are not yet standardized or contain just ideas that should be implemented in order to realize the Semantic Web. Until not all of these layers are standardized, the Semantic Web can not be achieved completely. Never the less, these technologies are already in development and partially already usable.

RIF/SWRL

The *Rule Interchange Format (RIF)* or the *Semantic Web Rule Language (SWRL)* are Semantic Web rule-languages, which support rules within the Semantic Web. They define relations that can't be directly described within the layers below. For example, a rule could state that: if *A is brother of B* and *B is a woman* we can infer that *B is sister of A* [5]. Even though RIF is already in a later stage of development, it is not finally decided which rule-language should be used.

Unifying logic

The target of this layer is to remove the differences between the logics of the layers below, creating an unified logic [26]. This should lead to a state allowing all layers above to have full access to the data from the layers below [5].

For example, developers can use different terms to describe a resource. While one developer may name a resource *fullName*, another one would only name it *name*. Even though the content of these resources are identical, machines are not able to "understand" that. The aim of the Unifying logic layer is to remove such inequalities.

Proof

This layer is intended to enable users (humans or machines) to request proof of certain data that is provided by a service. For example, a Web service would provide relevant Web pages as proof of the correctness of its information, if the service consumer requires it. Although services are still far from using the complete power of the Semantic Web, there are already existing services that are able to provide and exchange proofs (mostly implemented as a function of the specific Web application) [5].

Cryptography

Cryptography can be seen as the science of hiding information. Applications of cryptography include, for example, computer passwords or secure connections on the Web. Cryptography is important to ensure and verify that Semantic Web statements are provided by a trusted source. This can be achieved by appropriate digital signature of RDF statements [5].

Trust

Trust to derived statements will be supported by [5]:

(a) verifying that the premises come from trusted source and by

(b) relying on formal logic during deriving new information.

User interface and applications

The user interface is the final layer that enables humans to use Semantic Web applications such as Web sites or other applications. Further, this layer considers machines, which use the Semantic Web for different purposes (for example, processing information) [5].

2.4 The Resource Description Framework

The *Resource Description Framework (RDF)* is a framework developed by the World Wide Web Consortium to describe and represent resources on the Web. There are a variety of applications for RDF that increased the motivation for its development. An incomplete list of possible applications [51] is as follows:

- Web metadata: provide information about Web resources and the systems that use them.
- Machine processable information: allow data to be processed outside the particular environment in which it was created.
- Interworking among applications: combining data from several applications to create new information.
- Automated processing of Web information: allow data to be processed by software agents and increase efficiency of exploiting the information of the Web.
- Represent information in a minimal constraining and flexible way.

These applications inspired the development of RDF and to further improve it until it became a World Wide Web Consortium (W3C) recommendation. One reason of the success of RDF is due to its design. The design of RDF is intended to meet the following requirements [51]:

A simple data model

Create a simple data model usable for applications. The data model is independent of any specific serialization syntax [51].

Formal semantics and inference

RDF has a formal semantic, which provides a dependable basis for reasoning about the meaning of a RDF expression. In particular, it supports defined notions of entailment that can be used for defining rules of inference in RDF data [51].

Extensible URI-based vocabulary

The vocabulary is fully extensible, because it is based on URIs. It also allows optional fragment identifiers (URI references, or URIrefs). Another kind of value that appears in RDF data is a literal, which is described later in this section [51].

XML-based syntax

RDF has a recommended XML serialization form that can be used to encode the data model for exchange of information among applications [51].

Use XML Schema datatypes

Since RDF has a XML serialization, it can use values represented according to XMLS datatypes. This improves the exchange of information between RDF and other XML applications [51].

Anyone can make statements about any resource

To facilitate operation at Internet scale, RDF is an open-world framework, which allows anyone to make statements about any resource. Unfortunately, RDF does not prevent anyone from making assertions, which are nonsensical or inconsistent with other statements. This has to be considered by designers [51].

These requirements leaded to the definition of RDF [51] in terms of:

- an abstract syntax that reflects a simple graph-based data model and
- a formal semantic with a rigorously defined notion of entailment.

The concepts used to realize this definition are presented in the next section.

2.4.1 RDF Concepts

RDF uses a variety of concepts to ensure flexibility and extendability. The following presents a list of used concepts followed by a detailed description of each of them:

- Graph data model
- URI-based vocabulary
- Datatypes
- Literals
- Expression of simple facts
- Entailment
- XML serialization syntax

Graph data model

The underlying structure of any expression in RDF is a collection of triples [51]. Each triple consists of:

1. a subject that is represented as a node in the graph,
2. an object that is represented as a node in the graph and
3. a predicate (also called a property) that denotes a relationship and represents an arc in the graph that always points towards the object.

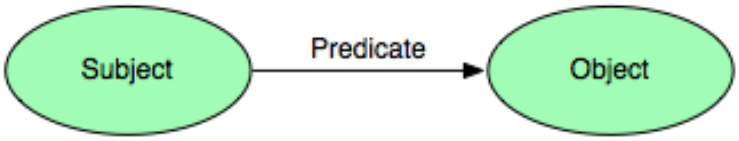


Figure 2.3: Graphical representation of a RDF triple.

The concept of such triple is illustrated in figure 2.3. A triple is also called a statement and a set of statements is called a RDF graph. Such graph can be

illustrated using nodes and directed-arcs, where each statement is represented as a node-arc-node link [51].

URI-based vocabulary and node identification

As mentioned before, the vocabulary of RDF is fully extensible, because RDF is based on URIs. Using URIs we can define two types of nodes in a RDF graph: an URI node or blank node [51]. Additionally, RDF allows, so called, literals.

An URI or a literal, used as a node, identifies what that node represents (for example, a person). An URI reference used as a predicate (arc) defines a relationship between the things represented by the nodes it connects. By default, it is not allowed to use a literal node as a subject [51].

To allow several statements referring the same unidentified resource, a blank node identifier can be used. This is a local identifier that can be distinguished from all URIs and literals of the graph. A blank node is a node that is neither a URI reference or a literal. In the RDF abstract syntax, a blank node is just an unique node that can be used in one or more RDF statements, but has no intrinsic name [51]. To use more blank nodes within the same graph an ID can be added.

Datatypes

Datatypes are used by RDF to represent values such as integers, floating point numbers or dates. They consist of a lexical space, a value space and a lexical-to-value mapping [51].

For example, the lexical-to-value mapping for the XML Schema datatype *xsd:boolean*, where each member of the value space (represented here as 'T' and 'F') has two lexical representations, is as follows [51]:

Value Space: {T, F}

Lexical Space: {"0", "1", "true", "false"}

Lexical-to-Value Mapping: {<"true", T>, <"1", T>, <"0", F>, <"false", F>}

RDF has no built-in concept of numbers, dates or other common values. Further, it does not provide any mechanisms for defining datatypes by itself. RDF rather depends on datatypes that are defined separately and can be identified using URI

references. The predefined XML Schema datatypes are widely used for this purpose [51].

Literals

Literals are used to identify values such as numbers and dates. Anything represented by a literal can also be represented by an URI, but often it's more convenient to use literals. A literal may be an object of a RDF statement, but it is not allowed to be used for a subject or a predicate [51].

RDF allows two kinds of literals, namely, plain or typed literals. A plain literal is a string combined with an optional language tag preferably used for plain text in a natural language. A typed literal is a string combined with a datatype URI and, therefore, it denotes the member of the identified datatype's value space. The value space is obtained by applying the lexical-to-value mapping to the literal string [51]. Continuing the *xsd:boolean* example, the typed literals that can be defined are shown in table 2.1 .

Table 2.1: Lexical-to-Value Mapping for the xsd:boolean example [51].

Typed Literal	Lexical-to-Value Mapping	Value
<xsd:boolean, "true">	<"true", T>	T
<xsd:boolean, "1">	<"1", T>	T
<xsd:boolean, "false">	<"false", F>	F
<xsd:boolean, "0">	<"0", F>	F

Table 2.1 illustrates four examples of typed literals that represent boolean values. The datatype *xsd:boolean*, defined by XMLS, allows the corresponding system to identify the literal value as boolean. Therefore, the system is able to use the Lexical-to-Value Mapping to map the string to a boolean value.

RDF Expression of Simple Facts

A simple fact that indicates a relationship between two things is represented as a RDF triple. In this triple the predicate names the relationship and the subject and object denote the related things. A row in a table of a relational database is a familiar representation of such a fact. The table contains one column for the

subject and the object. The name of the table can be seen as their relation and, therefore, as the predicate of the RDF triple [51].

Unfortunately, relational databases normally allow a table to have more than just two columns. Therefore, each row has to be decomposed to allow its data to be represented as RDF triples. A simple form of decomposition introduces a new blank node, corresponding to the row, and a new triple for each cell in the row. The subject of each triple is the newly created blank node, the predicate corresponds to the column name and the object corresponds to the value in the cell [51].

Table 2.2: Example of a relational table for storing student information.

ID	firstname	lastname	birthday
...	...	...	...
e0625287	Alexander	Aigner	28-04-1968
...	...	...	...

The simple table named *student_personalInf* represents a possible way to store personal information on students of a university. It contains an *ID* and the columns *firstname*, *lastname* and *birthday* for storing corresponding information of the student. A possible decomposition of table 2.2 is represented in figure 2.4.

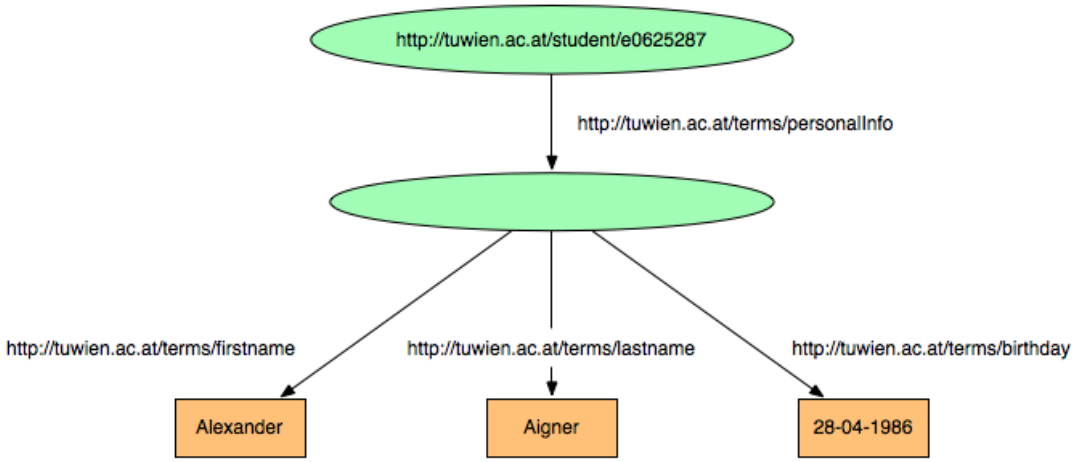


Figure 2.4: Decomposition of the relational table 2.2 to RDF.

Entailment

The ideas on meaning and inference in RDF are described by the formal concept of entailment (logical consequence) [51]. Entailment is a concept of the science of formal logic. For example, if a RDF expression *A* entails another RDF expression *B*, every possible arrangement of things in the world that makes A true also makes B true. In logics this would be described with the formula $A \models B$.

XML serialization syntax

A RDF graph contains nodes and directed arcs that connect pairs of nodes. As mentioned, this can be defined as triples containing a subject, predicate and an object [51].

In order to transform the graph to XML, all elements of the graph need to be interpretable in XML terms. To represent URI references, the RDF/XML syntax uses *XML QNames* as defined in *Namespaces*. Every QName has a namespace name and a short local name. Both together represent the URI reference [51, 62].

Code 2.1: Example of using QNames within XML.

```
1 <?xml version='1.0'?>
2 <student xmlns:x="http://tuwien.ac.at/student/">
3         <x:name>Alexander Aigner</x:name>
4 <student>
```

Code 2.1 illustrates an example of using QNames. The command *xmlns:x = "http://tuwien.ac.at/student/* in line two declares the prefix *x* to be associated with the namespace *http://tuwien.ac.at/student/*. Further on, this prefix can be used as abbreviation for this namespace. Subsequently, the tag *x:name* in line three is a valid QName, because it uses the *x* as namespace reference and *name* as local part. Therefore, <*x:name*> is an abbreviation of <*http://tuwien.ac.at/student/name*>. The tag <*student*> in line two also contains a valid QName, only consisting of a local part. QNames without namespace references generally use the default namespace.

As mentioned before, a graph can be seen as a collection of paths of the form *node-arc-node-arc-node-....-node*. In RDF/XML syntax, this is translated into

sequences of elements inside elements, where the node at the start of the sequence turns into the outermost and the node on the end in the innermost node [51].

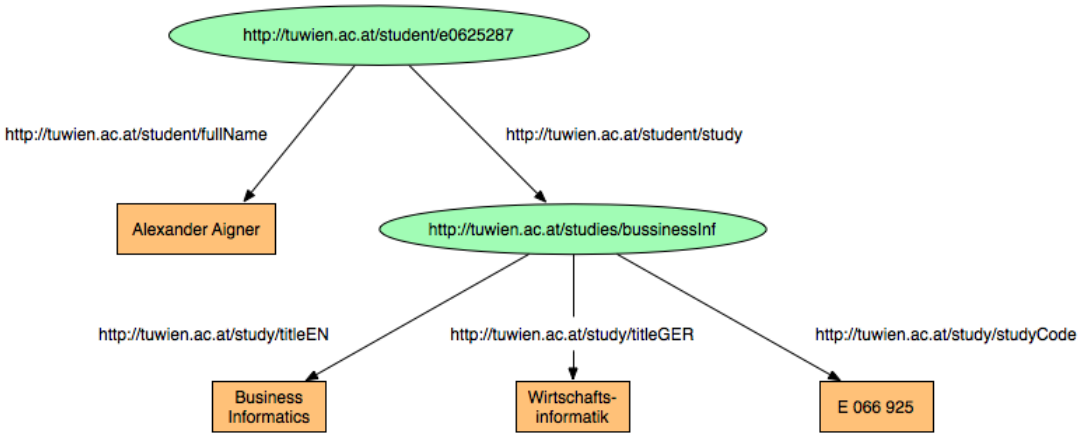


Figure 2.5: Example of an RDF graph.

Figure 2.5 illustrates a simple RDF graph for describing a student and what he/she is studying. The student has a name and is identified by an immatriculation number. Further, the study contains a study code as well as a german and english title. The RDF/XML code, shown in code 2.2, represents the serialized version of this RDF graph.

Code 2.2: Example of a RDF document.

```
1  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2                  xmlns:student="http://tuwien.ac.at/student/"
3                  xmlns:study="http://tuwien.ac.at/study/">
4
5  <rdf:Description rdf:about="student:e0625287">
6          <student:fullName>Alexander Aigner</student:fullName>
7          <student:study>
8                  <rdf:Description rdf:about="study:businessInf">
9                          <study:titleEN>Business Informatics</study:titleEN>
10                         <study:titleGER>Wirtschaftsinformatik</study:titleGER>
11                         <study:studyCode>E 066 925</study:studyCode>
12                 </rdf:Description>
13         </student:study>
14 </rdf:Description>
15
16 </rdf:RDF>
```

As mentioned prior, the node at the start of the sequence turns into the outermost and the node on the end in the innermost node. Since nesting can lead to very complex structures, the *rdf:resource* attribute enables humans to understand the context easier. For machines it does not matter if this attribute is used or not [51]. The altered code is shown in code 2.3.

Code 2.3: Example of using the *rdf:resource* attribute.

```
5  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6                   xmlns:student="http://tuwien.ac.at/student/"
7                   xmlns:study="http://tuwien.ac.at/study/">
8
9  <rdf:Description rdf:about="student:e0625287">
10         <student:fullName>Alexander Aigner</student:fullName>
11         <student:study rdf:resource="study:bussinessInf" />
12 </rdf:Description>
13
14 <rdf:Description rdf:about="study:businessInf">
15         <study:titleEN>Business Informatics</study:titleEN>
16         <study:titleGER>Wirtschaftsinformatik</study:titleGER>
17         <study:studyCode>E 066 925</study:studyCode>
18 </rdf:Description>
19
20 </rdf:RDF>
```

Further, the *rdf:resource* attribute is used to avoid multiple declaration of the same resource. If a few triples refer to the same object, this object does not have to be serialized multiple times (if it is an URI). Instead, it is declared only once and all other statements refer to the same object by using the *rdf:resource* attribute [51].

XML is not the only way of serializing RDF, but it describes the most common one. This is probably because the World Wide Web Consortium introduced the RDF/XML syntax and the RDF abstract syntax in the same time [51, 62]. Nowadays, there are various other serialization syntaxes like, for example, N-Triple [57], Turtle [63] or JSON [12].

2.5 PHP: Hypertext Preprocessor

PHP: Hypertext Preprocessor (PHP) is a general-purpose open source Web development language. PHP is the fourth most used programming language in the Web [13] and an important language to consider for developing Web projects. It is a server-side

scripting language, meaning that the code is interpreted on the server and only the result (normally HTML code) is send to the client. In contrast to Java Script, where the client has access to the complete code, PHP code is invisible to the client [40, 41, 42].

Since HTML is static, PHP can be embedded into HTML documents and, therefore, can be used for creating dynamic Web sites. A common use of PHP is querying data in a MySQL database and present them to the user using the HTML format (often used for for a news page). PHP provides all known concepts of other programming languages like, for example, loops, interfaces or classes [40, 41, 42].

During its development, PHP introduced a lot of new functions. For instance, PHP allows object-oriented programming paradigms (introduced in PHP release 5) or interoperability with other programming languages like, for example, Java or Perl. Even though PHP is also used for developing Desktop applications or command-line scripts, it's main usage is still in the field of Web development [40, 41, 42].

2.6 Application Programming Interface

An *Application Programming Interface (API)* is a set of rules and specifications to enable software programs access to resources or to communicate with services. It normally does not contain any user interface but rather serves as an interface between different software programs and facilitates their interaction, similar to the way the user interface facilitates interaction between humans and computers [48].

For this work we focus on APIs that enable developers to use RDF within their projects. A RDF API is a collection of methods or functions to simplify the work with RDF resources. By including a RDF API, developers can rather easily add semantic technologies to their projects.

CHAPTER 3

State of the art of RDF APIs

Since a few years the interest of using semantic technologies has steadily grown. New tools and frameworks have been developed and more developers started to use these technologies. Since the introduction of the Resource Description Framework (RDF) and its definition as a standard Web concept, a lot of different RDF toolkits were developed. Some of these tools support developers by using RDF within their projects. These tools are Application Programming Interfaces (APIs).

This chapter introduces the state of the art of APIs for processing RDF based data. In fact, the topic of this work is the development of an RDF API for PHP. Therefore, this chapter is even more important. First of all, we list and describe all available RDF APIs for PHP. Next, we point out some of their strengths and weaknesses. Since RDF APIs are available for nearly all available programming languages, the second part of this chapter introduces and describes a few popular APIs, which are based on other programming languages.

3.1 RDF APIs for PHP

Even though PHP is a popular programming language for Web development, PHP developers are provided by only four APIs, namely, *ARC* [30], *RAP* [33], *PEAR:RDF* [38] and *PHP XML Classes* [2]. Initially, this seems adequate, but, unfortunately, three of

these four APIs are outdated and not further developed. In the following these four APIs are described in more detail.

ARC (Appmosphere RDF Classes 2) is promoted as a lightweight, SPARQL-enabled RDF system. It facilitates the integration of RDF in PHP, basic SPARQL functions and allows configuration of a SPARQL endpoint. It supports a variety of RDF input and output formats like JSON, XML, COUNT, RDF/XML or Turtle/N3. ARC requires PHP (4.3 or higher) and MySQL (4.0.18 or higher). ARC is the only RDF API for PHP that is still up-to-date and fully compatible with the latest version of PHP. Therefore, most of the developers have to use this API, which results in a growing usage of ARC. ARC is currently in version v2 (07.01.2011) and recalled to *"ARC 2"* [30, 31].

RAP (RDF API for PHP) is a PHP API for parsing, querying, manipulating and serializing RDF models [33]. It supports a variety of ways to work with RDF and also allows querying using SPARQL. RAP allows importing and exporting RDF using formats like RDF/XML, N3, Nt and TriX. This API is probably still one of the most well known RDF APIs even though its development was already stopped a few years ago. The latest version of RAP is V0.9.6 (beta) and was committed on 29.02.2008 [33, 34].

PEAR:RDF is part of the PEAR framework, which also develops the PHPUnit[1] package. PEAR is a framework and distribution system for reusable PHP components. The RDF component of PEAR is an early and slightly modified version of the core RAP API. Therefore, PEAR:RDF provides the same functionalities as already presented for RAP. One of the main differences is that PEAR:RDF does not support as much different ways of storing RDF data. The latest version of PEAR:RDF is 0.2.0 (alpha) and was released on 05.10.2010 [38].

PHP XML Classes is a collection of PHP classes for processing XML-based documents in PHP [2]. One class of the collection, the RDF parser, is dedicated to handle RDF documents. This class provides an event-driven interface for parsing RDF statements. The document is passed to this parser, which then processes it.

[1] PHPUnit is a package developed within the PEAR framework, which allows to perform unit tests on PHP classes [3].

If an RDF statement is found, a handler (a PHP function), which is previously defined by the developer, is called and the RDF triple delivered. PHP XML Classes do not provide any other kind of manipulation or query methods for RDF. As a workaround, other XML classes of the collection can be used to manipulate the RDF/XML document, but this requires extra knowledge of the RDF/XML syntax. Further, there are no serializers and also the functionality of the parser is limited. The RDF parser is in version v1.1 (20.06.2002), but the whole PHP XML Classes package is still in beta phase [2].

Table 3.1: List of PHP APIs with latest release date, current version number and development status.

Name	Latest release date	Current Version	Development status
ARC	07.01.2011	2.0.0 (final)	in development
RAP	29.02.2008	0.9.6 (beta)	stopped
PEAR;RDF	05.10.2010	0.2.0 (alpha)	stopped
PHP XML Classes	20.06.2002	1.1.0 (final)	stopped

A comparison of release dates and development status is highlighted in table 3.1. As illustrated, the development of the PEAR:RDF, RAP and PHP XML Classes APIs has been stopped already. Thus, using these APIs often lead to incompatibilities and errors. The only way of using these APIs is if users fix all occurring errors by themselves. This may work for a period of time, but with future updates of PHP the necessary API changes increase. Therefore, ARC is left as the only future-proof option for PHP Web developers in the field of RDF manipulation and reasoning. Unfortunately, ARC is missing a usable documentation.

Although RAP is not developed any longer, it was the only API for PHP developers for a long time. Further, it is part of the PEAR framework and, therefore, we can assume that it is still used within some Web applications. RAP provides the most flexible way of working with RDF data for PHP by providing various ways of interaction. Thus, RAP, aside of ARC, will be included within the comparison and evaluation of the state of the art APIs in chapter 4.

We want to mention that this section exclusively presented RDF APIs for the programming language PHP. For PHP developers also other RDF tools are available like converters, browsers, editors or validators. These tools are Web applications on their own rather than interfaces that can be used by other developers.

Further we want to note that during the final stages of this work the development of the ARC API was officially cancelled [30].

3.2 RDF APIs for other programming languages

RDF APIs are implemented in various programming languages to support as much computer systems as possible. RDF APIs for other programming languages are often more advanced than for PHP, which makes this section quite interesting. It gives an introduction to well known and popular APIs for the common programming languages Java, .Net and C/C++.

Java represents an important programming language, which is already used by more then 6.5 million software developers. Besides supporting all common computer systems, Java allows software development on handhelds or mobile-phones [36]. Therefore, we decided to present two APIs for Java, namely, Jena [16] and Sesame [1].

Another programming language of growing interest is called .Net (dot Net). It is a platform for developing software introduced by Microsoft. Even though .Net is also available for unix-based systems, the full functionality can be only utilized on Windows-based systems [29]. The API we have chosen for representing .Net is called dotNetRDF [43].

The last and probably most popular programming language in our list is C/C++. It is a standardized and common used programming language. Lots of features provided by common operating systems are implemented using the C/C++ language and basically all operating systems allow interpretation of C/C++ code [49]. OpenLink Virtuoso [35] is used as API for this programming language.

These APIs present an incomplete list of API for their corresponding programming languages. Since Java, .Net and C/C++ are very popular programming languages, used for desktop and Web development, there are much more APIs than for PHP. Anyway, according to [53] they define some of the most popular and most used APIs for their pro-

gramming languages. For a list of all available APIs, the World Wide Web Consortium provides an up-to-date list at [53].

Jena is the first and probably most well known API for Java. It is a leading Semantic Web toolkit for Java programmers, first released in 2000 [16]. The API provides an environment for RDF, RDFS and OWL. Further, Jena supports SPARQL and includes a rule-based inference engine. It is open source and a result of research of the HP Labs Semantic Web Program (for more information see [19]). The Jena API allows reading and writing RDF in RDF/XML, N3 and N-Triples as well as in-memory or persistent storage. It provides a SPARQL query engine, but also methods for handling OWL documents. Jena is currently in version 2.8.8 released on 21.04.2011 [16, 27].

Sesame is another popular API based on Java. Sesame was developed during the EU research project OnToKnowledge (for more information see [25]). It supports the same functionalities as Jena with the extension that it can be deployed on top of a variety of storage systems (relational databases, in-memory, file-systems, keyword indexers, etc.). Sesame has been designed with flexibility in mind and offers a large scale of tools to developers to leverage the power of RDF and related standards. It enables full support of the SPARQL query language and offers transparent access to remote RDF repositories. Sesame supports all main stream RDF file formats, including RDF/XML, Turtle, N-Triples, TriG and TriX. Sesame is also the basis for other toolkits that are implemented as plug-ins for Sesame. The latest version of Sesame is 2.4.2 and was released on 14.07.2011 [1, 10].

dotNetRDF is an open source .Net library, using the latest versions of the .Net framework. It provides an API for working with RDF using the programming languages .Net or C# (C-Sharp). It provides support for different external stores (also, for example, Sesame). Further, there is a comprehensive SPARQL implementation that already includes support for many of the SPARQL 1.1 features. The API includes additional tools like, for example, a rdfEditor, a rdfConvert or a Store Manager. According to the developers, it is also usable for development on Windows Phone 7. dotNetRDF is still in beta state and currently in version 0.4.1 (released on 19.11.2010) [43].

OpenLink Virtuoso is the last API we want to present. It can be seen as a multi-platform API, because it is implemented using the programming language C/C++ and distributed as an executable. Therefore, it can be queried by all programming languages, which can access local processes, for example, Python, Java, Java Script or C#. Anyway, since the API is written in C/C++, we decided to use it for this language. According to the developers, it supports all common RDF manipulation methods, but also provides basic OWL manipulation capabilities. It does not only provide a SPARQL query engine, but also the possibility to set up an own SPARQL endpoint. Virtuoso is currently in version 6.2.0, which was released on 09.07.2010 [17, 35]. OpenLink Virtuoso is also the only commercial API we used. For testing this API we used a 15 days trial license.

Table 3.2: List of non-PHP APIs with latest release date, current version number and development status.

Name	Prog. Lang.	Latest release date	Current Version	Development status
Jena	Java	21.04.2011	2.8.8 (final)	in development
Sesame	Java	14.07.2011	2.4.2 (final)	in development
dotNetRDF	.Net	19.11.2010	0.4.1 (beta)	in development
Virtuoso	C/C++	09.07.2010	6.2.0 (final)	in development

A brief conclusion of the above presented APIs for programming languages, besides PHP, is presented in table 3.2.

Because ARC is the only PHP API, which is still in development, there is no point in evaluating only PHP APIs. ARC would outperform the other PHP APIs in all categories and, therefore, the evaluation would not have much significance. Thus, we decided that ARC and RAP, in conjunction with these four non-PHP APIs, will be subject to the evaluation and comparison in chapter 4.

CHAPTER 4

Comparison of RDF APIs

In this chapter we present an evaluation and comparison of popular APIs for PHP, Java, .Net and C/C++. The comparison of APIs independent of their programming languages, highlight strengths and weaknesses. APIs for programming languages, other than PHP, are generally more advanced. Therefore, with this comparison, we aim to discover valuable concepts that ease the use of an API, but were not used before in PHP APIs. Thus, the comparison of the APIs has great impact on the development of the PHP API in the practical part of this work.

In the first part of this chapter, we introduce the evaluation model, which we used for evaluation the APIs. Next, we present the evaluation catalogue as well as the evaluation method. In the last part of this chapter, the API evaluations are illustrated, the APIs are compared and the results are discussed.

4.1 The evaluation model

For a descriptive comparison it is necessary to choose a widely accepted evaluation model. Using a common model, we allow the reader to gain better understanding of the model as well as enable others to repeat our comparison.

There are numerous models available that are dedicated to quality evaluation. Anyway, for evaluating software products the well established *International Organization for Standardization (ISO)* [23] and the *International Electrotechnical Commission (IEC)*

[21] introduced the ISO/IEC 9126 standard in 1991 [22]. It was first revised in 2001 [22] and later on replaced by the ISO/IEC 25010 standard in March 2011 [24].

The objective of this standard is to provide a widely accepted way of measuring software quality during development as well as during usage. Since this model was designed to test software and provides a detailed definition of the quality characteristics, we think it is a good choice for evaluating APIs.

First, we want to give a brief introduction to the ISO/IEC 25010 standard, followed by a more detailed description of the *quality in use* model, which is the part of ISO/IEC 25010 and was used for the evaluation.

4.1.1 ISO/IEC 25010 quality model

The ISO/IEC 25010 standard was released in March 2011 by one of the departments of ISO called *SQuaRE (Software Product Quality Requirements and Evaluation)* [24]. It is a revision of ISO/IEC 9126 and inherited the well known views of software quality [22, 24], which are:

Internal quality refers to the static properties of the structure of the software product produced during the development process. Static properties are, for instance, the number of lines of code, modular complexity, number of faults found in a sequence or an activity diagram. It provides a white box[1] view of the product [24].

External quality refers to the software perspective on the computer system. It evaluates the software execution in a testing environment, on a specific hardware using a specific operating system. For example, it is possible to measure the number of faults detected during a test and so estimate the faults present in the whole software. It provides a black box[2] view of the product [24].

Quality in use refers to the perception of quality by the end users. This perception can be observed by executing the final software product for a specific context. Qual-

[1] White box test: the tester has access to the internal structures and algorithms including the code that implement these.

[2] Black box test: treats the software as a "black box", without any knowledge of internal implementation.

ity in use is effected by the external quality, which is depending on the internal quality. It provides a mixture of a black and grey box[3] view of the product [24].

The relationship between these software quality requirements and the whole system requirements is illustrated in figure 4.2.

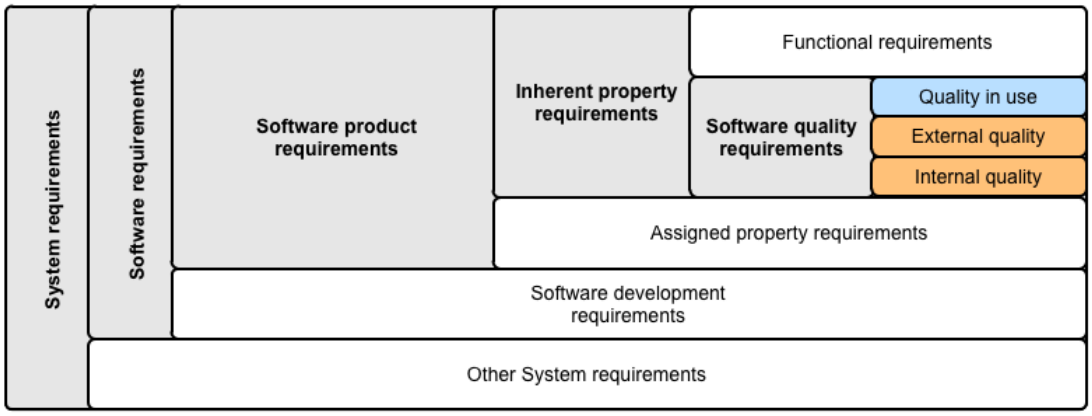


Figure 4.1: System requirement categorization (modified according to[22]).

These views can be used to measure different aspects of quality:

Internal and external quality is related to software quality during software development. To measure these qualities, a hierarchy of characteristics is used. This multi-level hierarchy has eight top level characteristics, which are further refined into sub-characteristics. The last abstraction level is constituted by the attributes. By assigning metrics to the attributes, they can be used as the measurable elements [8, 22, 24]. The hierarchy is presented in figure 4.2.

Quality in use can be seen as a traditional view of usability (during usage). Also the quality in use view uses quality characteristics as a set of attributes for measuring quality. It is also described as a hierarchy of characteristics, but in contrast to the internal/external quality hierarchy, the quality in use model contains only three top level characteristics.

[3]Grey Box test: involves having knowledge of internal data structures and algorithms, but testing at the user, or black-box level.

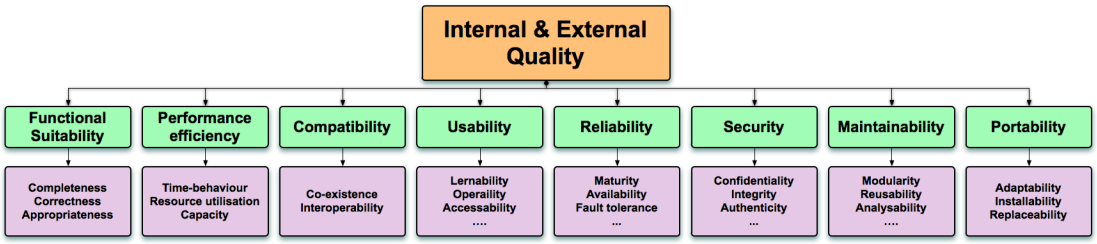


Figure 4.2: Product quality model (modified according to [24]).

Since the model defined as the quality in use view provides us a way of measuring the quality of an API during its usage, we used it for the evaluation of the APIs. Therefore, it is described in more detail in the following section.

4.1.2 The quality in use model

As mentioned above, the quality in use model can be seen as a traditional view of usability and represents the users perception of quality of a product. Usability is commonly defined as the combination of attributes of the user interface. Therefore, it describes if a product easy to use or not. A more general definition of usability is:

> *"Usability: the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use"* [22].

This definition was incorporated in the revision of ISO 9126 (2001) and renamed to quality in use. The quality in use hierarchy has three top level characteristics, namely usability, safety and flexibility [8, 24]. This hierarchy is illustrated in figure 4.3 and the characteristics are described in more detail below.

Usability describes the extent a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context [22]. It is the user's perspective of the quality when using a product. [8, 24]. The sub-characteristics of usability are effectiveness, efficiency and satisfaction [8, 24].

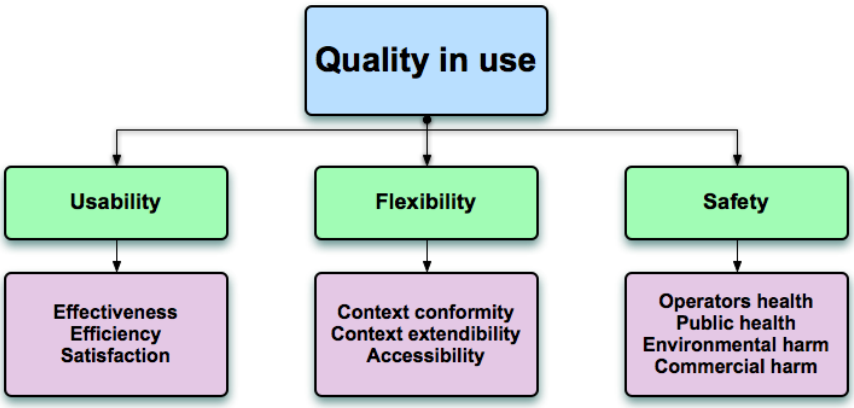


Figure 4.3: Quality in use model (modified according to [24]).

Effectiveness, which is characterized in terms of "accuracy and completeness", refers to doing the right things (do we achieve the wanted effect?). It constantly measures if the actual output meets the desired output [8, 24].

Efficiency is defined in terms of "expended resources" and checks if things are done in the right way. Scientifically, it is defined as the output to input ratio and focuses on achieving maximum output by using a minimum of resources [8, 24].

Satisfaction, previously interpreted in terms of "comfort and acceptability", has been given a broader interpretation in ISO/IEC 25010. Current approaches to satisfaction typically assess the users perception, so that if users perceive the product as effective and efficient, it is assumed to be satisfying [18]. In addition, aspects like fun or enjoyment can be part of the users experience, but were not considered before. Nowadays, we know that they contribute significantly to overall satisfaction with a product [8, 24].

In order to deal with the overall user experience, we need to further include experience, results and consequences (including safety). The satisfaction characteristic of the quality in use model suggested that satisfaction can be summarized within this four sub-characteristics [8, 24]:

Likability is represented in terms of "cognitive satisfaction" and describes the satisfaction with the ease of use and the achievement of pragmatic

goals (goals defined by the task).

Trust, specified in terms of "satisfaction with security", describes how the user is satisfied, because the product behaves as intended and with acceptable perceived consequences.

Pleasure is defined in terms of "emotional satisfaction" and describes the users satisfaction with their perceived achievement of hedonic goals (goals defined by the person), stimulation, identification and evocation as well as associated emotional responses.

Comfort is delineated in terms of "physical satisfaction". It describes the extend the user is satisfied with physical comfort.

Whereas efficiency and effectiveness can be measured rather simple, measuring satisfaction is rather complicated. For many developers satisfaction is seen as a personal response of the user that can hardly be quantified. Most usability tests obtain only qualitative feedback on satisfaction [8]. Qualitative feedback contains normally personal views on the quality. Sub-characteristics like trust or pleasure are often ignored.

Flexibility: As mentioned above, usability is related to a particular context. Therefore, considering the context is an important task. Often software products are used for more than just one context, but this can falsify the measured usability. Software that is usable in one context probably doesn't achieve same usability in another context with different users, tasks or environments [8, 24].

For example, it is possible to use a Web browser to browse the local file system, but it may not provide the same level of usability than for its intended context: surfing the Web. Unfortunately, developers sometimes trick users by demonstrate high levels of usability by a carefully selected, but unrepresentative, users, tasks and environments. To continue the previous example, this would mean to advertise the Web browser's functionality of browsing the file system with the usability value for browsing the Web.

The way a software product persists in its intended context or in other unintended contexts is described by the characteristic flexibility. Flexibility was firstly intro-

duced in the ISO/IEC 25010 and is described by three attributes, namely, context conformity, context extendibility and accessibility [8, 24].

Context conformity characterized the degree to which usability and safety meet requirements in all the intended contexts. This provides the basis for measuring the achieved level of usability in the intended contexts.

Context extendibility defines the degree of usability and safety in unintended contexts. It can be achieved by adapting a product for additional user groups, tasks and cultures. Context extendibility enables products to take account of circumstances, opportunities and individual preferences that are not anticipated in advance. If a product is not designed for context extendibility, it is not recommended to use the product in unintended contexts.

Accessibility specified the degree of usability of users with specified disabilities. Unfortunately, it is an objective, which is difficult to quantify.

Although it is hard to measure flexibility, it is an important task to specify the requirements. Specification of user groups, tasks and environments lead to additional design requirements for product features. Such requirements are mandatory to increase usability within all intended contexts. Unfortunately, not all contexts are testable. Context conformity in these contexts is assessed by expert judgment [8].

Also context extendibility is difficult to specify and to be measured in advance. Anyway, there are two ways to facilitate context extendibility [8]:

1. design a product, allowing it to be configured for specific needs (for example: language, culture, task steps), or
2. the product allows adaptation by the user to suit individual capabilities and requirements.

Products (especially software products) are frequently used for unanticipated purposes. This characteristic of a product often increases the usability significantly [8].

Accessibility can be tested by establishing objectives of usability for users with particular types of disabilities [8].

Safety is the last top characteristic of the quality in use model. It is defined as the degree of expected impact or harm to people, businesses, data, software, property or the environment that may happen during the intended contexts. While effectiveness and efficiency measure the positive outcome, safety is a way to measure the potential negative consequences, which might result from incomplete or incorrect output [8, 24]. Depending on the consumer and the product, negative outcome might vary from just personal inconvenience or loss of data up to financial loss. Safety has four attributes [8, 24]:

Commercial damage specifies the degree of expected impact of harm to commercial property, operations or reputation. This could include the administrative costs of correcting faulty output, inability to provide a service or loss of current or future sales. An example is the lack of sales due to poor web site design.

Operator health and safety defines the degree of expected impact of harm to the operator or user. For example, incorrect or unintended usage could result in higher risk of injury of the operator.

Public health and safety characterizes the degree of expected impact of harm to the public. For example, the public is introduced to extra risks due to incorrect usage of the product or development errors. For most software this attribute may not be that important, but it is crucial for, for example, software regulating a Nuclear power plant.

Environmental harm describes the degree of expected impact of harm to property or the environment. By extending the previous example of the Nuclear power plant, it is coherent that faulty software has also impact on the environment.

Specification and measurement of usability should always be considered in conjunction with associated safety risks. It is not easy to measure product safety, but it is usually possible to list the potential consequences of product failures or human errors based on previous experience with similar products or systems [8].

4.2 The evaluation catalogue

Although the quality in use model provides a sophisticated way of measuring quality of an API, we think it is still essential to specify a more API-related interpretation of the attributes and characteristics. The quality in use model is defined as a general model for evaluating all kinds of software rather than for evaluating APIs. Therefore, by defining an API-related interpretation we are able to perform a more API related evaluation. Further, we enable others to repeat the evaluation and allow them to gain a better understanding of the evaluation results. The interpretations of the quality attributes we used for the evaluation are defined in more detail below.

Usability

Effectiveness is measured by checking if the API works as intended. It can be interpreted as the questions: Does the API work as we expect? Are we able to achieve the expected effects/results?

Efficiency evaluates the input-output ratio in terms of invested resources in relation to output. Such resources are, for example, time or lines of code. We try to answer the question: How much resources we have to invest in order to achieve our goal?

Satisfaction

Likability is measured in terms of pragmatic goals. It can be interpreted as the questions: Do we like to use the API? Are we able to achieve all goals of the task?

Trust takes the personal feeling of security and support into account. It can be interpreted as the questions: Does the API work as it is intended? Do we accept the consequences of using the API?

Pleasure considers hedonic goals and can be interpreted as questions like: Are we satisfied with the API? Did we enjoy working with the API?

Comfort is originally defined in terms of physical satisfaction. We extended this category to be interpreted as the questions: Is there a usable documentation? Is the API easier to use than others? Are there usable code samples? Do we have less stress because of these benefits?

Flexibility

Context conformity is measured according to following simple context of use, which we defined as:

"The API should allow adding, updating, deleting and searching RDF resources. It should not just be able to present the result on screen but also store it in a file so it can be accessed by other APIs/software. It should also be possible to perform simple SPARQL queries."

To test this context we decided to create a simple RDF model. The RDF model contains *student* and *study* resources. The model represents students of an university and the study they are inscribed in. The student has a name and is identified by his immatriculation number. The study has a german and english title and is identified by the study code. This graph uses references as well as literals. Further, the study and the student resources are part of two different namespaces.

The target was to implement this model by using each API. Further, it is required that we can export the model as well as import it in order to continue working with it. This includes the search of statements as well as performing modifications (updating and deleting statements) within the model. At the end, we performed SPARQL select-queries on our model.

Context extendibility depends on the other functions of the API. It will be measured in terms of customizability of the API. A simple extended context is to use an API as replacement of a SQL database.

Accessibility is not measured since everybody, who uses a computer and has knowledge of a programming language, is able to use an API. The accessibility of the API depends on the accessibility of the device (computer) it is used on. Therefore, we would have to rank all APIs the equally. Since this would only distort the mean and standard deviation we decided to exclude this attribute for the evaluation of the APIs.

Safety

Commercial damage is measured in terms of support of the API. If a company is using an API that is regularly updated, there is low risk of commercial

damage. In contrast, if the development of an API was already stopped, the company has to fix errors by themselves, leading to a higher risk of commercial damage. This characteristic tries to answer the following questions: Are we able to rely on the API? Are there regular updates/bug fixes?

The characteristics "Operator health and safety", "Public health and safety" and "Environmental harm" depend on the kind of software the API is used in. Since the API is used by other software, these characteristics are more interesting for the main application. It is possible that an API failure could lead to an error in the main application but it is hard to measure since its not dependent on the API. Therefore we decided to quantify these characteristics equally for all APIs. Therefore, we decided to exclude these characteristics for the calculation of the characteristic safety, as they would just distort the result.

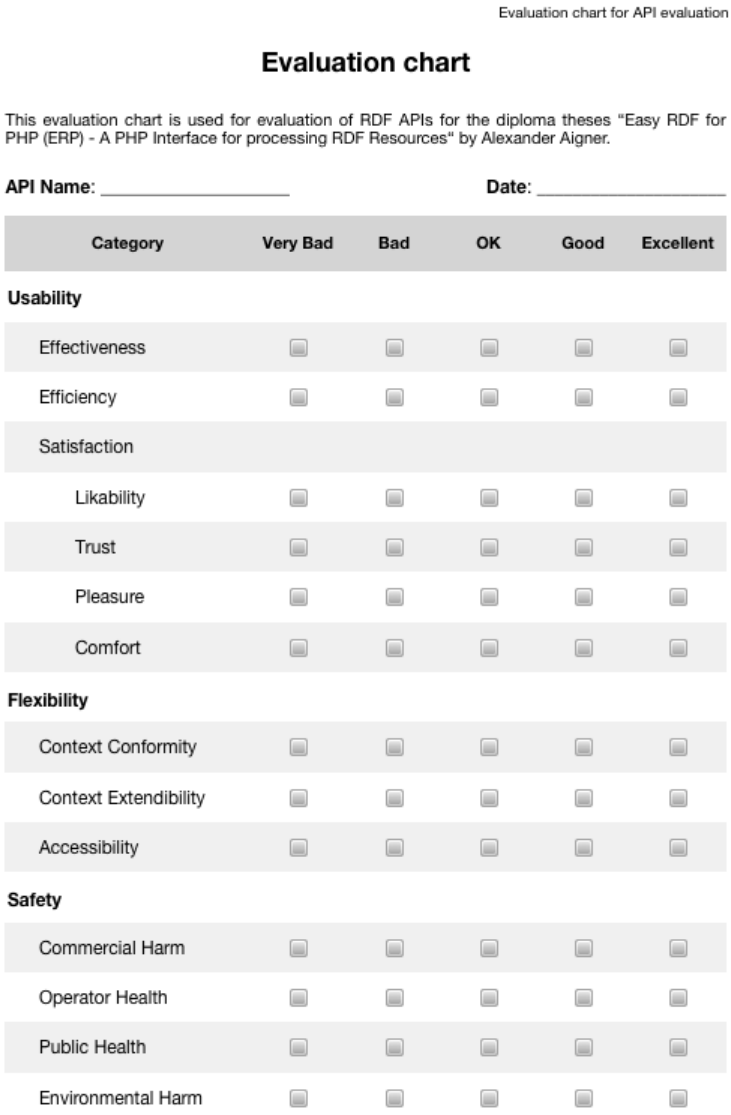
4.3 The evaluation method

Based on the quality in use model and the evaluation catalogue we already know what to measure and how the single characteristics are interpreted. The last point to define, before we can start the evaluation, is a way to measure the characteristics and to apply a rating to them.

For measuring the single characteristics, we decided to use a simple *Likert scale* [28]. Therefore, we defined each attribute as a *Likert item* and used a typical five-level format with the following categories.

1. Very bad
2. Bad
3. OK
4. Good
5. Excellent

By utilizing these items, we designed an evaluation chart, which is presented in figure 4.4. Using this chart we can select one of the predefined five categories and



Evaluation chart for API evaluation

Evaluation chart

This evaluation chart is used for evaluation of RDF APIs for the diploma theses "Easy RDF for PHP (ERP) - A PHP Interface for processing RDF Resources" by Alexander Aigner.

API Name: _____ **Date:** _____

Category	Very Bad	Bad	OK	Good	Excellent
Usability					
Effectiveness	☐	☐	☐	☐	☐
Efficiency	☐	☐	☐	☐	☐
Satisfaction					
Likability	☐	☐	☐	☐	☐
Trust	☐	☐	☐	☐	☐
Pleasure	☐	☐	☐	☐	☐
Comfort	☐	☐	☐	☐	☐
Flexibility					
Context Conformity	☐	☐	☐	☐	☐
Context Extendibility	☐	☐	☐	☐	☐
Accessibility	☐	☐	☐	☐	☐
Safety					
Commercial Harm	☐	☐	☐	☐	☐
Operator Health	☐	☐	☐	☐	☐
Public Health	☐	☐	☐	☐	☐
Environmental Harm	☐	☐	☐	☐	☐

Figure 4.4: Evaluation chart based on ISO/IEC 25010 quality in use model.

assign our rating to a specific quality attribute. The rating mirrors the testers personal perception of quality of a single characteristic.

To process the collected data of the evaluation and allow to compare the APIs we further assigned numeric values to each category. The assignment of the numerical

Table 4.1: Assigned numerical values to measured characteristics.

Category	Numerical value
Very Bad	20
Bad	40
OK	60
Good	80
Excellent	100

values to each characteristic is illustrated in table 4.1.

The values of the top level characteristics are defined as the arithmetic mean from all their sub-characteristics. Mathematically we defined the value of the top characteristic as follows:

> *If the sample space $A = \{a_1, \ldots, a_n\}$ describes the measured values for all n sub characteristics, the top characteristic is defined as $\bar{a} = \frac{1}{n} \sum_{i=1}^{n} a_i$ with $a_i \in A$. $\bar{a}$ then defines the arithmetic mean of all sub-characteristics and, therefore, the value of the top characteristic.*

Using this method of calculation, we are able to calculate a single value for each API, which defines its quality in use. Unfortunately, this value is not very expressive, because it is calculated from relatively independent values. Therefore, we decided to compare the APIs in terms of the three sub-characteristics usability, flexibility and safety instead of one quality in use value.

4.4 Evaluation results and comparison

After defining the evaluation catalogue as well as the method of measuring each quality characteristic, we are finally able to perform a descriptive evaluation and comparison of the APIs listed in chapter 3.

Since the evaluation covers different programming languages it is not easy to create an objective comparison. To define a foundation for maximizing the objectivity, we prepared a standard runtime environment for each programming language. If the environment had to be modified to enable execution of the API, deductions in the efficiency

attribute were accordingly given. Further, we did not deduct points for the implementation as long as it would be naturally to the programming language (no strange constructs, common function names, similar to the programming language). These specifications aim to increase the objectivity for measuring usability and to reduce subjectivity as much as possible.

As mentioned prior, the evaluation was performed by implementing the context by using each API. As expected, the level of usability, flexibility and safety are generally high due to the choice of popular, powerful and well established APIs. Another reason for the high levels is the choice of the relatively simple intended context. Unfortunately, we were limited to only one person, which performed all the API evaluations. Anyway, the results of the evaluation of the APIs are presented in table 4.2.

Table 4.2: Results of the evaluation of state of the art APIs.

Category	RAP	ARC	dotNetRDF	Sesame	Jena	Virtuoso
Usability	**50**	**75**	**70**	**77**	**98**	**37**
Effectiveness	80	100	80	80	100	20
Efficiency	20	60	60	80	100	40
Satisfaction	50	65	70	70	95	50
Likability	60	60	60	80	100	40
Trust	20	80	60	100	100	100
Pleasure	60	60	80	60	100	20
Comfort	60	60	80	40	80	40
Flexibility	**90**	**90**	**90**	**90**	**90**	**100**
Context Conformity	100	100	100	100	100	100
Context Extendibility	80	80	80	80	80	100
Accessibility	-	-	-	-	-	-
Safety	**40**	**80**	**80**	**100**	**80**	**100**
Commercial Harm	40	80	80	100	80	100
Operator Health	-	-	-	-	-	-
Public Health	-	-	-	-	-	-
Environmental Harm	-	-	-	-	-	-

In addition to the evaluation results, table 4.3 outlines the arithmetic mean and the standard deviation of all evaluated APIs.

Table 4.3: Mean and standard deviation of the evaluation results.

Category	Mean	Standard deviation
Usability	67.83	21.53
Flexibility	90.00	6.32
Safety	80.00	21.91

Even though the mean and standard deviation are calculated by a small sample size of only six APIs, it gives an impression of the overall performance. The following paragraphs compare the results of the top characteristics usability, flexibility and safety in more detail.

Usability is measured with an arithmetic mean of 67.83 points, which is relatively low in compare to flexibility and safety. The lowered usability mean is due to two APIs, namely, RAP and OpenLink Virtuoso, which got relatively low value for usability.

RAP is actually a very powerful API, but due to the fact that it was necessary to fix API related errors during the implementation it received lower points in efficiency. As it is not updated any longer, further deductions were given in the sub-characteristic trust. This leaded to a low usability value of 50 points.

OpenLink Virtuoso is also a very powerful API. Unfortunately, due to bad documentation of basic functionality and relatively complex implementation it received an usability value of only 37 points. This is the lowest measured value for usability, therefore, decreasing the arithmetic mean and increasing the standard deviation. Using the OpenLink Virtuoso API we had to spend most time for implement the intended context. Further, due to the time restriction of the trial version we were not able to perform all tests. Also by using the API's command line interaction we didn't achieve better results.

ARC (75 points), dotNetRDF (70 points) and Sesame (77 points) provide similar functionality, but all these APIs had their pros and contras. Therefore, these APIs are ranked between OK (60 points) and Good (80 points).

The best API in the characteristic usability is Jena. Jena has a well documented user guide and the implementation is very native to the language. Implementing the defined context in Jena was faster, but also more satisfying than with the other APIs. Therefore, Jena achieved the highest usability value of 98 points, which is very near to the maximum value of 100 points (Excellent). Jena allows a very efficient implementation using a object oriented approach. This level of usability is incomparable to the level of PHP APIs. The only PHP API that can compete with Jena is RAP (if it would not be necessary to fix API errors). Therefore, Jena and RAP present great opportunities to increase the efficiency for PHP APIs. We tried to consider these opportunities during the development of our own PHP API.

Flexibility has the highest mean (90 points) and is ranked directly between Good (80 points) and Excellent (100 points). This high rating is due to two reasons.

1. The first reason is that the context is chosen relatively simple. Since we compared popular and widely used APIs, all of them provide the necessary functions to implement the intended context. Therefore, all APIs achieved 100 points (Excellent) in the characteristic context conformity.

2. The second reason is also due to the choice of the APIs. All chosen APIs offer a lot of extra functions (for example, SPARQL, RDFS, OWL or various export formats) that can be used to extend the defined context.

   Even though OpenLink Virtuoso received a relatively low value for usability, it is by far the most flexible API of all evaluated ones. Since it is distributed as an executable, it can be used by all programming languages that can access local processes. Therefore, it can not only be used by C/C++ programs but also for various other programming languages. This leaded to the maximum value in the characteristic flexibility.

   Using the other APIs it is not that simple to make them accessible as a system process. They are intended to be used within their corresponding programming language rather than within other programming languages. Anyway, they allow modification of the source code and often provide a list of constants that can be changed according to the developers needs. Since they

all allow similar capabilities, they are ranked equally with 80 points in the characteristic context extendibility.

Safety is another characteristic, where most APIs achieved high ratings. In our evaluation, safety is defined only by the category commercial harm. To measure commercial harm, we tried to evaluate how long it would take to update an API to run on an updated system (for example a new Java version). Since all APIs except RAP are regularly updated, they all ranked between Good (80 points) and Excellent (100 points).

Because the development of RAP has been stopped, we expect that there are no further updates any longer. Thus, all errors that appear in future system updates have to be fixed by the company. RAP uses a lot of PHP functions, which means that it can still provide some security for future updates. Unfortunately, some of these functions are already deprecated and it is unclear how long they will still be part of the PHP framework. Therefore, RAP is ranked with only 40 points (Bad), which decreased the mean and increased the standard deviation of this characteristic.

In overall, we can state that the best API of our comparison is certainly Jena. While flexibility and safety are more or less equal throughout all APIs, it provided by far the best level of usability.

For the development of the ERP API the characteristic usability presents the most importance. Efficiency and effectiveness are of great importance, but also satisfaction presents a strong opportunity where our API can score. Even though we chose popular APIs, the rankings of the characteristic usability were rather disappointing. Only Jena could convince us. It scored the highest in the characteristic satisfaction as well as efficiency and effectiveness. Therefore, Jena became a great source of inspiration for the development of the ERP API. By introducing some of Jena's concepts to the ERP API, it might be possible to achieve similar values in effectiveness, efficiency and satisfaction.

Flexibility is achieved by gradually improving and extending the API during time. The target of the development of the ERP API was to provide the most important features. Therefore, the ERP API is not be able to provide the same flexibility as APIs that are developed for years. Anyway, during maturing of the API and by extending its

features, the flexibility of the API will increase. Thus, this characteristic is not of much importance at this point of the development.

Safety is another characteristic, in which we will not be able to achieve high rankings for the beginning. Even though we plan to continue the development of the ERP API, we still don't have the same support as the other APIs. Most of the APIs are developed by developer teams and supported by their users. Since our API is still unknown, we still can't achieve the same level of safety. Even though safety does not solely depend on the number of developers, it reduces the risk of using the API if there is more than one dedicated developer. Further, if users and developers work together, it is more likely to provide a good foundation for future updates. Anyway, it is possible to achieve higher rankings of safety if the ERP API gets more popular and is supported by its users and other developers.

Now that we have evaluated and compared the APIs for their usage, we still want to give a brief comparison of the features of these APIs. Following features were compared:

- OOP (Object-oriented programming) Support: This category defines if an API allows users to use object-oriented programming paradigms.
- RDFS/OWL Support: Since RDFS and OWL concepts can be created by using basic RDF, all APIs are able to create statements using these concepts. Anyway, if an API does not provide extra functions for creating these constructs (classes, properties, instances), it is categorized as unsupported.
- SPARQL Support: This category specifies if an API supports SPARQL.
- Add, Update, Delete, Search Statements: These categories check if the functionalities are supported.
- Modifiable: It checks if the API can be modified by the user to be adapted for certain needs.
- Import/Export Formats: Specifies how many formats are supported by the API.
- Documentation: Describes which kind of documentation is available for the API.

The comparison of these features are presented in table 4.4.

Table 4.4: Comparison of API functionalities.

Category	RAP	ARC	dotNetRDF	Sesame	Jena	Virtuoso
OOP Support	Yes	No	Yes	Yes	Yes	No
RDFS Support	Partial	No	No	No	Yes	No
OWL Support	Partial	No	No	No	Yes	No
SPARQL Support	Yes	Yes	Yes	Yes	Yes	Yes
Add Statements	Yes	Yes	Yes	Yes	Yes	Yes
Update Statements	Yes	Yes	Yes	Yes	Yes	Yes
Delete Statements	Yes	Yes	Yes	Yes	Yes	Yes
Search Statements	Yes	Yes	Yes	Yes	Yes	Yes
Modifiable	Yes	Yes	Yes	Yes	Yes	No
Import Formats	6	14	8	7	5	4
Export Formats	5	11	7	7	5	4
Documentation	Web, PHPDoc	Web	Web	Web, JavaDoc	Web, JavaDoc	Web

As illustrated in table 4.4, all APIs support the basic functionalities of adding, updating, deleting and searching RDF statements. All APIs support a variety of different import and export formats. The following formats are supported by all APIs:

- N-Triples,
- RDF/JSON,
- RDF/XML and
- Turtle.

Further, some APIs support other in-official formats such as *CSV (Comma-Separated Values)* [39] or *RSS (Really Simple Syndication)* [9].

As a positive, all APIs support the SPARQL query language. All APIs provide a documentation on the Web. some even provide a more comprehensive documentation of the API by using documentation generators such as PHPDoc or JavaDoc.

Unfortunately, most APIs don't provide extra functions for processing RDFS or OWL data. Therefore, users depend on the standard RDF functions. This sometimes requires workarounds and produces problems with parsers and serializers.

Also in the comparison of functionality, we considered Jena as the clear winner. It provides everything a user needs while working with RDF and more. Even though it supports less import/export formats than most other APIs, it provides all important ones. Considering the high level of usability and the great support for all important functionalities, we think that Jena is the best RDF API that is available at the moment. Therefore, we aim to enable PHP users to have an API with the same level of usability as Jena.

CHAPTER 5

Implementation of the ERP prototype

This chapter gives a detailed introduction to the prototype of the *Easy RDF for PHP API (ERP)*. The ERP API represents the practical part of this work and considers all the gathered information of the prior chapters.

First of all, we introduce our motivations that leaded to the development of the ERP API. Then we list and explain the API requirements, followed by a description of the major influences for the development of the ERP API. Next, we give an overview of the API's architecture and describe its functionality and usage.

Further, this chapter serves as a documentation of the ERP API and provides developers with all information necessary to include the ERP API in their projects.

5.1 Motivations for developing the ERP API

The idea for creating a RDF API was born during a seminar work on the technical university of Vienna in 2010. The task was to extend a PHP-based project management system to store RDF-based information and enable reasoning of the data using SPARQL.

Our first approach to this problem was to use the RAP API. Unfortunately, the RAP API was already outdated and various errors appeared during its usage. On beginning we tried to fix them and repair the API. After some time we reached a point, where we

created so many helper functions that the API was practically useless. Anyway, at this point we realized that there are no usable options for RDF APIs in the PHP language.

We also tried the ARC API, but, since the ARC is badly documented and we already had our own functions for processing RDF resources, we didn't use it for RDF. Anyway, we decided to use ARC to set up a SPARQL Endpoint, which we included within the application.

During this project the idea to develop a new, easy to use and powerful API evolved. Considering that PHP is one of the most used programming languages (ranked 4th after C, Java and C++) for Web development, it is necessary to provide a usable interface and thus support the vision of the Semantic Web.

Besides that, we want to introduce new concepts to PHP APIs to inspire the future development of RDF APIs for PHP. The ERP API shows that even complex manipulation of RDF documents can be achieved by using a simple and easy to use interface. This work and the API should inspire developers of APIs to simplify their APIs, so that also inexperienced users can include RDF within their projects.

Till now, the inclusion of sematic technologies within Web applications seemed like a lot of extra effort without any real benefit. Even though we can't increase the benefit for developers, the ERP API aims to minimize extra work by increasing efficiency. Only when the usage of semantic technologies increases, developers will be able to gain the benefits they expect.

5.2 Objectives and requirements for developing the ERP API

To achieve an API that is easy to use as well as powerful, we created a list of objectives. These objectives aimed to create an API that can outperform its concurrence and inspire other API developers to rethink their, often too complex, interfaces. We gathered this list of objectives during the evaluation and comparison of the APIs. Further, since we aim for high usability, we also considered the quality in use model for defining some objectives such as effectiveness and efficiency. Our main objectives are presented below:

- **Effectiveness:** Create an API that achieves the desired effect for the user.
- **Efficiency:** Achieve high efficiency by allowing easy inclusion and handling. Also use common names for functions that are also present in other APIs, so that a switch to ERP can be processed relative easily.
- **Simple and complex:** Since we want to provide an easy to use and powerful API, we want to provide simple usage for inexperienced users as well as enable more complex usage for experienced users.
- **Unit tests and code coverage:** Perform tests on all classes and achieve a high code coverage (percentage of tested code). This allows users to perform adaptions to the API without using the functionality, since all changes can be tested immediately.
- **Fast:** Design the API in the way that it doesn't reduce the speed of the main application. The best way to achieve a usable performance is to heavily rely on available PHP functions. They are normally implemented more efficient than own functions.
- **Formats:** Allow a variety of common import and export formats. The comparison of functionalities in chapter 4 pointed out that RDF/XML, RDF/JSON, Turtle and N-Triple are most important.
- **Natural implementation:** Design the API in the way that the used paradigms are natural to PHP. Every programming language has its own syntax. Therefore, the API needs to be naturally implementable for every PHP user.
- **Fast and easy inclusion:** Allow developers to include the API in a time-saving way by minimizing the required lines of code.

Using these objectives and the results of the comparison of chapter 4, we were able to define a list of requirements for the development of the ERP API.

Most of the requirements are oriented on the best practice methods of programming (for instance: loose coupling, unit testing, interfaces, ...)[1]. Other requirements are

[1]For more info's on best practice for programming see [47]

simply due to the required functionalities or result of the findings of the comparison in chapter 4. And the last part of the requirements are simply due to our experiences with previous projects. In the following, the architectural, functional and non-functional requirements are listed.

Architectural requirements

- The ERP API is implemented by using the PHP programming language.
- RDF manipulation without depending on other technologies like, for example, MySQL.
- Provide a modular architecture to allow exchangeability of single modules.
- Coupling between classes should be minimal.
- Define often used classes by interfaces.
- Reuse code by using inheritance.

Functional requirements

- Provide static functions for performing various checks on objects and parameters (for example: checking instances, value ranges or regular expressions).
- Creation of a RDF graph/model.
- Enable retrieval of model size (number of statements in the model).
- Creation and manipulation of nodes objects (Resource node, Literal node, Blank node).
- Creation of nodes by using static API functions to increase efficiency.
- Creation of nodes by the model. If the model creates a node it is done by a name parameter and by using the predefined base namespace.
- Creation of resource nodes by allowing either full URIs or a combination of namespace and name parameters.
- Creation and manipulation of plain and named literals.
- Predefine important constants (standard URIs, namespaces, datatypes, ...).
- Creation and manipulation of statements containing three nodes.

- Enable the user to create all kinds of RDF statements including RDFS (for example: Concepts, RDFS properties) and OWL.
- Statement-centric way of working with the API. To allow the creation and manipulation of statements within the model (objects, interfaces, classes).
- Storing all statements as a RDF model.
- Add statements to an RDF model.
- Update statements within the RDF model.
- Delete statements within the RDF model.
- Searching statements within the RDF model.
- Resource-centric way of working with the API. Allow the creation of resources as an object with properties as well as the manipulation of these objects. Therefore, enable the user to use modern programming paradigms.
- Storing all resources and their properties as a RDF model.
- Add resources to a RDF model. Resources without properties are not allowed.
- Update resources within the RDF model.
- Delete resources within the RDF model.
- Searching resources within the RDF model.
- Retrieve a list of all statements/resources.
- Transform a list of statements to a list of resources.
- Transform a list of resources to a list of statements.
- Provide the following serializers and parsers: N-Triple, Turtle, RDF/JSON, RDF/XML.
- Enable serializers to return a string representation of the model or save it to a file.
- Provide access to parsers and serializers within the model class as well as by static getter functions of the API.
- Retrieve a string representation of the model by using different formats (Nt, Turtle, JSON, XML).

- Format a list of statements/resources to a string in different formats (Nt, Turtle, JSON, XML).
- Enable simple inclusion of the API by providing an Autoloader class that loads all necessary classes for using the API.

Non-functional requirements

- The source code should be formatted according to the PHP guidelines.
- Perform parameter input checks for every function to prevent errors and wrong usage.
- Provide comments for functions and global variables.
- Use PHPDoc comments and prefixes to enable the use of PHPDoc.
- Use static code analysis tools to scan the code for unwanted antipatterns.
- Provide maximum possible code coverage.
- After each development iteration, it is important to ensure that the existing tests pass.
- Make the source code easy to scan by using commonly known patterns and formats.
- Create compact but expressive source code.
- Comment only what the source code can't say.
- Make the source code readable by using spacing, both horizontal and vertical.
- Make the source code self-documenting by choosing descriptive (but relatively short) names for objects, types, functions, etc.
- Avoid using modifiable global variables.
- Each variable should have the smallest possible scope. For example, a local object can be declared right before its first usage.
- Make functions short and focused on a single task.
- Functions should have few parameters (four is a good upper bound).

- Try to get some developers to support the API by sharing it on a social coding platform.
- Minimize necessary configurations.

These are the requirements that we defined for the development of the ERP API. Further, the functional requirements can be seen as a list of all the features of the ERP API. A few examples of the usage of the API and how these functional requirements are implemented are given later this chapter in section 5.5.

5.3 Influences from other APIs for developing the ERP API

The comparison of the APIs in chapter 4 visualized that there are ways of implementing an API to achieve high usability without resigning powerful functionality. The best example for this is the Jena API for Java. Jena nearly achieved the top ranking in all three characteristics by providing a simple, but powerful way of implementation.

Therefore, Jena became one of the most influencing APIs for the development of the ERP API. We tried to utilize some of Jena's concepts (for example, the resource-centric approach of the ERP API is similar to Jena's implementation approach) and introduce them to PHP.

Unfortunately, it was not as easy as expected. Since concepts from other APIs are often possible only due to the corresponding programming language, it is sometimes hard to introduce them to another programming language. Even though PHP does not support some programming paradigms in the same way as Java, we managed to transfer a few of Jena's concepts into our API.

Other influences were the ARC and RAP APIs. ARC is a powerful API, allowing the user to perform complex tasks like, for example, creating a SPARQL store. Since we also want to enable our users to perform such tasks, we included some of ARCs functionalities. ARC mostly influenced us for the creation of the parsers and serializers.

RAP allows inexperienced users to use RDF by providing a simple way of working with statements. RAP provides various ways of creating RDF models and thus it is probably the most flexible API from all APIs that we compared. RAP was a great

inspiration for the statements-centric approach of the ERP API and, therefore, it also had some influence on the ERP API.

5.4 Architecture of the ERP API

The ERP API is a powerful API that allows creation, storage and manipulation of RDF based documents. This includes RDF as well as concepts from RDFS and OWL. Since RDFS and OWL concepts can be seen as special URIs, basically all concepts are supported. Anyway, as discussed later in this chapter, to use these concepts the user still has to use some workarounds.

The prototype of the API provides a modular, easy understandable and extendable platform for developers. It can be easily included within existing or new Web projects and allows two ways of interaction. For inexperienced users there is a simple statement-centric approach and for experienced users we provide a resource-centric approach.

The statement-centric approach allows users to create statements and add them to the model. On the other hand, the resource-centric approach enables the user to create and handle the data using more complex object oriented structures. Both approaches are explained in more detail later this chapter.

The latest version of the API, published on September 1st, 2011, consists of 30 PHP files (50 including PHPUnit test files). These files contain 1.937 lines of code (3.511 with tests). Further, the API contains 2.038 lines of comments that allow the users to gain a better understanding of the API. The API, including PHPDoc, is available at its project Web site at `http://github.com/m0mo/ERPAPI`. The API is implemented by using the latest version of PHP, which is 5.3.8.

This section deals with the technical implementation of the ERP API and gives an introduction to its classes and packages.

5.4.1 Classes of the ERP API

To illustrate the architecture of the API, figure 5.1 presents a simplified version of the ERP API class diagram.

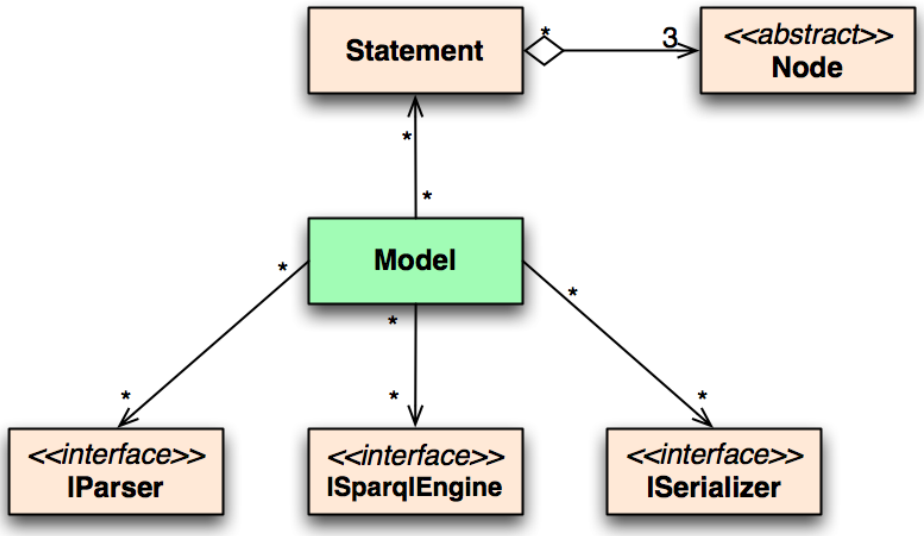


Figure 5.1: Simplified class diagram of the ERP API.

As highlighted, the whole API is centered around the *Model* class. An object of this class represents an in-memory RDF model (the model is stored in the computer's memory) and provides various functions for processing its statements and resources.

A *Model* object contains all the RDF data as a list of statements. Each statement consists of three nodes. We distinguish between three kinds of nodes in our API:

Literal Node: A literal node is represented by a *LiteralNode* object. This class supports both kinds of available literals in RDF, namely, plain or typed literals.

Resource Node: A resource node basically represents an object or a Web resource. It is implemented by the *Resource* class and identified by an URI.

Blank Node: A blank node is a node that is neither a resource or a literal. In RDF's abstract syntax, a blank node is defined as an unique node that can be used in one or more RDF statements, but has no intrinsic name [51]. Blank nodes within the ERP API are represented by the *BlankNode* class. To use more blank nodes within the same graph the API allows to identify them by an ID.

The hierarchical structure of the nodes is highlighted in figure 5.2.

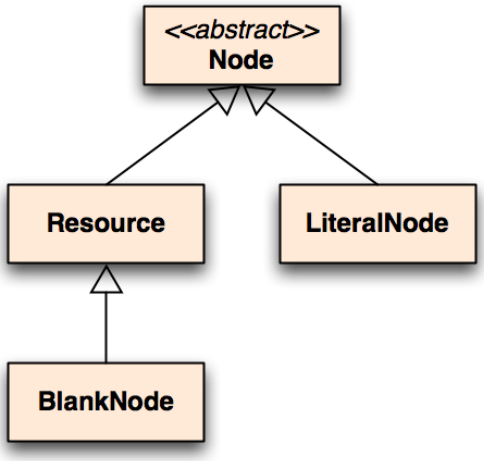


Figure 5.2: Class-diagram of the nodes of the ERP API.

Since resources described by RDFS or OWL can be represented as RDF statements, the API has no problem in processing and storing such kind of data. Therefore, these three nodes enable the user to create all kinds of RDF based documents. A more detailed description on RDFS and OWL is presented in section 5.5.6. Further, the *Model* class allows parsing (importing) and serializing (exporting) the model. Parsers and serializes as well as the supported formats are described in section 5.5.5 later this chapter.

In addition, the *Model* class allows to be queried by using SPARQL. More information on the SPARQL implementation is presented in section 5.5.7.

5.4.2 Packages of the ERP API

To ensure modularity we aimed for loose coupling between the classes and packages. The current version of the ERP API consists of six packages, namely, *model*, *serializer*, *parser*, *util*, *sparql* and *sparqlEngine* (shown in figure 5.3).

By defining extra interfaces to describe structures of classes, they or even complete packages can be altered or replaced by different implementations. Thus, these interfaces also allow developers to extend the API. For example, all new parsers and serializers that include the corresponding interfaces will work with the API.

In order to use the ERP API, users have to simply include the *API.php* file within their projects. This file requires all necessary packages and classes and, therefore, allows

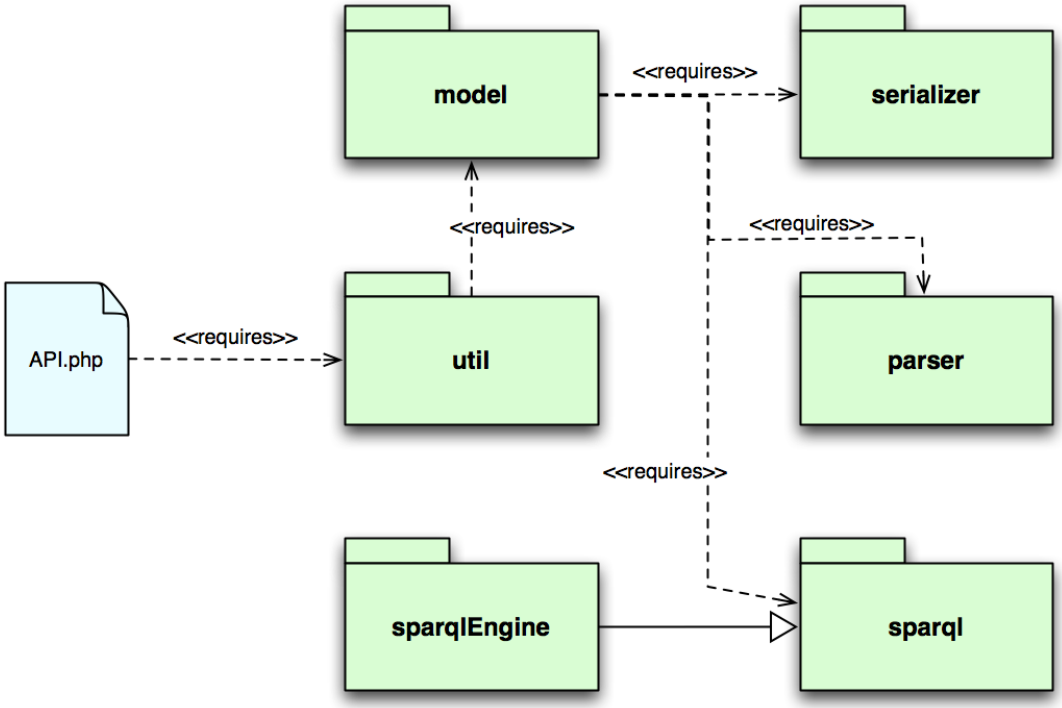


Figure 5.3: Package diagram of the ERP API

general usage. This includes the *model* package as well as all helper classes of the *util* package. The *model* package contains all node classes as well as the Statement and Model class. All other packages or classes are only included if they are needed. This allows the API to be more memory efficient.

We already gave a brief introduction to the packages *model*, *serializer*, *parser* and *sparql* corresponding to the classes presented prior, the packages *util* and *sparqlEngine* were not explained till now.

The *util* package provides useful static functions for the users and the API. Some examples for such functions are functions for checking the format of URIs or if an object is of a specific instance.

The *sparqlEngine* package is the SPARQL implementation of the ERP API. It is implemented as a plug-in and defined by the *ISparqlEngine* interface. By using this interface, it is also possible to use different SPARQL engines.

5.4.3 Advantages of PHPDoc

To ease the use of the API, all functions and global variables of the ERP API are described by using PHPDoc. PHPDoc extracts comments of functions and classes and generates a documentation of all available functions. Development environments take use of such comments and present the user a description of the available functions, their parameters and return values. This allows the users of the ERP API to achieve a better understanding of the API and, therefore, use it more efficiently. In addition, PHPDoc allows the definition of datatypes, which is not supported by PHP by default [15]. An example of the benefits of adding PHPDoc is presented in figure 5.4.

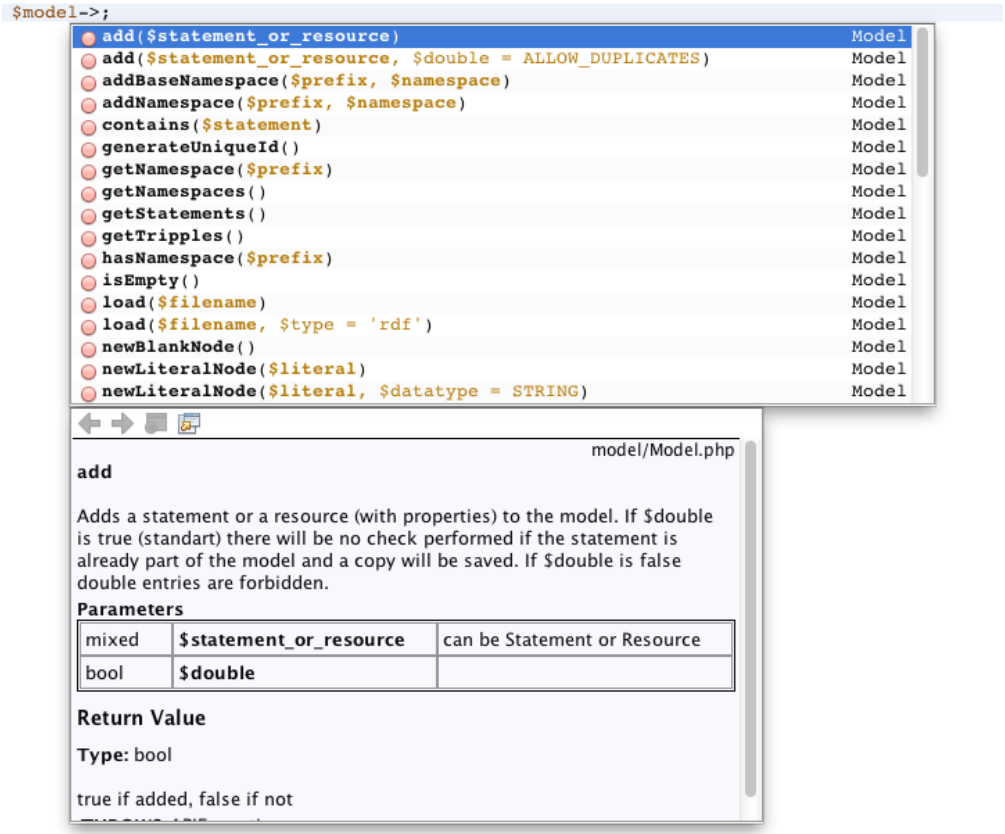


Figure 5.4: Example of the benefits of using PHPDoc within the ERP API.

5.5 Usage of the ERP API

As we discussed, it is easy to include the ERP API within new or existing projects. The easiest way is to include the *API.php* using the code *require_once 'path/to/API.php';*, which will load all necessary classes. The only setting a user has to configure (depending on the system) is the inclusion path, which is located in */util/Constants.php*. After that, the API is fully functional and ready to use. Further, the API provides static helper functions for creating all kinds of objects (for instance, line *$model = ERP::getModel();* creates a *Model* object). Therefore, the user can fully concentrate on the task rather than learn the API's architecture.

5.5.1 Statement-centric usage

As mentioned prior, the API provides two approaches for storing data: a statement-centric and a resource-centric approach. The statement-centric approach is similar to the approach from RAP. The user simply creates a few *Statement* objects and sequently adds them to the model. Code 5.1 illustrates the usage of this method.

Code 5.1: Example of using the statement-centric approach of the ERP API.

```
1  <?php
2          require_once 'path/to/API.php';
3
4         $model = ERP::getModel(); // create a new model using ERP function
5
6          //Create statements
7          $statement1 = new Statement(new Resource(NS . "e0625287"),
8                                      new Resource(NS . "firstName"),
9                                      new LiteralNode("Alexander"));
10         $statement2 = new Statement(new Resource(NS . "e0625287"),
11                                     new Resource(NS . "lastName"),
12                                     new LiteralNode("Aigner"));
13         // ...
14
15         // Add statements to the model
16         $model->add($statement1);
17         $model->add($statement2);
18         // ...
19 ?>
```

Further, line two of code 5.1 presents a common use of the static helper functions of the ERP API by creating a *Model* object using the code *$model = ERP::getModel();*.

5.5.2 Resource-centric usage

For more experienced users the ERP API provides the resource-centric approach. Here, the user creates a *Resource* or *BlankNode* object, which can be seen as the subject of a statement. These objects provide a function to add properties. Each property consist of two parameters: a predicate and an object node, for example, *$subject->addProperty($predicate, $object);*.

To increase efficiency we implemented "chain-adding". Chain-adding allows the user to minimize the written code by adding more properties in the same line. For example, the code *$subject->addProperty($p1, $o1)->addProperty($p2, $o2);* adds two properties to the *$subject*. As mentioned before, this concept was adopted from a similar concept of Jena. The ERP API is the only RDF API for PHP that enables this possibility. Example of the resource-centric approach with and without chain-adding is presented in code snippet 5.2.

Code 5.2: Example of using the resource-centric approach of the ERP API.

```
1  <?php
2          require_once 'path/to/API.php';
3
4          $model = ERP::getModel(); // create a new model using ERP function
5          $model->addBaseNamespace(PREFIX, NS); // setting a base NS
6
7          // Without chain-adding
8          $res = $model->newResource("e0625287");
9          $res->addProperty($model->newResource("firstName"),
10                                       new LiteralNode("Alexander"));
11         $res->addProperty($model->newResource("lastName"),
12                                       new LiteralNode("Aigner"));
13
14         // With chain-adding
15         $res = $model->newResource("e0625287")
16                 ->addProperty($model->newResource("firstName"),
17                                       new LiteralNode("Alexander"))
18                 ->addProperty($model->newResource("lastName"),
19                                       new LiteralNode("Aigner"));
20
21         // Add to model
22         $model->add($res);
23 ?>
```

Code 5.2 also shows usage of *$model* as node creator. The benefits from using this way of creating nodes is that the user can define a base namespace and all nodes that are created by the *$model* are created within this namespace.

For example, if the constant *NS* is defined as *http://example.org/*, the code *$model->newResource("firstName")* creates a new resource object, which is identified by the URI *http://example.org/firstName*. This is due the definition of a base namespace by using the code *$model->addBaseNamespace(PREFIX, NS);* in line five.

As we can see, such a definition is particularly useful, especially, if most nodes within a model belong to the same namespace. In code 5.1 we create the nodes by ourselves and have to assign a namespace by hand. Also here we used the namespace saved in the constant *NS*. It is obvious that for big RDF models it is probably more efficient to use the base namespace method.

Besides defining a base namespace, it is also recommended to define all used namespaces by adding them to the model using the line *$model->addNamespace($prefix, $ns);*. This ensures correct serialization and execution of SPARQL queries.

To add a statement or a resource we use the *$model->add($object, $bool)* function of the *Model* class. By default, this function allows to add duplicate entries. Therefore, it is not necessary to pass the second parameter. Anyway, it is possible to disable this by defining *$bool = false;*. It is important to note, that this method disables double entries only for a single addition. If the user wants to forbid double additions globally, he/she can disable it in the */util/Config.php* file.

5.5.3 Edit statements or resources of a model

Another important feature is to allow resources or statements to be edited. For this reason, the *Model* class provides a simple function called *edit()*. An example for editing a *Statement* object is illustrated in code 5.3.

Code 5.3: Example of editing statements using the ERP API.

```
1  <?php
2          require_once 'path/to/API.php';
3          // create or load a model, define a statement, e.g., $oldStatement
4
5          // Add to model
6          $model->add($oldStatement);
7
8          // modify statement
9          $model->edit($oldStatement, $newStatement);
10 ?>
```

As shown, the function *edit()* takes two parameters, the original statement and the edited one. For the resource-centric approach, the same function can be used by passing the original and edited *Resource* object.

5.5.4 Search statements or resources within a model

Further, it is possible to search statements in the model. The search function contains three parameters, namely, subject, predicate and object, and returns an array containing all results. If a parameters is defined as *NULL*, it can be seen as a placeholder and is true for all statements in the model. Since we want to enable users to further use the resource-centric approach we added two search functions. The function *$result = $model->search($subject, $predicate, $object)* merely returns a list of statements, while the function *$result = $model->searchResources($subject, $predicate, $object)* returns a list of resources with properties. Usage of the search functions is presented in code snippet 5.4.

Code 5.4: Example of searching statements or resources in the ERP API.

```
1  <?php
2          require_once 'path/to/API.php';
3
4          // create or load model
5          // create the $subject, $predicate, $object
6
7          // returns list of statements
8          $result = $model->search($subject, $predicate, $object);
9
10         // returns list of resources + properties
11         $result = $model->searchResources($subject, $predicate, $object);
12 ?>
```

5.5.5 Parsing and serializing models

Supported formats of the ERP API

One of our objectives was to support a variety of import and export formats. For the prototype of the API we included parsers and serializes allowing users to import or export RDF based documents of the following formats:

- RDF/XML
- N-Triple
- Turtle
- RDF/JSON

These four formats are supported by the ERP API as parsers as well as serializes. All of these parsers and serializes implement the corresponding ISerializer or IParser interface. Therefore, we allow easy extendability of the packages or exchange of the implementation.

As mentioned, they don't cover all available formats, but provide a usable foundation for the ERP API. Further, they are the most common ones used (see the comparison of APIs in chapter 4) by other APIs. Therefore, it is possible to import RDF documents created with other APIs to the ERP API. To parse or serialize a model, we only need one line of code as presented in code 5.5.

Code 5.5: Example of using ERP parsers and serializes.

```
1  <?php
2          require_once 'path/to/API.php';
3
4          $model = ERP::getModel(); // create a new model using ERP function
5
6          // Parsing a file
7          // $type is one of: rdf, ntriple, turtle or json
8          $model->load($filename, $type);
9
10         // process model ...
11
12         // Serializing to a file
13         // $type is one of: rdf, ntriple, turtle or json
14         $model->save($filename, $type);
15 ?>
```

Both the *$model->load($filename, $type);* and the *$model->save($filename, $type);* have two parameters. The parameter *$filename* simply defines the name of the file for loading or saving the model. The second parameter, *$type*, is more interesting. By default, the variable *$type* is set to *rdf*, which stands for the RDF/XML format.

To further illustrate the usage of parsers and serializers, we want to present the output of our serializes using the same example as for comparing the APIs. In summary, we created a model with one student identified by a matriculation number. Further, information about the birthday, name and the inscribed studies are added. For a specific study we added the studies code as well as an english and a german title. The creation of this model (using the resource-centric approach) is shown in code 5.6.

Code 5.6: Implementation of the example model for illustrating the serializers output.

```
<?php
        require_once 'path/to/API.php';

        $model = ERP::getModel();
        $model->addBaseNamespace("ex", "http://example.org/");

        $res = $model->newResource("e0625287")
         ->addProperty($model->newResource("firstName"),
                 new LiteralNode("Alexander", STRING))
         ->addProperty($model->newResource("lastName"),
                 new LiteralNode("Aigner", STRING))
         ->addProperty($model->newResource("birthday"),
                 new LiteralNode("1986-04-28", DATE))
         ->addProperty($model->newResource("studies"),
                  $model->newResource("businessInf")
                         ->addProperty($model->newResource("titleEN"),
                         new LiteralNode("Business Informatics", STRING, "en"))
                         ->addProperty($model->newResource("titleDE"),
                         new LiteralNode("Wirtschaftsinformatik", STRING, "de"))
                         ->addProperty($model->newResource("studyCode"),
                         new LiteralNode("E 066 925"))
                 );

         $model->add($res);
?>
```

RDF/XML format

The first format we want to discuss is the RDF/XML format. RDF/XML is probably the most important format for serializing a RDF graph. We implemented this format by using the XML functions that are already provided by PHP. Therefore, we provide increased speed and a solid foundation for easy extendability. The ERP API is the only PHP API using this way of implementation for the XML parser and serializer. For improving the parsers speed we took advantage of the XPath query engine. XPath can be described as a query language for querying XML based documents. Using XPath, it

is not necessary to process every line of the document, since we can jump directly to the nodes that are important for us. More information on XPath can be found at [56].

Using the command *$model->save($filename);* we serialize (save) the model into a file, identified by the variable *$filename*. Since RDF/XML is the default format, we don't have to pass the *$type* variable to the save function. For all serializers counts that a new file is created or overwritten if it already exists. This means that the ERP API (like all other APIs that are used for writing files) need to have permissions to create files on the local system. The content of the created file of our example model is shown in code 5.7.

Code 5.7: Example of RDF/XML code produced by the ERP API.

```
1  <?xml version="1.0"?>
2  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3                  xmlns:ex="http://example.org/">
4          <rdf:Description rdf:about="http://example.org/businessInf">
5                  <ex:titleEN rdf:datatype="xmlns:string" xml:lang="en">
6                          Business Informatics
7                  </ex:titleEN>
8                  <ex:titleDE rdf:datatype="xmlns:string" xml:lang="de">
9                          Wirtschaftsinformatik
10                 </ex:titleDE>
11                 <ex:studyCode rdf:datatype="xmlns:string">
12                         E 066 925
13                 </ex:studyCode>
14         </rdf:Description>
15         <rdf:Description rdf:about="http://example.org/e0625287">
16                 <ex:firstName rdf:datatype="xmlns:string">
17                         Alexander
18                 </ex:firstName>
19                 <ex:lastName rdf:datatype="xmlns:#string">
20                         Aigner
21                 </ex:lastName>
22                 <ex:birthday rdf:datatype="xmlns:date">
23                         1986-04-28
24                 </ex:birthday>
25                 <ex:studies rdf:resource="http://example.org/businessInf"/>
26         </rdf:Description>
27 </rdf:RDF>
```

N-Triple format

N-Triple and Turtle are related formats, as Turtle is a superset of N-Triple. Both parsers and serializers are implemented by using the PHP's file writer and reader functions.

The N-Triple format can be seen as a list of statements. Every line contains a string with three parts: the subject, predicate and object. These three sub-strings are separated by a whitespace. A dot on the end of the line indicates that the statement is complete.

URIs are represented by simply enclosing them in angle brackets, for example, *<http://example.org/e0625287>*. Literals are enclosed by quotation marks (for example, *"Alexander"*). The datatype of the literal is added by separating the literal value by two circumflexes and the string representation of the datatype enclosed in angle brackets (for example, *"Alexander"*^^<string>). The language part is identified by an "at sign" (@) and a language tag identified by two characters (for example, *"Business Informatics"@en*) [57, 58].

To include the output of the N-Triple serializer within this work, we had to re-format the code by adding a line break before the object string. Unfortunately, this was necessary to be able to include the code in this document. By default, the strings representing the subject, predicate and object are printed in one line. The (modified) content of the produced output file, using the command *$model->save($filename, "nt");*, is presented in code 5.8.

Code 5.8: Example of N-Triple code produced by the ERP API.

```
1  <http://example.org/e0625287> <http://example.org/firstName>
2                  "Alexander"^^<string> .
3  <http://example.org/e0625287> <http://example.org/lastName>
4                  "Aigner"^^<string> .
5  <http://example.org/e0625287> <http://example.org/birthday>
6                  "1986-04-28"^^<date> .
7  <http://example.org/businessInf> <http://example.org/titleEN>
8                  "Business Informatics"@en^^<string> .
9  <http://example.org/businessInf> <http://example.org/titleDE>
10                 "Wirtschaftsinformatik"@de^^<string> .
11 <http://example.org/businessInf> <http://example.org/studyCode>
12                 "E 066 925"^^<string> .
13 <http://example.org/e0625287> <http://example.org/studies>
14                 <http://example.org/businessInf> .
```

Turtle format

As mentioned prior, Turtle is a superset of N-Triple. It extends N-Triple by adding the support of using prefixes. Prefixes are defined on the beginning of each document by using the term *@prefix*, for example, *@prefix ex:<http://example.org/>*. The de-

fined prefix can be used to abbreviate the full namespace (like in XML) and, therefore, shorten the output code. Generally, Turtle has the same syntax as N-Triple. It also represents a RDF model as a list of statements, separating subject, predicate and object by whitespaces and a dot on the end of each line [63].

Since we can use prefixes, there is a difference in the representation of URIs. As mentioned, they are abbreviated using the pre-defined prefixes. Further, if using a prefix, they don't need to be enclosed within angle brackets. Anyway, if an URI does not fit a pre-defined prefix, it is saved using the N-Triple notation (full URI within angle brackets) [63]. To pursue our example, the command *$model->save($filename, "turtle");* creates a turtle document, which's content is presented in code 5.9.

Code 5.9: Example of Turtle code produced by the ERP API.

```
1  @prefix ex:<http://example.org/> .
2  ex:e0625287 ex:firstName "Alexander"^^<string> .
3  ex:e0625287 ex:lastName "Aigner"^^<string> .
4  ex:e0625287 ex:birthday "1986-04-28"^^<date> .
5  ex:businessInf ex:titleEN "Business Informatics"@en^^<string> .
6  ex:businessInf ex:titleDE "Wirtschaftsinformatik"@de^^<string> .
7  ex:businessInf ex:studyCode "E 066 925"^^<string> .
8  ex:e0625287 ex:studies ex:businessInf .
```

RDF/JSON format

The JavaScript Object Notation (JSON) is another important format. JSON's primary use is to transmit data between a server and Web application, serving as an alternative to XML [12, 14]. JSON is built on two structures [12, 14]:

1. A collection of name/value pairs. This is often realized as, for example, an object.
2. An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

While the ERP serializer is implemented as string serialization, the ERP parser uses the build in PHP JSON decoder. Therefore, we achieve increased speed and provide further reliability.

Code 5.10: Example of RDF JSON code produced by the ERP API.

```
{         "http:\/\/example.org\/businessInf":
        {
                "http:\/\/example.org\/studyCode":[
                        {
                                "value":"E 066 925",
                                "type":"literal",
                                "datatype":"string"
                        }],
                "http:\/\/example.org\/titleEN":[
                        {
                                "value":"Business Informatics",
                                "type":"literal",
                                "datatype":"string",
                                "language":"en"
                        }],
                ...
        },
        "http:\/\/example.org\/e0625287":
        {
                "http:\/\/example.org\/birthday":[
                        {
                                "value":"1986-04-28",
                                "type":"literal",
                                "datatype":"date"
                        }],
                "http:\/\/example.org\/firstname":[
                        {
                                "value":"Alexander",
                                "type":"literal",
                                "datatype":"string"
                        }]
                ...,
                "http:\/\/example.org\/studies":[
                        {
                                "value":"http:\/\/example.org\/businessInf",
                                "type":"uri"
                        }]
        }
}
```

Unfortunately, since the code is very space taking, we had to shorten the output code of our JSON serializer. However, the concept of JSON should be still understandable. The shortened RDF/JSON output, for our example, is presented in code 5.10.

As we can see in the JSON output, the ERP serializer also provides character escaping for the URIs and literals.

5.5.6 RDFS and OWL

As mentioned prior, the ERP API supports RDFS and OWL constructs. Since they can be described by using RDF triples, it is possible to simply create a *Statement* object to define RDFS or OWL constructs. To increase efficiency, we provide predefined constants for namespaces and prefixes as well as for important URIs such as *owl:Class*, *owl:subClass* or *rdfs:Comment*.

As we presented prior, it is easy to add a statement to the model. We simply have to use the node and statement objects of the ERP API and create elements using the right URIs. For instance, code 5.11 illustrates the creation of a statement for describing an OWL class called *Person*.

Code 5.11: Example of creating an OWL class within the ERP API.

```
1  <?php
2          $s = new Statements(
3                   new BlankNode("Person"),
4                   new Resource(RDF_TYPE),
5                   new Resource(OWL_CLASS)
6           );
7  ?>
```

If we add this statement to a model, we created our first OWL statement. As usual, other statements can simply refer to this statement. Code 5.12 illustrates, how we can simply define the sutdent of the previous example to be an instance of the *Person* class.

Code 5.12: Example of using an OWL statement within the ERP API.

```
1  <?php
2          $r = $model->newResource("e0625287")
3                   ->addProperty(
4                           new Resource(RDF_TYPE),
5                           new BlankNode("person)
6                   );
7  ?>
```

Alike this example, we can also define RDFS and other OWL resources for the model. Using the same way as presented in code 5.12 it is also possible to define relationships such as *rdfs:subClassOf* (for example, the line *$r->addProperty(new Re-*

source(RDFS_SUBCLASSOF), new BlankNode("animal")) defines the class *person* to be a sub-class of the *animal* class).

Further, since these constructs are creating using the statement-centric or resource-centric approach, they can be simply edited, removed or searched by using the same functions as normal statements or resources. For instance, the command *$result = $model->search(NULL, NULL, OWL_CLASS);* returns a list of all OWL classes. An example for editing a class is illustrated in code 5.13.

Code 5.13: Example of editing an OWL class using the ERP API.

```
1  <?php
2          require_once 'path/to/API.php';
3
4          // create or load a model
5          // define a statement, e.g., $oldStatement
6
7          $oldClass = new Statements(
8                  new BlankNode("Person"),
9                  new Resource(RDF_TYPE),
10                 new Resource(OWL_CLASS)
11         );
12
13         $newClass = new Statements(
14                 new BlankNode("PersonNew"),
15                 new Resource(RDF_TYPE),
16                 new Resource(OWL_CLASS)
17         );
18
19         // modify statement
20         $model->edit($oldClass, $newClass);
21 ?>
```

Serializers such as Turtle or N-Triple, which simply use the URI strings for describing a triple, have no problem to serialize statements as defined in code 5.14. Anyway, other serializers and parsers may need further adaptation. For instance, the RDF/XML serializer. By default, subject nodes are described using a *rdf:Description* XML node. An example of this serialization of the previously defined OWL class is illustrated in code 5.14.

Even though the fragment in code 5.14 is a valid description of an OWL class, it is not what more experienced users are expecting. For the definition of an OWL class, experienced users expect output such as *<owl:Class rdf:about="#person" />*.

Anyway, The RDF fragment of code 5.14 presents the the raw form of the RDF

Code 5.14: Example of a default OWL class created by the RDF/XML serializer of the ERP API.

```
1  <rdf:Description rdf:ID="person">
2      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class">
3  </rdf:Description>
```

serialization into XML for such a statement. In terms of RDF's information model, it expresses the same semantics as the compressed form *<owl:Class rdf:about="#person" />*. Therefore, the two code fragments are equivalent.

However, *<owl:Class rdf:about="#person" />* is considerably easier to read and tends to be the form most people come across when reading OWL. Therefore, we implemented a variable called *$type* that can be used to specify the name of the subject node for the RDF/XML serialization. For instance, using the constant *OWL_CLASS*, we can define the type of a subject *$r* by using the function *$r->setType(OWL_CLASS)*.

As expected, the default type is *rdf:Description* or, in terms of available constants, *RDF_DESCRIPTION*. If types are set correctly, it is possible to create a more defined serialization output. For example, by defining the type of the subject (using the command *$r->setType(OWL_CLASS)*) we achieve a serialization of the form *<owl:Class rdf:about="#person" />*.

Anyway, we still consider this implementation as a "workaround". Future development for the ERP API will contain the implementation of special classes and object (for instance, an OWL class could be defined by using the code *$c = new OWLClass("person")* to further ease the creation of OWL and RDFS constructs. Another idea how to refine the use of OWL and RDFS is to extend the API by a further RDF/XML serializer to provide both, the raw RDF output as well as a compressed output.

5.5.7 Perform SPARQL queries

SPARQL is an important part of an API, because it enables the user to perform complex reasoning tasks. Therefore, it was important to provide a SPARQL implementation, even though the development time was restricted.

To test the modularity of the ERP API, we implemented the SPARQL engine as a

module for the ERP API by using the *ISparqlEngine* interface. By sending a SPARQL query to the engine, the query string is first parsed to a SPARQL query object. This query object contains all necessary information and is used by the SPARQL engine to create a pattern for efficient querying.

The pattern is build according to this simple method: First, we check all elements of the *WHERE* block of the query. Next, the algorithm checks if some of this statements are connected (for example, the object of one statement is the subject of another one). If that is the case, the algorithm creates a tree, where the outermost "node" is the most general query and the innermost "node" is the most specific query. This algorithm expects, that statements, which are related to other statements by an object-subject connection, are more general. On the other hand, statements that are not related to others over their objects most likely contain actual data (no variables). The created pattern is then used to query the RDF model starting with the innermost queries. Further, if there are more possible choices of queries to start, the one with the least variables is used first.

The ERP SPARQL engine allows two return formats. By default, the engine returns the data as an array containing string values, commonly known by other SPARQL implementations. Since we wanted to go a step further, we decided to allow the API to return actual objects by providing an optional parameter. Therefore, users of the ERP API are able to directly process the returned information rather than simply presenting it to the user.

For both formats, the engine returns an array containing three elements: query, time and table.

1. The *query* element is the original query as a string.

2. The *time* element contains the time the query needed for execution. This can be used to tune the query for a better performance.

3. The *table* entry is the entry that contains the actual returned data in form of a two dimensional array (either strings or objects).

An example of using the SPARQL engine is shown in code snippet 5.15.

Code 5.15: Example of querying the model using SPARQL.

```
1  <?php
2          require_once 'path/to/API.php';
3
4          // create or load the model ...
5
6          // performing a query against the model
7          // returns a array containing strings
8          $model->sparqlQuery($query);
9
10         // performing a query against the model
11         // returns a array containing objects
12         $model->sparqlQuery($query, "objectarray");
13 ?>
```

5.6 Tests of the ERP API

To ensure the functionality of the ERP API, we performed a number of tests using PHPUnit [3] and by implementing it into two existing Web applications, namely, `www.boogiehasen.com` and `www.dancebook.at`.

In overall, we implemented 165 unit test cases, which perform 437 assertions. Using all these test cases, we tested all classes that belong to the ERP API.

As mentioned prior, one of our objectives was to achieve high code coverage. We managed to achieve 100% code coverage, which means that our tests executed all of the 1.937 lines of code at least once. Even though we can't ensure that there are no errors left, we can provide a high level of correctness of our API. Further, these tests present all possibilities of the ERP API and developers can use them as examples.

To perform further tests and collect data for the evaluation and comparison of the ERP API, we included it within two existing web-projects:

1. `www.boogiehasen.com`: This is a Web site of a successful Austrian Boogie Woogie dancing Club. The Web site is developed mainly by using PHP and Java Script. Most information is stored in a MySQL databases. Further, they maintain an admin-area, containing a list of all members. in addition, they store a list of all couples, danced competitions and results. The ERP API was included within the couple system and thus saving the couples and their competition results as a RDF document. By simply adding four lines of code (include the API, load the

model, add the data and save the model), we were able to achieve the inclusion of the API within this Web application. We included it in the way that, if new data was added, it would be automatically be saved within an RDF model. We further performed some simple SPARQL queries and searches. We also discovered that there is still potential for increasing the speed of SPARQL execution. This can be achieved by improving the pattern used for the execution.

2. `www.dancebook.at`: Dancebook is a growing social network for dancers in Europe. It maintains information about member profiles, events and more. By September 2011 dancebook lists 389 members and needs to process a vast amount of information. Together with the developer of dancebook, we included the ERP API within the member storage. Also here we included it in the way that new data was automatically added to a RDF model. The target was to test the API with a huge amount of data. Even though the ERP API is still in an early phase of development, it performed quite well in processing the information and didn't decrease the speed of the Web application significantly. For security reasons, we didn't process or save any usernames or passwords.

For both Web applications, we decided to save the data using the RDF/XML format. Further, we exported the model using other formats, but the RDF/XML format was the most convenient for the Web application's developers.

To include data that was already saved, we had to write a small PHP script, which iterates through the data and transforms it into a RDF model. We had to create such script for both Web applications.

In overall, the ERP API performed quite well if we consider the current level of its development. Even though the API works as intended, we are aware that there is still potential for improving the API.

CHAPTER 6

Comparison between ERP and ARC

This chapter outlines a final comparison between the ARC and ERP APIs. As mentioned prior, the ARC API is the only PHP API that is still in development. Therefore, it presents the only future proof option for developers, who want to use RDF within their projects.

This means that ARC marks the only concurrence for the ERP API. Thus, we think that it is important to create this comparison. It allows us to determine which fields it can still improve or where the ERP API is performing better.

6.1 Evaluation of ERP and ARC

We evaluated the ERP API based on the same evaluation model as used in chapter 4. Thus, we could reuse the evaluation results of the ARC API. Further, then both APIs are evaluated according to the same evaluation method and can be compared in all the previously defined characteristics. Therefore, we can also draw a more comprehensive comparison between the evaluation results of the ERP API and the results of chapter 4. The results of the evaluations of the ERP and the ARC APIs are presented in table 6.1.

Table 6.1: Results of the evaluation of the ARC and ERP APIs

Category	ARC	ERP
Usability	**75**	**95**
Effectiveness	100	100
Efficiency	60	100
Satisfaction	65	85
Likability	60	100
Trust	80	60
Pleasure	60	80
Comfort	60	100
Flexibility	**90**	**70**
Context Conformity	100	100
Context Extendibility	80	40
Accessibility	-	-
Safety	**80**	**60**
Commercial Harm	80	60
Operator Health	-	-
Public Health	-	-
Environmental Harm	-	-

6.2 Comparison of ERP and ARC

As we expected, the ARC API achieved better results in the characteristics flexibility and safety. As discussed prior, these characteristics raise as the API matures.

In the flexibility characteristic, the ERP API achieved 70 points. Since the ERP API is still in an early state of development, it is clear that the ARC API and its plug-ins present a more extendable API. Anyway, the ERP API provides good documented source code and usable interfaces. Thus, it allows developers to relatively easy extend its codebase. Further, the ERP API allows a flexible way of handling RDF data and enables users to take advantage of two different approaches to work with RDF models. The ERP API also provides a PHPDoc, which allows to get a better understanding of the functions and therefore how to use them more intelligent.

Also safety, defined by the characteristic commercial harm, is ranked lower than ARC. We previous defined commercial harm as the time it would take to update an API

to run on an updated system (for example, a new PHP version). Since ARC, nowadays, is developed by more people, the risk is relatively low. Anyway, the plug-ins for ARC don't provide the same reliability. Further, the API wasn't updated for quite a while, therefore, leading to a value of 80 points. In addition, ARC does not include any test cases that allow users to test the correctness of the API. The ERP API, on the other hand, presents higher risk due to low support by other developers. Since the ERP API is new and unknown in compare to ARC, the risk lowers as the API gets more popular and supported. In addition, the ERP API includes various PHPUnit test cases that make it easier to find errors and thus reduce the risk. Further, it is developed using the latest version of PHP and thus lowering the risk of errors in future PHP versions. Therefore, the ERP API achieved 60 points in the safety characteristic.

In the last characteristic, usability, the ERP API reached more points then the ARC API, which happens due to various reasons.

First, the use of PHPDoc. While ARC has an external documentation it does not use the power of PHPDoc. As mentioned above, by using PHPDoc the user can see a short description of all classes as well as their functions, which allows them to be used more effective. Since ARC does not use such documentation, most of its functions are undocumented. Therefore, the user often has no idea about the effect of a function as well as which parameter datatypes are required or what is the return value. Further, this makes it hard to describe all functionalities of the ARC API, since the documentation is rather spare.

The second reason is the use of unit tests. Unfortunately, the ARC API does not include test cases or examples. The only source of information is the spare documentation. The documentation provides enough information for setting up the general usage, but most of the functions are not documented. Therefore, it is hard to efficiently use the API, even it has more functionalities than the ERP API.

Third, ARC does not enable its users to take advantage of object-oriented programming paradigms. While the ERP API allows the creation of Node, Statement or Model objects to handle the data, ARC goes a different direction. It stores most of the data in multi-dimensional arrays. This approach is not as bad, but since most developers are already common by using objects, it is rather uncomfortable. For example, if we want to create a simple statement, we have to create arrays within arrays and fill them with

URIs by hand. Since PHP is relatively flexible with the creation of arrays (in compare to, for example, Java), it is easy to make mistakes on filling the arrays. The ERP API has various predefined constants for standard namespaces and prefixes, which is not available in the ARC API. Therefore, all URIs or namespaces have to be written as strings by the user. Functions such as adding a base namespace are not supported. Since ARC stores the data in arrays, it does not provide any helper functions for creating important objects. Thus, the users have to deal with the array structures themselves.

The fourth reason is the relatively complex setup of the API. Even though the setup is described in the documentation, it is not a common way of setting up an API for PHP. Mostly, PHP applications provide a configuration file where the user can find predefined variables that need to be changed according to the users needs. A common example is to define the MySQL username and password in the config file. Also for this the ARC API relies on arrays. The configuration of the API is performed by calling an API function and submitting an array of settings. Unfortunately, the possible settings are not well described and it is often not clear what they mean.

On the positive side, the SPARQL implementation of the ARC API is probably the API's most powerful functionality. It is even possible to set up a remote SPARQL store for querying the RDF document. The SPARQL implementation of the ARC API is very powerful and basically provides all possibilities of SPARQL. Unfortunately, ARC requires a MySQL database to perform SPARQL queries. Before the user can send SPARQL queries, the RDF document has to be transformed into a MySQL database. Therefore, also all SPARQL queries need to be transformed to MySQL queries. Since there is no available documentation of this SPARQL implementation, we can not really give a detailed description on the way ARC executes queries.

These are further system requirements that need to be considered and require additional configurations. Further, this approach also has downsides. Unfortunately, while testing we experienced that the MySQL version of our RDF model was not updated as we performed changes. This sometimes leaded to data loss and wrong SPARQL results. For example, the database still contained resources and statements that we have already deleted or newly added statements were not added. Even though ARC provides a setting to refresh the database on every use, the only way we could remove these flaws was to write a few lines of code that erased the database before sending the query. This forced

the API to reload all data and we achieved up to date information. Therefore, ARC received lower points in the efficiency characteristic.

The SPARQL implementation of the ERP API is still not as powerful, but queries can be performed without any additional requirements. ERP only relies on PHP functions to extract, filter and return the queried data. As mentioned before, the ERP SPARQL implementation allows to return objects rather than just string results. Therefore, efficiency can be increased, because the user can just continue working rather than create all objects again. Further, the use of PHP functions ensures that only the actual model is queried rather than a previously stored one. Unfortunately, due to the restricted development time, not all concepts of SPARQL are supported by the ERP SPARQL module till now. However, the SPARQL engine is already prepared for including the rest of the functionality. Further information on the ERP SPARQL implementation can be found in chapter 5. Due to these reasons, the ERP API was ranked with 95 and the ARC API with 75 points in the characteristic usability.

Generally, the functionalities of the ARC and ERP APIs are very similar. Both allow processing RDF elements as well as RDFS and OWL constructs. Up until now, the ERP API does not provide extra classes for RDFS and OWL elements. Even though RDFS and OWL can be used already, there are still opportunities to improve their usage. ARC does not provide any functions for RDFS or OWL. Within ARC they are seen as other kinds of URIs that need to be put in arrays.

The main functionality in which the ARC API is still leading is the implementation of SPARQL. Even though it has its problems, it presents a lot of opportunities for the ERP API. A SPARQL store allows the user to send remote queries and extends the possibilities of an API. Fortunately, such a store is not that hard to implement in a basic version.

The ARC API also provides more serializers and parsers. Due to the time restriction of the development, we were forced to choose some of the formats. During future development, new parsers and serializers will be included.

To conclude the differences of the two APIs, a brief comparison of functionalities is presented in table 6.2.

In a general comparison using all the evaluates APIs of chapter 4 the ERP API

Table 6.2: Comparison of functionalities of the ERP and ARC APIs.

Category	ERP	ARC
OOP Support	Yes	No
RDFS Support	Partial	No
OWL Support	Partial	No
SPARQL Support	Partial	Yes
Add Statements	Yes	Yes
Update Statements	Yes	Yes
Delete Statements	Yes	Yes
Search Statements	Yes	Yes
Modifiable	Yes	Yes
Modular	Yes	Yes
PHP Unit Tests	Yes	No
Import Formats	4	14
Export Formats	4	11
Other requirements	none	MySQL
Documentation	Web, PHPDoc	Web

ranks second place behind Jena in the characteristic usability. As mentioned before, the context we used for the evaluation is defined relatively simple. Therefore, we can't infer that the ERP API is better than all the other APIs. Anyway, we can state that the ERP API offers higher usability for basic RDF manipulation functionality. It should be clear that the other APIs still outmatch the ERP API in overall functionality, but, by providing a solid foundation, we increase the functionality of the API during maturing.

Even though, we can't say that the ERP API is performing better than the others in real usage, we think that the basic level of processing RDF documents is the most important. If common APIs don't provide high usability for this level, users might not be able to use the API at all.

Note: During the final stage of this work, the active code development of the ARC API was officially discontinued due to lack of funds and the inability to efficiently implement the ever-growing stack of RDF specifications. The ARC API is now hosted on

the same social-coding platform as the ERP API (`github.com`) and only updated by user submissions. Therefore, The ERP API is the only RDF API for PHP that is still in development.

CHAPTER 7

Conclusion

During writing this work, we realized that the situation of APIs is worse than we initially imagined. The comparison of popular APIs in chapter 4 pointed out that even well established APIs are still not able to provide easy interfaces for general users. The only satisfying API for RDF is Jena. It allows complex processing of RDF documents by providing easy to use interfaces. Jena is one of the best documented APIs and thus allowing the user to gain a better understanding of the API's functionality.

The results of the comparison were partially expectable, but still disappointing. We expected that by using only well established APIs we would be able to extract more strengths, but we mainly found weaknesses. We want to mention again, that we were limited by only one person who evaluated all the APIs. Anyway, during developing the ERP API, we tried to use these weaknesses as opportunities and we managed to create a considerable RDF API for PHP.

Even though the ERP API is still in an early stage of development, we can see great potential for this API. The evaluation of state of the art APIs from different systems pointed out lot of opportunities for the ERP API. ERP combines the strengths of famous RDF APIs like Jena, ARC or RAP and redefines the standard of RDF APIs for PHP.

Our research pointed out that there is a lack of usable PHP APIs and that the existing APIs are often too complicated for average users. Developers, who use such APIs, often see just the extra work, lowered performance and more problems. Unfortunately, till now there is no real benefit for average developers to use semantic technologies. The

benefits of using semantic technologies can only be enabled if these technologies get used more often. The ERP API might not increase the benefit to its users now, but it provides a simple and useful interface. Thus, it tries to reduce the negative impressions of using RDF APIs and like that encourage other developers to use them.

While Jena is a great example for an easy to use but powerful API for Java, PHP developers had no other future proof choice than using the ARC API. ARC also presents a powerful API, but does not provide usable interfaces for regular users. Further, ARC only provides a spare documentation and most of its functionalities are not documented.

By providing simple interfaces for adding, updating, deleting and searching RDF statements, the ERP API states an easy to use API. Further, it allows execution of SPARQL queries without any further requirements, besides PHP. It can parse and load as well as create a string representation of an RDF model by providing four different formats. By using interfaces to describe the single classes, the API provides an extensible and modular architecture. It already provides basic usage of RDFS and OWL and, therefore, it can be seen as a powerful API.

Further, the ERP API takes advantage of using PHPDoc. Thus, it provides a comprehensive documentation of all available functions and classes of the API. By providing various unit test cases, we include a lot of examples of using the API as well as make it easy to find errors during development. Therefore, users can easily gain understanding of the API and use it more efficiently.

We hope that the ERP API inspires developers of other RDF APIs for PHP or other languages to focus on regular users and provide easy and customizable interfaces. Only by providing usable APIs we can achieve the vision of the Semantic Web.

CHAPTER 8

Future Work

This work pointed out that semantic technologies, especially RDF, are still not used as often as desired. While there are already some promising APIs available, others often lack usable interfaces for average users. Therefore, the development of the ERP API aimed to combine valuable concepts from different APIs and show that a powerful API does not need to be complicated to use.

In the first step of the development of the ERP API, the target was to provide the most important functionalities. Besides providing an effective and efficient set of interfaces for manipulating and creating RDF documents, we managed to implement a good foundation for the API. Thus, we as well as other developers can improve and extend it. We hope that other (API) developers will contribute to the ERP API or get inspired by the newly introduced concepts of the API. Also, by providing the API on a social coding platform, we hope to encourage other developers to join the development.

Since the ERP API is still a prototype, there might be still errors or problems which we did not find. Even though, we performed extensive testing using PHPUnit tests and by implementing the API in existing Web applications, majority of problems will arise during the usage of the API by different users or by using the API in different contexts. To keep track of eventual problems, we can take advantage of the social coding platform where the API is provided. There, users can inform us about issues or even contribute to the ERP API.

But there are already a few points that we think are of importance for the future

development. The first point is the implementation of RDFS and OWL. Even though it is already possible to create OWL models or use RDFS within the ERP API, it might be still inconvenient. Therefore, in further versions of the API we will improve this functionality by implementing further interfaces for these uses.

Second point are the formats. The ERP API already provides four popular formats for serializing and parsing RDF models. Anyway, there are a lot of other formats available for that purpose. One important format that we still have to include is the N3 format. Since it is a superset of Turtle, we already have a good foundation for implementing this format.

Third, it is still necessary to increase the number of unit tests. Even though we have a high test coverage, it is possible that not all functionalities are tested extensive enough.

Fourth point is the search algorithm within the Model class. Until now the search algorithm iterates the statements to find the ones fitting to the input parameters. This method is commonly used by other APIs but can be still improved by indexing the statement array. Without indices and large amount of data, we might have to, in the worst case, iterate all statements to find the right one on the last position. By indexing the array in different ways, it is possible to achieve a higher efficiency of the search function, because the search function can use the indexes to order the array.

The fifth and most valuable point is the SPAQRL implementation of the ERP API. Due the complexity of SPARQL, we were not able to implement the complete syntax in the current state of the API. Up till now, the query functionality is implemented completely. Fortunately, we already prepared the ERP SPARQL engine so that the implementation of the rest of the syntax can be achieved rather easily. Anyway, to create an effective SPARQL implementation we need to perform further tests and require feedback by users. In addition, it is still possible to increase the efficiency and, therefore, the execution speed of the SPARQL engine by further improving the pattern used for querying the data.

Another point related to SPARQL is the implementation of the SPARQL parsers. This parser transforms the query string into a query object, so it can be used by the engine. While we increase the functionality of the engine, we also have to improve the parser. Until now, we realized the parser by using regular expressions to check the format as well as extract the necessary information of the query. By exploring other

possibilities to parse the query we might still be able to increase the speed of the parser.

While there might be other points too, in our eyes, these are the most important ones. By implementing these functionalities, we can create an API that can not only compete in the basic manipulation level but also in more complex areas.

List of Figures

List of Tables

List of Codes

Bibliography

[1] Aduna. openrdf.com . . . home of sesame. http://www.openrdf.org/. Last Accessed: 2011-08-02.

[2] Luis Argerich. Php xml classes a collection of classes and resources to process xml using php. http://phpxmlclasses.sourceforge.net/. Last Accessed: 2011-08-02.

[3] Sebastian Bergmann. Phpunit. http://github.com/sebastianbergmann/phpunit, September 2011. Last Accessed: 2011-09-07.

[4] Tim Berners-Lee. Www: Past, present and future. *Computer*, 29(10):69–77, October 1996.

[5] Tim Berners-Lee. The semantic web. *Scientific American*, pages 34–43, 2001.

[6] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world-wide web. Magazine: Communications of the ACM, Volume 37 Issue 8, Aug. 1994, August 1994.

[7] Tim Berners-Lee, Larry Masinter, and Michael McCahill. Uniform resource locators (url). Technical report, Network Working Group T. Berners-Lee Network Working Group, 1994.

[8] Nigel Bevan. Extending quality in use to provide a framework for usability measurement. In Masaaki Kurosu, editor, *Human Centered Design*, volume 5619 of *Lecture Notes in Computer Science*, pages 13–22. Springer Berlin / Heidelberg, 2009.

[9] Advisory R. S. S. Board. RSS 2.0 Specification. Technical report, RSS Advisory Board, 06 2007.

[10] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In Ian Horrocks and James Hendler, editors, *The Semantic Web — ISWC 2002*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer Berlin / Heidelberg, 2002.

[11] Vannevar Bush. As we may think. *interactions*, 3:35–46, March 1996.

[12] Douglas Crockford. Introducing json. http://www.json.org/. Last Accessed: 2011-09-07.

[13] DedaSys LLC. Programming language popularity. http://langpop.com/. Last Accessed: 2011-08-09.

[14] ecma INTERNATIONAL. ECMA Script Language Specification (Standard ECMA-262). Technical Report Edition 5.1, ecma INTERNATIONAL, June 2011.

[15] Joshua Eichorn. phpdocumentor. http://www.phpdoc.org/, March 2008. Last Accessed: 2011-09-07.

[16] Epimorphics Ltd. Jena – a semantic web framework for java. http://jena.sourceforge.net/. Last Accessed: 2011-08-02.

[17] Orri Erling and Ivan Mikhailov. Rdf support in the virtuoso dbms. In Tassilo Pellegrini, Sóren Auer, Klaus Tochtermann, and Sebastian Schaffert, editors, *Networked Knowledge - Networked Media*, volume 221 of *Studies in Computational Intelligence*, pages 7–24. Springer Berlin / Heidelberg, 2009.

[18] Marc Hassenzahl. The Effect of Perceived Hedonic Quality on Product Appealingness. *International Journal of Human-Computer Interaction*, 13(4):481–499, 2001.

[19] Hewlett-Packard Development Company, L.P. Hp labs semantic web research. http://www.hpl.hp.com/semweb/, 10 2009. Last Accessed: 2011-11-03.

[20] Ian Horrocks. Semantic web: the story so far. In *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, W4A '07, pages 120–125, New York, NY, USA, 2007. ACM.

[21] International Electrotechnical Commission. International electrotechnical commission. http://www.iec.ch/, 10 2011. Last Accessed: 2011-11-03.

[22] International Organization for Standardization. Iso/iec 9126:2001, 2001.

[23] International Organization for Standardization. International organization for standardization. http://www.iso.org/, 10 2011. Last Accessed: 2011-11-03.

[24] International Organization for Standardization. Iso/iec fdis 25010:2011, 03 2011.

[25] Karlsruher Institut für Tehcnologie. Content-driven knowledge-management through evolving ontologies - http://www.aifb.kit.edu/web/on-to-knowledge, 09 2002. Last Accessed: 2011-11-03.

[26] Michael Kifer, Jos De Bruijn, Harold Boley, and Dieter Fensel. A realistic architecture for the semantic web. In *In RuleML*, pages 17–29. Springer, 2005.

[27] Brian McBride. Jena: Implementing the RDF Model and Syntax Specification. In *Semantic Web Workshop 2001*, 2001.

[28] Car McDaniel and Roger Gates. *Marketing research essentials*. International Thomson Publishing, 2nd edition, 1998. Pages 247-249.

[29] Microsoft Corporation. About dotnet. http://www.microsoft.com/net. Last Accessed: 2011-09-29.

[30] Benjamin Nowack. Easy rdf and sparql for lamp systems. http://arc.semsol.org/. Last Accessed: 2011-08-02.

[31] Benjamin Nowack. Arc: appmosphere RDF classes for php developers. In Sören Auer, Chris Bizer, and Libby Miller, editors, *Proceedings of 1st Workshop on Scripting for the Semantic Web (SFSW05)*, volume 135 of *CEUR Workshop Proceedings ISSN 1613-0073*, June 2006.

[32] Marek Obitko. Semantic web architecture. http://obitko.com/tutorials/ontologies-semantic-web/semantic-web-architecture.html. Last Accessed: 2011-05-12.

[33] Radoslaw Oldakowski, Christian Bizer, and Daniel Westphal. Rap - rdf api for php v0.9.6. http://www4.wiwiss.fu-berlin.de/bizer/rdfapi/. Last Accessed: 2011-08-02.

[34] Radoslaw Oldakowski, Christian Bizer, and Daniel Westphal. Rap: Rdf api for php. In *IN PROC. INTERNATIONAL WORKSHOP ON INTERPRETED LANGUAGES*. MIT Press, 2004.

[35] OpenLink Software. Virtuoso universal server. http://virtuoso.openlinksw.com/. Last Accessed: 2011-08-02.

[36] Oracle Corporation. About java. http://java.com/en/about/. Last Accessed: 2011-09-29.

[37] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34:16:1–16:45, September 2009.

[38] Davey Shafik. Pear package information: Rdf. http://pear.php.net/package/rdf. Last Accessed: 2011-08-02.

[39] Y. Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. Technical Report RFC 4180, IETF, October 2005.

[40] Steve Suehring, Tim Converse, and Joyce Park. *PHP 6 and MySQL*. Wiley Publishing, Inc., 2009.

[41] The PHP Group. php.net. http://php.net/. Last Accessed: 2011-09-07.

[42] Larry Ullman. *PHP 6 AND MySQL 5*. Peachpit Press, 2007.

[43] Rob Vesse. dotnetrdf - semantic web/rdf library for c#/.net. http://www.dotnetrdf.org/. Last Accessed: 2011-08-02.

[44] Noah Wardrip-Fruin. What hypertext is. In *HYPERTEXT '04 Proceedings of the fifteenth ACM conference on Hypertext and hypermedia*, 2004.

[45] Wikipedia. Unicode. http://en.wikipedia.org/wiki/unicode. Last Accessed: 2011-05-12.

[46] Wikipedia. Universal character set. http://en.wikipedia.org/wiki/universal_character_set. Last Accessed: 2011-05-12.

[47] Wikipedia. Best coding practices, http://en.wikipedia.org/wiki/best_coding_practices, 01 12. Last Accessed: 2012-01-19.

[48] Wikipedia. Application programming interface. http://en.wikipedia.org/wiki/api, 10 2011. Last Accessed: 2011-11-03.

[49] Wikipedia. C (programming language). http://en.wikipedia.org/wiki/c_(programming_language), 09 2011. Last Accessed:2011-09-29.

[50] Wikipedia. The internet. http://en.wikipedia.org/wiki/internet, 09 2011. Last Accessed: 2011-09-29.

[51] World Wide Web Consortium. Resource description framework (rdf): Concepts and abstract syntax. http://www.w3.org/tr/rdf-concepts/. Last Accessed: 2011-07-29.

[52] World Wide Web Consortium. Semantic web. http://www.w3.org/standards/semanticweb/. Last Accessed: 2011-05-12.

[53] World Wide Web Consortium. Tools that are listed as relevant to rdf. http://www.w3.org/2001/sw/wiki/rdf. Last Accessed: 2011-08-02.

[54] World Wide Web Consortium. Overview of sgml resources. http://www.w3.org/markup/sgml/, 2004 03. Last Accessed: 2011-11-03.

[55] World Wide Web Consortium. Xml schema. http://www.w3.org/xml/schema, 2011 11. Last Accessed: 2011-11-03.

[56] World Wide Web Consortium. Xml path language (xpath). http://www.w3.org/tr/xpath/, 11 1999. Last Accessed: 2011-11-03.

[57] World Wide Web Consortium. N-triples - w3c rdf core wg internal working draft. http://www.w3.org/2001/sw/rdfcore/ntriples/, September 2001. Last Accessed: 2011-09-07.

[58] World Wide Web Consortium. Rdf test cases. http://www.w3.org/tr/rdf-testcases/#ntriples, February 2004. Last Accessed: 2011-09-07.

[59] World Wide Web Consortium. Rdf vocabulary description language 1.0: Rdf schema. http://www.w3.org/tr/rdf-schema/, February 2004. Last Accessed: 2011-09-08.

[60] World Wide Web Consortium. Sparql query language for rdf. http://www.w3.org/tr/rdf-sparql-query/, January 2008. Last Accessed: 2011-09-08.

[61] World Wide Web Consortium. Owl 2 web ontology language document overview. http://www.w3.org/tr/owl2-overview/, October 2009. Last Accessed: 2011-05-12.

[62] World Wide Web Consortium. Extensible markup language (xml). http://www.w3.org/xml/, 04 2011. Last Accessed: 2011-11-03.

[63] World Wide Web Consortium. Turtle - terse rdf triple language. http://www.w3.org/teamsubmission/turtle/, March 2011. Last Accessed: 2011-09-07.

Index