Diploma Thesis

Exposing Domain Models as Linked Data

Sebastian Kurfürst

Dresden University of Technology
Faculty of Computer Science

Institute for System Architecture
Chair for Computer Networks

Professor:
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill, TU Dresden

Supervisor:
Dipl.-Medien-Inf. David Urbansky

14.11.2011

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe. Wörtliche und sinngemäße Zitate sind als solche gekennzeichnet. Diese Arbeit wurde noch keiner Prüfungskommission in gleicher oder ähnlicher Form vorgelegt, und auch nicht veröffentlicht.

Dresden, den 10.11.2011

_____

Sebastian Kurfürst

CONTENTS

CHAPTER

ONE

INTRODUCTION

Linked Data is a technology which can revolutionize the way we work with information on the web. However, it is only used on a few hundred sites so far. In order to foster participation in the Linked Data movement, we build a connection from Linked Data to Domain-Driven Design, a widely applied development practice for building enterprise applications. Specifically, we create a framework for application developers which makes participation in the Linked Data movement straightforward.

1.1 What Is Linked Data and Why Should We Care?

Hypertext forms the foundation of the World Wide Web (WWW). It is a powerful infrastructure to link resources together. The web is centered around *documents*, whereas each document can link to an arbitrary number of other documents. Through the use of Uniform Resource Identifiers (*URIs*), it is possible to reference other target documents not being on the same server. The domain name acts as a namespace and encodes the origin of a document. Because of that, one can view the World Wide Web as a huge *distributed graph of interlinked documents*. The most prominent benefit is that interlinking reduces the duplication of data in the network, because instead of embedding a document directly (and thus duplicating its content), one can simply cross-link it[1].

However, when looking at this graph in detail, it is often visible that it is not the document itself which is relevant to the user, but the information embedded *inside this document*. Thus, at the next level, it makes sense to link the information embedded in documents together, reducing duplication again, thus enabling better re-use of information. That is the core idea of Linked Data.

All this interlinked information also forms a (distributed) graph, which is often called the *Giant Global Graph*. By exposing *semantic information* in this graph, a computer can extract the structure of this information, instead of only the structure of documents available in the WWW. Thus, we can now use computers to specifically query the information stored in the graph.

Because we now work on the semantic level, we can directly work with knowledge. This can be the basis for semantic search engines, information mashups or data aggregation platforms, to name a few.

Imagine you want to find a list of all events happening at a certain date in your hometown. Right now, you would enter some queries into a search engine (like "Events Washington 2011"), and get back a list of documents where parts of this search text are found – the search engine works

[1] It is hard to imagine today's internet without linking. Following links is such a core interaction pattern of the Internet that literally every user is accustomed to it, without even thinking about it consciously.

on a *syntactical level.* Then, you need to browse through the result list, and manually extract the needed information from the result documents. Still, you will certainly find events multiple times (on different webpages), so you need to manually build a list of all events in your hometown on a given date, filtering duplicates by hand. While you would certainly do this manual searching once in a while, if you need the event information every few days or even daily, it will quickly become a very tedious task. Furthermore, you would certainly get many results not relevant to you, as *"Washington"* is ambiguous – the system does not know if you mean the former president of the United States, the capital of the US or the state.

Contrary, if the information about the events is stored as Linked Data, you can specifically query a search engine to get a list of *events* on the given *date* in the *town Washington* - now you can express the query on a much more fine-grained level, so the search engine can collect the information you wanted to have without you having to take manual action.

There is another benefit of Linked Data: As the information is now explicitely represented, it is possible to apply *reasoning techniques* when querying Linked Data, making implicit knowledge explicit. For example, we can also modify the query from above to fetch all events within a range of 20 km around our hometown; and while the location of these cities is part of the information exposed, the relative distance to all other cities is not. However, it can be inferred by some geometrical calculations. This would not be possible in a generic way without Linked Data.

The use cases explained above deal with *data aggregation*. It means the extraction relevant information from multiple sources and presenting it to the user. Nowadays, this kind of application is called *mashup*. However, todays mashups need to query the information from the source documents via proprietary APIs, making it difficult to create universal mashups not tied to a particular backend system. Furthermore, the data integration step must be programmed manually in most cases. Contrary, mashups based on Linked Data mashups can use a set of standardized tools to modify and extract information, as the core idea of Linked Data is a *universal language to express information*. Thus, writing data aggregation applications based on Linked Data will become much easier than it was before.

Now we have seen two use cases which clearly show the benefits of Linked Data for consumers of information (i.e. the casual internet user). However, what are the benefits of *content providers* to take part in the Linked Data movement? When consumers interact with a semantic search engine, the search result listing might already contain all the expected information. Thus, the user will find the information earlier than right now, maybe not even needing to visit the webpage of the content provider; so the user will not see any advertisements shown to refinance the operation of the service. So, at first sight, this seems to be clearly a drawback from the perspective of a content provider. However, by providing Linked Data, one can also reach potential customers in a much quicker way, as search engines started to rank these elements higher than normal results, and also shown in more detail (which – in turn – helps to gain visibility). Furthermore, consuming Linked Data from other sources in order to provide additional information would be very helpful for many companies as well. If all the information was stored in one exchangeable format, it becomes a lot easier to integrate information from other companies, potentially making the companys employees more productive.

Generally each company needs to decide by itself if it wants to create a "walled garden" or participate in an "open landscape" in terms of information.

Because of its distributed and universal nature, Linked Data is a very promising concept for revolutionarizing the way data is dealt with on the information level.

However, as of today, there are only a few hundred web sites which export their information as Linked Data. Interestingly, enterprise web applications such as blogs, web shops or content management systems are not participating on a big scale in the Linked Data movement. In the next section, we will analyze this problem further and see how we can make participation in the Linked Data cloud easier for this target group.

1.2 Relation to Domain-Driven Design

Development of software is a creative process. When implementing an application, the developer first has to find out the goals of the customer, and has to understand the problem the customer wants to solve using the to-be-created application – so the developer has to understand the so-called *application domain*. Often, there exist big communication problems between the customer (who is the *Domain Expert*) and the developer, as they often speak a different language (see [saiedian99]). *Domain-Driven Design (DDD)* [Evans] is a development strategy which, besides technical concepts, also contains ideas to structure the creative process in software development. This makes the developer more effective and lessens the risk of misunderstandings with the customer.

Because of the above reasons, Domain-Driven Design is widely applied in the development of business applications, as it provides a conceptual framework for solving the customers' problems in a well-structured manner.

One core idea of DDD is to create a software representation of the problem domain with objects, the so-called *Domain Objects*. These should focus on the real world behavior and state, not containing infrastructural logic like persistence, export/import or security checks. As they are simple objects, they are quite easy to read and understand, and one of the goals of DDD is to be able to discuss critical control flows directly using the Domain Objects.

While the core idea of DDD is independent of any technical framework, there exist frameworks in many languages which are centered around the principles of Domain-Driven Design. One of these frameworks is *FLOW3*, which is built a PHP-based framework used to create web applications. Next to Domain-Driven Design, it contains many powerful features such as Dependency Injection, Aspect-Oriented Programming (AOP), a flexible MVC stack, Security Framework, and a robust object-relational mapper (Doctrine2). This raises the abstraction considerably, making it possible as developer to only work with objects, and focus on the modelling process.

Now, what has all this to do with Linked Data? If we look at the state of Linked Data right now, as it is visualized by Figure 1.1, it becomes obvious that it has not yet reached widespread adoption, having only a few hundred participating entities right now. Compared to the huge number of enterprise applications used throughout the world, this is a very small number. Now, if it was possible for developers of these enterprise applications to *easily participate* in the Linked Data cloud, the amount of data in the Linked Open Data (LOD) cloud could literally explode. We will take FLOW3 as an example for an Enterprise Web Framework, featuring many powerful abstractions and technologies, and build support for Linked Data right into the framework. Then, every application built on FLOW3 will have the chance to easily participate in Linked Data, exposing more and more data, growing the Linked Data ecosystem.

To stress it again, the focus of this work lies in the *ease of use* for application developers, so the goal is to support the developer whereever we can, such that he does not need to be a Linked Data expert to export his data.

FLOW3 serves only as a case-study here – most of the techniques used in this work will be applicable to other state-of-the-art web application frameworks.

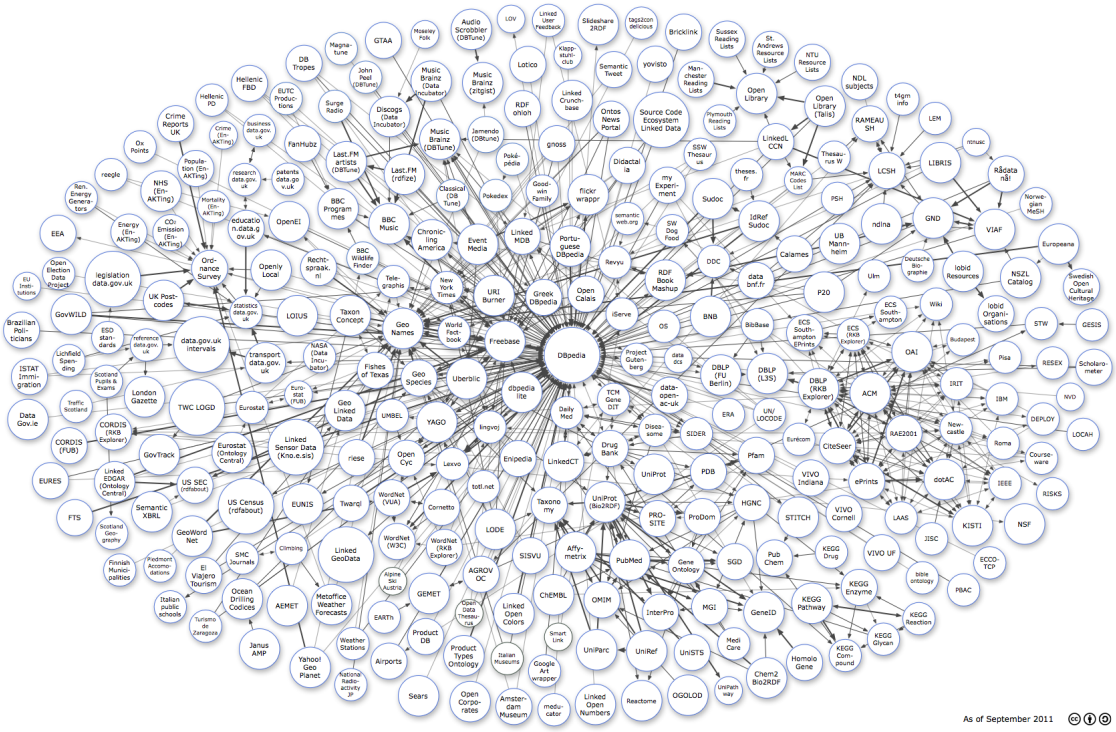

Figure 1.1: The current state of Linked Data. While the graph contains a few hundred nodes, it is very small compared to the millions of nodes in the document-based internet. Source: Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch, http://lod-cloud.net/

1.3 Research Questions

Who benefits from this work?

In order to define the goals and questions of this work, we need to define some common vocabulary of stakeholders who benefit from this work.
First, there is the *developer* who implements a FLOW3 based web application. He should experience productivity improvements when working with the presented framework.
Second, there is the *user* who is interacting with the web application the *developer* has built. He usually manages information inside the web application, and is responsible for the main part of the content in the system. For him, an easy- to-use user interface is most important, as using Linked Data might increase the amount of work he needs to do.
Third, there is the *visitor* of the web application, who is mainly profiting from the information in the web application. He does not care about backend technologies, but he will benefit from semantic markup and the use of RDF, as this can potentially change the way he interacts with the web application.

1. How is it possible to export all data in the system as Linked Data with minimal involvement of the *developer*?

   The first possibility to grow the popularity of Linked Data is to have more data available. Thus, it must be as easy as possible to export all data in the web application into formats supported by the Semantic Web.

   Thesis: It is possible to export all data in the system as well-formed RDF with minimal knowledge of the developer. The generated RDF needs to adhere to best practices in the industry, like [RDFpub08].

2. How can a *user* who browses a classical, document-based website, discover additional information through Linked Data?

   When the user browses through the document-based web, the browser must be able to detect if there is further semantic information available. RDFa bridges this gap between the document-based and semantic internet. Thus, in order to connect these two worlds together, we need to enrich web application output with RDFa information.

   Thesis: It is possible to automatically enrich an MVC-based web application by inserting RDFa tags, without further developer interaction. The generated RDFa contains links to further information.

3. How can we present the *user* a possibility to create *backlinks* to other Linked Data sources, f.e. places, persons or events?

   Linked Data gains an enormous amount of its value by *interlinking* with other Linked Data sources. Thus, there needs to be a way to create links to other sources. A fully-automated linking solution is not desirable, as it inevitably will add incorrect links, and thus the links cannot be trusted anymore.

   Thus, Backlinks to the LOD cloud must at least be verified by the *user*, such that we can confidently assume they are correct.

   Thesis: It is possible to build an intuitive user interface which hides the complexity of Linked Data searching. This user interface does not have to be built by the *developer*, but will be directly integrated into the framework. It is easily possible to embed this in a wide variety of web applications.

4. How can continous text be enriched with semantic data?

   Information in web applications can be roughly categorized into two groups: First, there is *explicit information* which is used inside the web application in various ways. An example for this category would be the `price` in a web shop application, as it is used in various places of the application, like for calculating taxes, shipping costs, or a total price. Normally, this kind of data is stored in separate fields in the Domain Model.

   Secondly, information can be more unstructured, for example just being continuous text. In a web shop, the article description is such an example: It contains a lot of information, but which is only accessible to humans, not being machine-readable. This kind of information is often stored inside a `string` field in the Domain Model.

   While we dealt with structured information in research question 3, we will focus on unstructured information here.

Thesis: We can automatically extract certain entities like persons, companies or places from unstructured text, and present an interface to the *user* where he can verify them. It is also possible to learn from the user's decisions in the past, and adapt the system to the behavior of the user.

1.4 Overview of This Work

In *Chapter 1: Introduction*, the first introduction into the topic of this work takes place. This involves explaining the benefits of Linked Data, and connecting this to Domain-Driven Design. Furthermore, the research questions are explained.

Chapter 2: Foundational Techniques is intended for people new to Linked Data or Domain-Driven Design. For both topics, an introduction is given, laying the groundwork for the remainder of this work.

After the foundations, the current state-of-the-art is analyzed in *Chapter 3: Related Work*, highlighting pros and cons of many existing solutions.

Chapter 4: Concepts introduces the reader to the concepts developed in this work, connecting Linked Data and Domain-Driven-Design in a novel way.

After introducing the concepts from a high-level standpoint, *Chapter 5: Implementation Details* focusses on the specific challenges which were solved during implementation. Furthermore, this chapter contains a walk-through for people who want to use the developed framework, demonstrating it step-by-step.

Chapter 6: Evaluation introduces measurements and assessments of the developed work, again comparing it with the solutions which existed before.

In *Chapter 7: Future Work*, possible next development steps and research questions are discussed. Furthermore, the long-term vision is shaped, showing how the developed application is embedded in a greater Semantic Web context.

This work ends with *Chapter 8: Summary*, which sums up the main contributions of this thesis.

CHAPTER

TWO

FOUNDATIONAL TECHNIQUES

In this chapter, basic technologies which are the foundation of this work are introduced and explained: Linked Data and Domain-Driven Design. We focus on how information is modelled in both techniques, show the technical underpinnings of each technology and related standards and techniques.

2.1 Linked Data

Linked Data describes an idea of a globally interlinked *information graph*, built on web standards. It is commonly believed that Linked Data is the basis for the next iteration of the world-wide web, superseeding and extending the document-based internet everybody knows and uses today. Here, we will describe the core ideas of Linked Data, first from a conceptual angle, and then discuss the implementation details.

We will start with a basic concept of the world, which is very important for modelling information: Identity.

2.1.1 The Concept of Identity

According to *Leibnitz's Law* [leibnitz08], two objects are identical, if they have all their properties in common. On the contrary, if only a single property between the objects differ, they are two different objects having two different identities.

However, in computer systems, it is often helpful to abstract parts of the real world away, in order to simplify the model of the real world. This is why many computing systems introduce a notion of *identity properties*: If all identity properties of two objects are equal, these objects are considered the identical objects. Thus, this is a specialized version of Leibnitz's Law, not taking into account all properties, but only a defined subset of them.

When modelling the real world in a computer system, it has to be consciously decided what the identity properties of the real-world objects should be, as the following examples demonstrate:

> A house is located at a particular address (which could be used as an identity), a city is located at a certain geographical location, and a country can be uniquely identified by its country name. Thus, it makes sense to use these properties (which are just a subset of all the properties of the object) as identity properties.

Identity properties such as the above are called *natural properties*, as they are inherent to the object described, and also transport some semantical meaning.

Contrary to that, in many cases, *artificial identities* are used which *do not carry any meaning*, but are just used to determine if two objects are similar or not. Artificial identity properties are used in a lot of places in computing systems, and in the real world as well:

> Passport numbers have been introduced because name, birth date and -place are not sufficent to distinguish all people. The German government assigns a unique tax identification number to people, and books are identified by ISBNs.

However, it is not always easy to decide what the identity of an object should be, as the following example demonstrates:

> To determine the identity of a book, the ISBN number can be used to disambigurate it from other books. Now, if a library has multiple books of the same ISBN in stock, it somehow needs to distinguish between the book title, and the particular "instance of the book".
>
> Thus, in this context, it needs to use some other identity properties than a book review web site, where the ISBN is a good identity property. This example shows that deciding on which identity to use can depend on the current use-case, and that there is not always a single universal identity which fits all use cases.

After this digression about identity properties, we will now see how RDF can be used to model information. We will see that identity plays a crucial role there.

2.1.2 Modelling Information with RDF

The Resource Description Framework (RDF) is a standard being used to describe *resources* and information about them. It is the core standard of Linked Data and the Semantic Web.

A *resource* is an object with a unique *identity*. In RDF, the identity of a resource is specified by an Uniform Resource Identifier (URI)[1]. This URI is an *artificial identifier:* It does not convey any meaning.

To describe a resource, one basically constructs sentences whose subject is the resource, the predicate identifies the property being described, and the object is the value of the property. Thus, the sentences have the form `<Subject> <Predicate> <Object>`, such as: `<Chris> <lives in> <Berlin>`. These sentences are called *Triples.*

From these triples, a directed graph can be constructed, and every RDF graph can also be described with such triples. As an example, the graph representation for Listing 1 can be found in Figure 2.1.

```
1 <Chris> <lives in> <Berlin>.
2 <Chris> <is born in> <Dresden>.
3 <Sebastian> <has friend> <Chris>.
4 <Sebastian> <lives in> <Dresden>.
```

Listing 1: Information modelling using triples

[1] Technically, it is an IRI (Internationalized Resource Identifier), which allows special characters in the URI. However, this is a detail which is not relevant in our case, and as URI is used more commonly, we use URI in the remainder of this work.

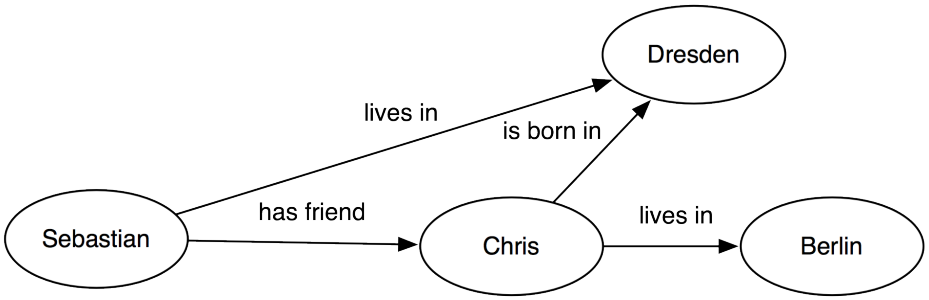

Figure 2.1: Triples represent a directed graph

However, the above example is not entirely correct, as the subjects of the graph are *resources*, and as such they have to be identified by an URI. Thus, we will assign a unique URI to each resource, as defined by the following mapping:

```
Sebastian: http://sebastian.kurfuerst.eu/
Chris: http://purl.org/325532
Dresden: http://sws.geonames.org/2935020/
Berlin: http://sws.geonames.org/2950159/
```

For example, the node `Sebastian` from above will be represented by `http://sebastian.kurfuerst.eu/` which is the URI for me personally. Still, it is a coincidence that my name is somehow encoded in the URI, as the URI itself is an artifical identity without any meaning. This can be seen in the URI for `Chris` or `Dresden`, which is just an arbitary (but fixed) unique URI with an artificial identifier in it. Thus, in order to encode the information stored in the above graph, we also need to explicitely add the names of the people and the cities to the graph.

Using Compact URIs

As URIs are very verbose to write, we use a short form called *Compact URIs (CURIEs)* as defined by [curie10]. CURIEs have the form `[prefix:suffix]`. They can be *expanded* to an URI by looking up the prefix in a prefix map, and then appending the suffix.
A short example will demonstrate that: If we have a CURIE `[geonames:2935020/]` and a prefix mapping `geonames: http://sws.geonames.org/`, then the above CURIE can be expanded to the URI `http://sws.geonames.org/2935020/`. The prefix mapping used in this work can be found in Appendix A.

The new graph, which is now valid RDF, can be seen in Figure 2.2 respectively Listing 2.

```
purl:325532 foaf:based_near geonames:2035020/.
purl:325532 cv:birthPlace geonames:2950159/.
<http://sebastian.kurfuerst.eu/> foaf:knows purl:325532.
<http://sebastian.kurfuerst.eu/> foaf:based_near genames:2935020.
<http://sebastian.kurfuerst.eu/> foaf:firstname "Sebastian".
purl:325532 foaf:firstname "Chris".
geonames:2935020/ geonames:name "Dresden".
geonames:2950159/ geonames:name "Berlin".
```

Listing 2: Information represented as RDF

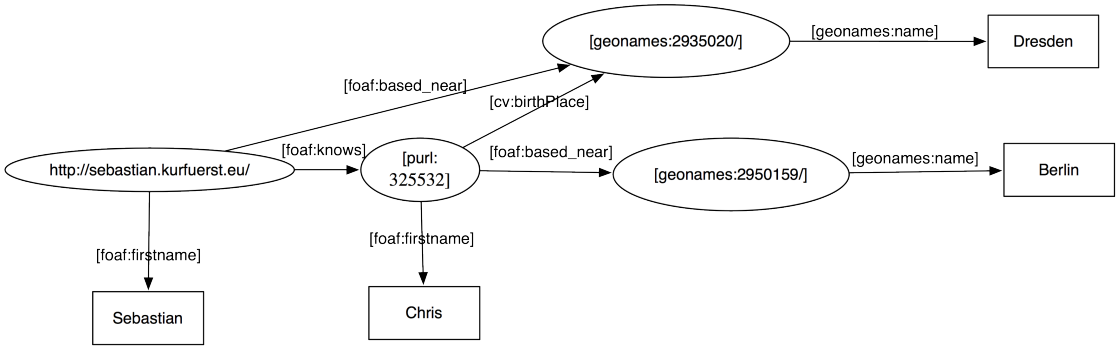

Figure 2.2: Representing information as RDF graph

We see that there are two types of nodes in the graph: First, there are *literals*, which contain simple *values* such as strings, integers, or dates. They are denoted by rectangles in our visual graph represetation. Second, there are *Named Nodes*, which are resources, being identified by an URI.

The notation introduced in the last example is called N3[2], and is an easy to read text-based format for the shown graph. We will use this notation throughout this work to represent RDF graphs in a textual notation.

Now, we have seen how information could be modelled using RDF, by putting them into a graph, and storing this graph as RDF.

2.1.3 Defining the Meaning of Information with Schemata

The example in Listing 2 quietly introduced URIs for the *predicate* of each triple as well – which we will now explain. If we have a triple like `<Sebastian> <has friend> <Chris>`, we have already explained that we need to introduce artificial identifiers for the subject and the object of this triple. Additionally, we need to define what `<has friend>` actually means. So, we need to define the *language* being used for relations between resources.

Such definitions occur in a so-called *schema*, which specifies the meaning of a certain relation. In our example, we use `foaf:knows`, and the according *Friend of a Friend (FOAF)* schema explains how `foaf:knows` should be used, and how its meaning should be understood (excerpt):

> The *knows* property relates a `Person` to another `Person` that he or she knows.

[2] http://www.w3.org/DesignIssues/Notation3

> We take a broad view of 'knows', but do require some form of reciprocated interaction (ie. stalkers need not apply). Since social attitudes and conventions on this topic vary greatly between communities, counties and cultures, it is not appropriate for FOAF to be overly-specific here.
>
> If someone knows a person, it would be usual for the relation to be reciprocated. However this doesn't mean that there is any obligation for either party to publish FOAF describing this relationship. A *knows* relationship does not imply friendship, endorsement, or that a face-to-face meeting has taken place: phone, fax, email, and smoke signals are all perfectly acceptable ways of communicating with people you know.
>
> —FOAF Specification 0.98

Thus, by using `foaf:knows` instead of `has friend`, we have explicitely stated what "knowing another person" actually means for us. When many people publishing RDF documents express such a common understanding (i.e. by using the `foaf:knows` to express relationship to other people), we can automatically answer questions of who knows whom. Thus, schemata are a way to define a *common language*, and provide most benefits if they are widely re-used across the Semantic Web.

Besides definitions for properties, schemata can also contain certain *classes* of resources. In our example, the `foaf` schema defines a `foaf:Person`:

> The `Person` class represents people. Something is a `Person` if it is a person. We don't nitpic about whether they're alive, dead, real, or imaginary.
>
> The `Person` class is a sub-class of the `Agent` class, since all people are considered 'agents' in FOAF.
>
> —FOAF Specification 0.98

If we agree on this definition for a person, we should state that `<http://sebastian.kurfuerst.eu/>` is of type `foaf:Person`. This can be done using the predicate `rdf:type`, so we can correctly formulate the above statement as triple:

```
<http://sebastian.kurfuerst.eu/> rdf:type foaf:Person.
```

The schemata themselves are usually also specified in RDF and are thus machine-readable. Thus, the more complete example graph, with some of its schema relations can be seen in Figure 2.3.

First, we see that all persons are of type `foaf:Person`, and the places are of type `geonames:Feature`. Furthermore, the schema specifies that both `foaf:Person` and `geonames:Feature` are a subclass of `geo:SpatialThing`.

Furthermore, the definition of `foaf:knows` specifies the type of subjects on which this property can be applied, using `rdfs:Domain`. It also specifies the type of the object which is allowed as target with `rdfs:Range`. In the above example, this means that for a sentence `<Subject> foaf:knows <Object>`, both Subject and Object must be of type `foaf:Person`.

We have also shown the definition of `foaf:based_near`, which relates two `geo:SpatialThings` with each other. As both `geonames:Feature` and `foaf:Person` are subclasses of `geo:SpatialThing`, it is possible to relate a `foaf:Person` and a `geonames:Feature` together with `foaf:based_near`. This example illustrates that it is possible to mix and extend schemata as needed.

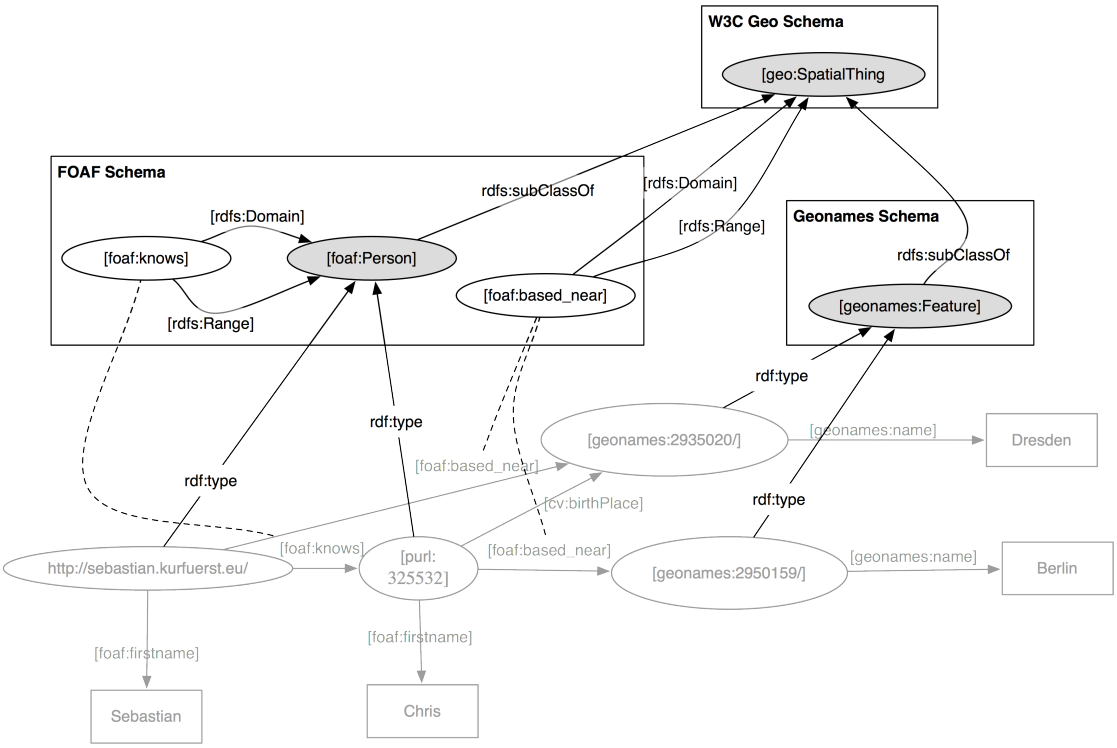

Figure 2.3: Graph with schema relations

The schemata themselves are written in RDF as well, using the vocabularies of *RDF Schema (RDFS)* and *OWL*. RDFs is a simple vocabulary, used above, which can be used to express inheritance and basic schema properties. OWL comes in three variants with different levels of expressiveness, the most widely used one being *OWL-DL*, which matches SHOIN(D) Description Logic in terms of expressiveness.

For this work, an in-depth understanding of schemata and ontologies is not needed. However, understanding the general concepts and ideas is very helpful for the remainder of this work, as we will extract information from schemata at various places. At last, it should be mentioned that a lot of research focusses exclusively on schemata and ontologies.

Relation to Description Logics

OWL-DL is matching Description Logics in terms of expressiveness. The TBox in Description Logics roughly matches the schemata and ontologies in RDF, and the ABox contains the instance data.
In the RDF graph, there is no formal distinction between the TBox and ABox, but it can be segmented into these two parts.

2.1.4 The Open World Assumption

RDF is built on a very important assumption: the *open world assumption*. It states that if a triple is *not found* in the set of all triples, it *cannot* be inferred that this triple does not hold – it is unknown whether the triple holds or not.

This is used to encode that we only ever have *partial knowledge* in the information system, and are not able to fully describe the world.

This assumption, while it looks abstract at first, has far-reaching practical consequences. For instance, it is not possible to express that an object *is only allowed* to have a certain set of relations, *and no more*. If a statement was `true` under a set of triples T, it is also `true` under a set of triples T', where $T \subseteq T'$.

This is one of the key differences to other information representation formats like XML: In XML Schema, everything which is not allowed explicitely is forbidden, and thus it is not possible to extend an arbitary XML Schema. Thus, XML Schema has a more closed-world approach, as every element which is not explicitely allowed is forbidden.

2.1.5 The Linked Data Cloud

Before, it has been stated that resource URIs are just used as artificial identifier without any meaning. Still, it is best practice in Linked Data to make all URIs *dereferenceable* using HTTP, so that a suitable description of the data can be found at the location described by the URI. This means when publishing information, one should not generate random URIs, but the hostname should be under control of the owner of the data. When following this best practice, the resource URIs implicitely contain information about *who is responsible for the given data* (also called *provenance*), as the URI contains a hostname which is resolved using DNS. Thus, when exposing information as Linked Data, make sure to use a domain name which you control as the base namespace for resource URIs.

When all URIs in the Linked Data graph are dereferenceable, additional information about certain resources can be automatically fetched. As an example, the object in the triple `<http://sebastian.kurfuerst.eu/> foaf:interest <http://dbpedia.org/resource/TYPO3>` can be automatically dereferenced to fetch additional information about TYPO3.

Because of this characteristics, the Linked Data ecosystem is also called *Giant Global Graph (GGG)*, as each triple can be seen as part of a gigantic graph containing all information of the Semantic Web. Figure 1.1 visualizes this in a nice way, showing the cross-references between datasets.

It is also important to remember that schemata are also a special form of resource, so URIs used in schemata should also be dereferenceable to fetch the schema in a machine-readable way.

The best practices of publishing data are known as the *Linked Data Principles*[3], which should be followed for all data publishers:

1. Use URIs as names for things.
2. Use HTTP URIs, so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs, so that they can discover more things.

Especially the third point needs some further explanation. According to [heath11], two different URIs need to be distinguished for a given entity. First, there needs to be an URI for the *real-world*

[3] http://www.w3.org/DesignIssues/LinkedData

object, and then there needs to be another URI for the *document which describes this real-world object*. By separating this, we can separate between statements for a document and a real-world object. As an example, the creation date of a book can be different than the creation date of a document about this book.

How is this implemented in practice? When dereferencing an URI for a real-world object, the system should return a `303 See Other` HTTP redirect, pointing to the *document* containing information about the real-world object. Depending on the `Accept` headers of the client, redirections to different data formats, like HTML or RDF, can be done.

Alternatively, one can also use hash URIs to implement the distinction between real-world objects and documents describing them. Thus, we can add a fifth Linked Data guideline:

5. Use `303` redirects or hash URIs for distinguishing between real-world objects and documents describing them.

Furthermore, several RDF features should be *avoided* in Linked Data context, according to [heath11], as they did not reach widespread adoption yet and make it too difficult for clients to receive data:

6. Avoid RDF reification. RDF reification expresses a `<Subject> <Predicate> <Object>` triple as the following:

   ```
   _:xxx rdf:type rdf:Statement.
   _:xxx rdf:subject <Subject>.
   _:xxx rdf:predicate <Predicate>.
   _:xxx rdf:object <Object>.
   ```

   This makes it possible to add additional metadata to a triple. However, as this notation adds another indirection level and is not adopted in a wide-spread manner, it should be avoided in a Linked Data context. Instead of using reification, rule 5. should be implemented.

7. Avoid *RDF Collections* and *RDF Containers* as long as the order of items does not matter, as this makes SPARQL queries a lot more difficult. Instead, multiple triples with the same predicate should be added.

8. Avoid using blank nodes as much as possible, as they cannot be referenced outside of a given document.

2.1.6 Storing RDF Data

In order to effectively process and query RDF data, we need some kind of database for triples. Such databases are called *triple stores*, being able to store several billions of triples.

Examples for Open-Source triple stores include Virtuoso[4], BigData[5] and 4Store[6]. Usually, they can be filled and queried via web services.

Many triple stores store one additional piece of information per triple[7]: The *source document*. This is especially helpful for updating triples in the store, as the stores usually provide an all-or-nothing semantics for a given document: When you update a document, all triples in the store

[4] http://virtuoso.openlinksw.com/
[5] http://www.bigdata.com/
[6] http://4store.org
[7] That's why they are also referred to as *Quad Stores*.

for this document are removed, and then all sent triples are added again — and all this happens atomically.

While the source document is usually not needed for querying, it is very helpful for updating triples in our scenario, as we will see in Chapter 4.1.5.

2.1.7 Querying RDF Data with SPARQL

Now we are able to model arbitrary information as RDF graph, and by following the principles of Linked Data, we can dereference the entity URIs to gradually traverse the Linked Data Graph.

However, for many use-cases, we are not so much interested in traversing the graph, but rather would like to *extract information* from the Linked Data Graph. For example, we might want to ask a question such as *Find all Persons living in Dresden*, or *Show me all events starting after 6 PM this evening in Berlin*.

For these queries, there is a defined *query language* called *SPARQL*. With this language, it is possible to specify certain graph fragments to be searched for, including placeholders. Thus, to find all persons located in Dresden, one could use the query shown in Listing 3. To list all triples available in the system, the query from Listing 4 can be used.

```
SELECT ?person
WHERE {
  ?person rdf:type foaf:Person.
  ?person foaf:based_near geonames:2935020/.
}
```

Listing 3: A SPARQL query which returns all persons located in Dresden

```
SELECT ?s ?p ?o
WHERE {
  ?s ?p ?o.
}
```

Listing 4: A SPARQL query which returns all triples

A SPARQL query must be sent to a *SPARQL endpoint*, which will evaluate the query and run the results. Almost all triple stores implement a SPARQL endpoint.

The use of SPARQL is especially helpful for data-aggregation scenarios. If data has to be aggregated across many subsystems, it is a common pattern to store all data in a central triple store, and then run SPARQL queries for aggregating information.

2.1.8 Embedding RDF into Web Pages with RDFa

So far, we have presented RDF as universal information description format, which, however, does not have any ties to the classical document-based web. Now, we will introduce a standard called *RDF Annotations (RDFa)*, which bridges the gap between the document-based internet and the Semantic Web.

RDFa allows to embed RDF triples directly inside an XHTML document (or any other XML document), and it thus allows the following interaction pattern:

> A user is browsing a classical, document-based event calendar web site. As this site uses RDFa to highlight places and events, the browser can ask the user questions such as "Do you want to see more events from the same City?" – and use a semantic search engine to search for other events not available on the particular website.

It will still take some time until the above scenario gets implementable, but there are first browser extensions available[8] which highlight semantic information in a HTML page, and enable the user to dive into the Semantic Web.

So, how can one embed RDFa into HTML? We will take an example of a blog posting, and embed RDF information into it. In Listing 5, the HTML structure of the non-enriched blog post can be found, while Listing 6 shows the RDFa version of the snippet.

```
<article>
    <h2>Linked Data to rule the world</h2>
    <h3>by Sebastian Kurfuerst</h3>
    <div>
        Lorem Ipsum... here comes the content of this blog post!
    </div>
</article>
```

Listing 5: The HTML structure which shall be enriched using RDFa

```
<article about="http://my.blog.kurfuerst.eu/2011/07/linked-data"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:sioc="http://rdfs.org/sioc/ns#">
    <h2 property="dc:title">Linked Data to rule the world</h2>
    <h3 property="dc:creator" content="http://sebastian.kurfuerst.eu/">
        by Sebastian Kurfuerst
    </h3>
    <div property="sioc:content">
        Lorem Ipsum... here comes the content of this blog post!
    </div>
</article>
```

Listing 6: RDFa is embedded into the HTML structure

RDFa introduces some universal tag attributes, the most important ones being listed below:

- `about` is used to speficy the *subject* of the RDF triple.
- `property` specifies the *predicate* as CURIE. The prefix mapping is defined using a standard XML namespace declaration with `xmlns`.
- The `content` attribute specifies the *object* of the triple. If no `content` attribute is given, the inner HTML of the tag is used as content.

Thus, the example from Listing 6 yields the triples which are shown in Listing 7.

[8] https://addons.mozilla.org/de/firefox/addon/operator/, http://rdfa.digitalbazaar.com/fuzz/trac/

```
<http://my.blog.kurfuerst.eu/2011/07/linked-data>
  dc:title "Linked Data to rule the world".
<http://my.blog.kurfuerst.eu/2011/07/linked-data>
  dc:creator <http://sebastian.kurfuerst.eu/>.
<http://my.blog.kurfuerst.eu/2011/07/linked-data>
  sioc:content "Lorem Ipsum ..... post!".
```

Listing 7: Triples from Listing 6

Besides yielding these triples, RDFa can contain even more information, as it uses the nesting of HTML elements to express meaning: If a tag does not have the `about` property set, then the `about` property of its nearest ancestor is used. In the example above, this means that the `<h2>` tag automatically uses the `about` property of the `<article>` tag as subject of the RDF triple.

Thus, it is also possible to analyze RDFa from a *structural view*: An RDFa-capable browser could for example highlight the *whole article* when the user selects a single element from it, as it knows that the whole `<article>` tag has a certain RDF subject.

To sum it up, there are two requirements for RDFa generation:

1. *Validity:* The RDF triples generated from the RDFa must be semantically valid. This means they should be consistent with the RDF triples generated at other points of the application.
2. *Structural View:* If some DOM nodes with RDFa annotations have the same subject, it should be checked if the subject RDFa annotation could be moved into a common ancestor of these DOM nodes. This could give a hint to an RDFa-aware browser about the relation between DOM structure and semantical structure of a HTML document.

In practice, requirement 1 is relevant as of today, as semantic search engines are extracting RDF triples from RDFa. As there are no browsers which analyze the structural RDFa information on a page, requirement 2 is not relevant for practical implementations today.

2.1.9 Conclusion

You now have a good understanding of the core concepts behind Linked Data. After focussing on how to model information, the relevant standards have been introduced to read, store and process RDF.

In the remainder of this chapter, another paradigm for modelling the world in software is introduced: Domain-Driven Design. This work then proceeds on connecting Domain-Driven Design and Linked Data together.

2.2 Domain-Driven Design

Domain-Driven Design (DDD), as introduced in [evans04], is a development technique which focuses on understanding the customer's problem domain[9]. It not only contains a set of technical ideas, but it also consists of techniques to structure the creativity in the development process.

[9] The complete section about Domain-Driven Design has been translated and adapted from [rau10], as that book has been written by the same author as this Diploma Thesis.

The key of Domain-Driven Design is understanding the customers needs, and also the environment in which the customer works. The problem which the to-be-written program should solve is called the *problem domain*, and in Domain-Driven Design, development is guided by the exploration of the problem domain.

While talking to the customer to understand his needs and wishes, the developer creates a model which reflects the current understanding of the problem. This model is called *Domain Model* because it should accurately reflect the problem domain of the customer. Then, the Domain Model is tested with real use-cases, trying to understand if it fits to the customer's processes and way of working. Then, the model is refined again – and the whole process of discussion with the customer starts again. Thus, Domain-Driven Design is an iterative approach to software development.

Still, Domain-Driven Design is very pragmatic, as code is created very early on (instead of extensive requirements specifications); and real-world problems thus occur very early in the development process, where they can be easily corrected. Normally, it takes some iterations of model refinement until a Domain Model adequately reflects the problem domain, focussing on the important properties, and leaving out unimportant ones.

In the following sections, some core components of Domain-Driven Design are explained. It starts with an approach to create an ubiquitous language, and then focuses on the technical realization of the domain model. After that, it is quickly explained how FLOW3 enables Domain-Driven Design, such that the reader gets a more practical understanding of it.

We do not explain all details of Domain-Driven Design in this work, as only parts of it are important for the general understanding needed for this work.

2.2.1 Creating an Ubiquitous Language

In a typical enterprise software project, a multitude of different roles are involved: For instance, the customer is an expert in his business, and he wants to use software to solve a certain problem for him. Thus, he has a very clear idea on the interactions of the to-be-created software with the environment, and he is one of the people who needs to use the software on a daily basis lateron. Because he has much knowledge about how the software is used, we call him the *Domain Expert*.

On the other hand, there are the developers who actually need to implement the software. While they are very skilled in applying certain technologies, they often are no experts in the problem domain. Now, developers and domain experts speak a very different language, and misconceptions happen very often.

To reduce miscommunication, an *ubiquitous language* should be formed, in which key terms of the problem domain are described in a language understandable to both the domain expert and the developer. Thus, the developers learn to use the correct language of the problem domain right from the beginning, and can express themselves in a better way when discussing with the domain expert. Furthermore, they should also use the ubiquitous language throughout all parts of the project: Not only in communication, design documents and documentation, but the key terms should also appear in the Domain Model. Names of classes, methods and properties are also part of the ubiquitous language.

By using the language of the domain expert also in the code, it is possible to discuss about difficult-to-specify functionality by looking at the code together with the domain expert. This is especially helpful for complex calculations or condition rules. Thus, the domain expert can decide whether the business logic was correctly implemented.

Creating a ubiquitous language involves creating a glossary, in which the key terms are explained in a way both understandable to the domain expert and the developer. This glossary is also updated throughout the project, to reflect new insights gained in the development process.

2.2.2 Modelling the domain

Now, while discussing the problem with the domain expert, the developer starts to create the Domain Model, and refines it step by step. Usually, UML is employed for that, as it is a succinct way of expressing the relevant information of the problem domain.

The Domain Model consists of objects (as DDD is a technique for object-oriented languages), the so-called *Domain Objects*.

There are two types of domain objects, called *entities* and *value objects*. If a domain object has a certain *identity* which stays the same as the objects changes its state, the object is an *entity*. Otherwise, if the identity of an object is built from *all properties*, it is a *value object*. We will now explain these two types of objects in detail, including practical use-cases.

Furthermore, association mapping is explained, and aggregates are introduced as a way to further structure the code.

Entities

Entities have a unique identity, which stays the same despite of changes in the properties of the object. For example, a user can have a user name as identity, a student a student's ID. Although properties of the objects can change over time (for example the student changes his courses), it is still the same object. Thus, the above examples are *entities*.

The identity of an object is given by an immutable property or a combination of them. In some use-cases it can make a lot of sense to define identity properties in a way which is *meaningful in the domain context*: If building an application which interfaces with a package tracking system, the tracking ID of a package should be used as identity inside the system. Doing so will reduce the risk of inconsistent data, and can also speed up access.

For some domain objects like a `Person`, it is highly dependent on the problem domain what should be used as identity property. In an internet forum, the e-mail address is often used as identity property for people, while when implementing an e-government application, one might use the passport ID to uniquely identify citizens (which nobody would use in the web forum because its data is too sensible).

In case the developer does not specify an identity property, the framework assigns a universally unique identifier (UUID) to the object at creation time.

It is important to stress that identity properties need to be set *at object creation time*, i.e. inside the constructor of an object, and are not allowed to change throughout the whole object lifetime. As we will see later, the object will be referenced using its identity properties, and a change of an identity property would effectively wipe one object and create a new one without updating dependent objects, leaving the system in an inconsistent state.

In a typical system, many domain objects will be *entities*. However, for some use-cases, another type is a lot better suited: Value objects, which are explained in the next section.

Value Objects

PHP provides several value types which it supports internally: Integer, float, string, float and array. However, it is often the case that you need more complex types of values inside your domain. These are being represented using *value objects*.

The identity of a value object is defined by *all its properties*. Thus, two objects are equal if all properties are equal. For instance, in a painting program, the concept of *color* needs to be somewhere implemented. A color is only represented through its value, for instance using RGB notation. If two colors have the same RGB values, they are effectively similar and do not need to be distinguished further.

Value objects do not only contain data, they can potentially contain very much logic, for example for converting the color value to another color space like HSV or CMYK, even taking color profiles into account.

As all properties of a value object are part of its identity, they are not allowed to be changed after the object's creation. Thus, value objects are *immutable*. The only way to "change" a value object is to create a new one using the old one as basis. For example, there might be a method `mix` on the `Color` object, which takes another `Color` object and mixes both colors. Still, as the internal state is not allowed to change, the mix method will effectively return a new `Color` object containing the mixed color values.

As value objects have a very straightforward semantic definition (similar to the simple data types in many programming languages), they can easily be created, cloned or transferred to other subsystems or other computers. Furthermore, it is clearly communicated that such objects are simple *values*.

Internally, frameworks can optimize the use of value objects by re-using them whenever possible, which can greatly reduce the amount of memory needed for applications.

Entity or Value Object?

An object can not be ultimately categorized into either being an entity or a value object – it depends greatly on the use case. An example illustrates this: For many applications which need to store an *address*, this address is clearly a value object - all properties like street, number, or city contribute to the identity of the object, and the *address* is only used as container for these properties.

However, if implementing an application for a postal service which should optimize letter delivery, not only the address, but also the person delivering to this location should be stored. This name of the postman does not belong to the identity of the object, and can change over time – a clear sign of *address* being an entity in this case. So, generally it often depends on the use-case whether an object is an entity or value object.

People new to Domain-Driven Design often tend to overuse entities, as this is what people coming from a relational database background are used to.

Associations

Now, after explaining the two types of domain objects, we will look at a particularily important implementation area: Associations between objects.

Domain objects have relationships between them. In the domain language, these relations are expressed often as follows: A *consists of* B, C *has* D, E *processes* F, G *belongs to* H. These relations are called *associations* in the Domain Model.

In the real world, relationships are often inherently bidirectional, are only active for a certain time span, and can contain further information. However, when modelling these relationships as associations, it is important to simplify them as much as possible, encoding only the relevant information into the Domain Model.

Especially complex to implement are bidirectional many-to-many relations, as they can be traversed in both directions, and consist of two lists of objects which have to be kept in sync manually in most programming languages (such as Java or PHP).

Still, especially in the first iterations of refining the Domain Model, many-to-many relations are very common. The following questions can help to simplify them:

- Is the association relevant for the core functionality of the application? If it is only used in rare use cases and there is another way to receive the needed information, it is often better to drop the association altogether.
- For bidirectional associations, can they be converted to unidirectional associations, because there is a main traversal direction? Traversing the other direction is still possible by querying the underlying persistence system.
- Can the association be qualified more restrictively, for example by adding multiplicities on each side?

The more simple the association is, the more directly it can be mapped to code, and the more clear the intent is.

Aggregates

When building a complex Domain Model, it will contain a lot of classes, all being on the same hierarchy level. However, often it is the case that certain objects are parts of a bigger object. For example, when modelling a `Car` domain object for a car repair shop, it might make sense to also model the wheels and the engine. As they are a part of the car, this understanding should be also reflected in our model.

Such a part-whole relationship of closely related objects is called *aggregate*. An aggregate contains a root, the so-called *aggregate root*, which is responsible for the integrity of the child-objects. Furthermore, the whole aggregate has only one identity visible to the outside: The identity of the aggregate root object. Thus, objects outside of the aggregate are only allowed to persistently reference the aggregate root, and not one of the inner objects.

For the `Car` example this means that a `ServiceStation` object should not reference the engine directly, but instead reference the `Car` through its external identity. If it still needs access to the engine, it can retrieve it through the `Car` object.

These referencing rules effectively structure the Domain Model on a more fine-grained level, which reduces the complexity of the application.

2.2.3 Life Cycle of Objects

Objects in the real world have a certain life cycle. A car is built, then it changes during its lifetime, and in the end it is scrapped. In Domain-Driven Design, the life cycle of domain objects is very similar, as it is shown in Figure 2.4.

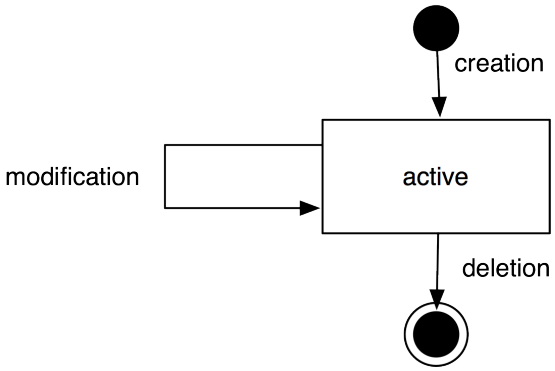

Figure 2.4: Simplified life cycle of objects

Because of performance reasons, it is not feasible to keep all objects in memory forever. Some kind of persistent storage, like a database, is needed. Objects which are not needed at the current point in time should be persistently stored, and only transformed into objects when needed. Thus, we need to expand the `active` state from Figure 2.4 to contain some more substates. These are shown below in Figure 2.5.

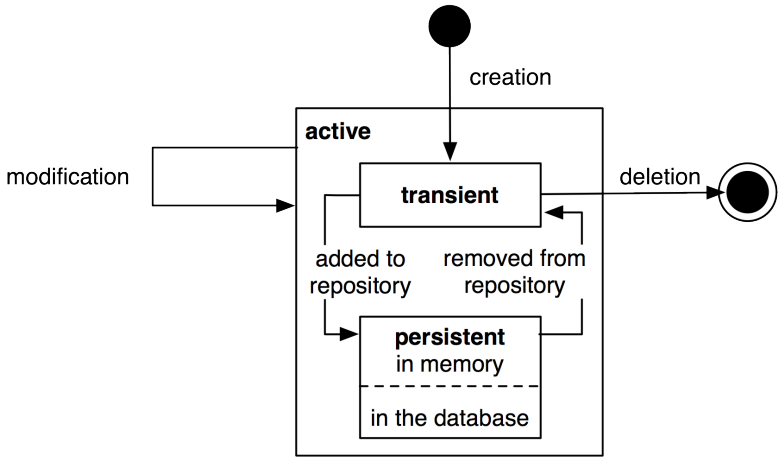

Figure 2.5: The real life cycle of objects

If an object is newly created, it is *transient*, so it is being deleted from memory at the end of the current request. If an object is needed permanently across requests, it needs to be transformed to a *persistent object*. This is the responsibility of *Repositories*, which allow to persistently store and retrieve domain objects.

So, if an object is *added* to a repository, this repository becomes responsible for saving the object. Furthermore, it is also responsible for persisting further changes to the object throughout its lifetime, automatically updating the database as needed.

For retrieving objects, repositories provide a query language. The repository automatically handles the database retrieval, and makes sure that each entity is only once in memory.

Despite the object being created and retrieved multiple times during its lifecycle, it logically continues to exist, even when it is stored in the database. It is only because of performance and safety reasons that is is not stored in main memory, but in a database. Thus, Domain-Driven Design distinguishes *creation* of an object from *reconstitution* from database: In the first case, the constructor is called, in the second case the constructor is not called as the object is only converted from another representation form.

In order to remove a persistent object, it needs to be removed from the repository responsible for it, and then at the end of the request, the object is transparently removed from the database.

For each *aggregate*, there is exactly one repository responsible which can be used to fetch the *aggregate root* object.

2.2.4 How FLOW3 Enables Domain-Driven Design

FLOW3 is a web development framework written in PHP, with Domain-Driven Design as its core principle. Now, we will show in which ways Domain-Driven Design is supported by FLOW3.

First, the developer can directly focus on creating the domain model, using unit testing to implement the use-cases needed. While he is creating the Domain Model, he can use plain PHP functionality, without caring about any particular framework. The PHP Domain Model he creates just consists of plain PHP objects, with no base class or other magic functionality involved. Thus, he can fully concentrate on modelling, without thinking about infrastructure yet.

This is a core principle of FLOW3: All parts of it strive for maximum focus and cleanness of the Domain Model, keeping the developer focused on the correct implementation of it.

Furthermore, the developer can use source code annotations to attach metadata to classes, methods or properties. This functionality can be used to mark objects as entity or value object, and to add validation rules to properties. In the domain object in Listing 8, a sample of such an annotated class is given. As PHP does not have a language construct for annotations, this is emulated by FLOW3 by parsing the source code comments.

In order to mark a domain object as *aggregate root*, only a repository has to be created for it, based on a certain naming convention. Repositories are the easiest way to make domain objects persistent, and FLOW3 provides a base class containing generic `findBy*` methods. Furthermore, it supports a domain-specific language for building queries which can be used for more complex queries, as shown in the `AccountRepository` from Listing 9.

Now, this is all the developer needs to do in order to persistently store domain objects. The database tables are created automatically, and all objects get a UUID assigned (as we did not specify an identity property).

From the infrastructure perspective, FLOW3 is structured as MVC framework, with the model being the Domain-Driven Design techniques. However, also in the controller and the view layer, the system has a strong support for domain objects: It can transparently convert objects to simple types, which can then be sent to the client's browser. It also works the other way around: Simple

```
/**
 * @entity
 */
class Account {

    /**
     * @var string
     */
    protected $firstName;

    /**
     * @var string
     */
    protected $lastName;

    /**
     * @var string
     * @validate EmailAddress
     */
    protected $email;

    //... getters and setters as well as other functions ...
}
```

Listing 8: A simple domain object being marked as entity, containing a validation for an email address

```
class AccountRepository extends \F3\FLOW3\Persistence\Repository {
    // by extending from the base repository, there is automatically
    // a findBy* method available for every property, i.e.
    // findByFirstName("Sebastian") will return all accounts with
    // the first name "Sebastian".
    public function findByName($firstName, $lastName) {
        $query = $this->createQuery();
        $query->matching(
            $query->logicalAnd(
                $query->equals('firstName', $firstName),
                $query->equals('lastName', $lastName)
            )
        );
        return $query->execute();
    }
}
```

Listing 9: A simple repository with a custom finder method

types will be converted to objects whenever possible, so the developer can deal with objects in an end-to-end fashion.

Furthermore, FLOW3 has an Aspect-Oriented Programming framework at its core, which makes it easy to separate cross-cutting concerns. There is a security framework in place (built upon AOP) where the developer can declaratively define access rules for his domain objects, and these are enforced automatically, without any checks needed in the controller or the model.

There are a lot more features to show, like rapid prototyping support, dependency injection, a signal-slots system and a custom-built template engine, but all these should only aid the developer in focussing on the problem domain and writing decoupled and extensible code.

2.3 Conclusion

In this chapter, the basic technologies this work is built upon are introduced. First, the core data structures of RDF are explained, showing how it can be used to represent arbitary information graphs. Furthermore, companion technologies and standards like RDFa and SPARQL are explained.

In the second part, an introduction to Domain-Driven Design is given, explaining its core principles. The chapter ends with a small outlook of how FLOW3 enables Domain-Driven Design.

As we have a basic understanding of Domain-Driven Design and Linked Data now, we will look at related work which can help us in exporting Domain Models to RDF.

CHAPTER

THREE

RELATED WORK

In this chapter, we will look at frameworks and solutions which have similarities with this work in certain aspects.

We will look at three different groups of applications, corresponding to the different research questions of this work. First, solutions for generating RDF from pre-existing data like databases or Java objects are shown. Second, frameworks which aid in generating RDFa are introduced. Third, Named Entity Recognition (NER) and Linkification research is presented.

3.1 Exporters to RDF

First, we cover solutions which export data from a proprietary, domain specific format to RDF. As an example, we will highlight two common conversion paths: From a relational database to RDF, and from Java classes to RDF.

3.1.1 Database to RDF: D2R

D2R[1] is a system which exports relational database contents to RDF. The system assigns URIs to the entries of the database and automatically generates a schema for them. Thus, it is possible to link from other Linked Data sources to the exported data, effectively making database contents available as Linked Data.

There is at least one alternative to D2R available which is called Triplify[2], which works in a similar manner. We introduce D2R here, as it is more actively maintained.

The mapping from database columns to RDF types is adjustable, making it possible to map legacy data to pre-existing, well-known schemata. Furthermore, the system provides a SPARQL endpoint for advanced queries.

The system does not contain any out-of-the-box linkification possibilities, i.e. it is not possible to map an `author` field with contents `Sebastian Kurfürst` to the URI `http://sebastian.kurfuerst.eu/` (which is not managed by D2R). This is understandable, as D2R is implemented as web service, and does not contain any kind of user interface for managing and checking such mappings.

[1] http://www4.wiwiss.fu-berlin.de/bizer/d2r-server/
[2] http://triplify.org/

Furthermore, D2R fundamentally works on the granularity level of database fiels, and can not parse a single field and generate multiple inter-related entities out of it. This would be especially helpful for converting longer texts which are stored in a single database field, enriching them with semantic data.

Generally, D2R works on a very low level of abstraction, directly exporting the database contents to RDF. Nowadays, many (if not most) web applications are built on top of object-relational mappers such as Hibernate[3] or Doctrine[4]. They often only use a subset of the features available in today's databases. Instead, they have a more expressive *Domain Model* at its core. Thus, D2R is not easily applicable to this kind of application. That is why we will now look at another solution which generates RDF from Java objects.

3.1.2 POJO to RDF Mapping: Empire

Empire[5] is a framework which maps Plain Old Java Objects (POJOs) to RDF. The developer annotates the corresponding class with annotations, and the system can then generate RDF triples out of a domain object. As an example, an annotated object looks as shown in Listing 10.

```
@Entity
@Namespaces({"foaf", "http://xmlns.com/foaf/0.1/"})
@RdfsClass("foaf:Person")
public class Person implements SupportsRdfId {
    @RdfProperty("foaf:firstName")
    private String firstName;

    @RdfProperty("foaf:surname")
    private String lastName;

    @RdfId
    @RdfProperty("foaf:mbox")
    private String email;

    @RdfProperty("foaf:homepage")
    private URI homepage;

    @RdfProperty("foaf:depiction")
    private Image depiction;

    @RdfProperty("foaf:knows")
    private Collection<Person> knows;
    // normal bean-style getters and setters...
}
```

Listing 10: POJO class anotated by Empire annotations

In the example, the `Person` class is mapped to `foaf:Person`, and each property of the class is mapped to the corresponding FOAF property.

[3] http://www.hibernate.org/
[4] http://www.doctrine-project.org/
[5] https://github.com/mhgrove/Empire

Furthermore, Empire implements the Java Persistence API (JPA), making it possible to store annotated objects such as the above in a quad store like 4store[6], and query them using SPARQL. This allows for working with objects, although the data is stored as RDF in a quad store.

The mapping between the class and the RDF schema can only be specified using annotations, and it is not possible to override this using configuration. This is a clear drawback, especially when third-party Java classes should be exported to RDF which shall not be directly modified by the developer.

Empire does not make sure the generated URIs are dereferenceable, and does not provide a defined workflow for exporting objects to RDF. Instead, it is a more low-level library which could be a building block for a high-level RDF handling framework.

In this work, the idea of defining the schema mapping through annotations has been reused. However, our RDF framework can also work with other schema sources except code annotations.

There are some alternatives to Empire which use the same concepts, namely Jenabean[7], RDFBean[8] and RDFReactor[9]. As they work based on the same principles as Empire, and only differ in minor implementation details, only Empire has been introduced as representative example.

3.2 RDFa Generators

In this section, frameworks which aid in creating RDFa are introduced. There are a lot of unmaintained and abandoned projects in this area, that is why we only show actively maintained frameworks.

3.2.1 Grails

Grails[10] is a web framework built on Java, Groovy, Spring and Hibernate – following a Domain-Driven Design approach. It is following many design decisions of Ruby on Rails, such as convention-over-configuration and focus on ease-of-use for developers.

For Grails, a plugin[11] for RDFa exists. With this plugin, the developer has to take the following steps for RDFa output:

1. Define a mapping from the Domain Model to RDF Schema in the model, as shown in Listing 11.
2. Use special tags inside the view to output RDFa, depicted in Listing 12. It manually has to be specified which model property should be used, using the `rel` attribute.
3. As the system automatically generates a resource URI of the form `http://my.domain/person/resource/[id]`, the developer needs to write a `PersonController` which implements a `resource` action. There, the RDF data would need to be generated manually, and returned to the client once this URI is dereferenced.

[6] http://4store.org/
[7] http://code.google.com/p/jenabean/
[8] http://source.mysema.com/display/rdfbean/RDFBean
[9] http://semanticweb.org/wiki/RDFReactor
[10] http://grails.org/
[11] http://www.grails.org/plugin/rdfa

```
class Person {
        static rdf = [
                '':'http://xmlns.com/foaf/0.1/Person',
                'name':'http://xmlns.com/foaf/0.1/name',
                'friends':'http://xmlns.com/foaf/0.1/knows'
        ]
        String name
        static hasMany = [friends:Person]
}
```

Listing 11: Grails model with RDF mapping definitions

```
<rdfa:ul about="${person}">
        <g:each var="friend" in="${person.friends}">
                <rdfa:li rel="friends" resource="${friend}"
                                property="name" />
        </g:each>
</rdfa:ul>
```

Listing 12: Grails template with RDFa tags

To sum it up, while the Grails plugin provides some help for generating RDFa, it cannot generate a full RDF representation of a model. Furthermore, generating RDFa still involves adjusting the templates; and knowledge about RDFa semantics is definitely needed.

The solution developed in this work is centered around a Domain Model to RDF mapping, as it is also done in this Grails RDFa plugin. However, we use the mapping information at many more places, fully leveraging the model annotations in all parts of the framework stack. Thus, in the solution developed for this work, the view layer does not need to be touched for implementing RDFa support.

3.2.2 Drupal

Drupal[12] is an open source content management system, being hugely popular around the world and deployed at many thousands of websites.

Its core data model uses a generic *node* data structure for representing any type of content. A node always has a *content type* which specifies the properties and data types of a node. As such, Drupal content types are some kind of *schema*.

For providing support for Semantic Web technologies, Drupal builds upon a content type to RDF schema mapping. Based on this mapping, it can generate *RDF Annotations (RDFa)* in the rendered pages. Furthermore, through plugins, it also provides a SPARQL endpoint, such that it is possible to query the exported data of a drupal instance from the outside.

While RDF support in Drupal is done in a very transparent manner, it is fully bound to Drupal's proprietary *node* data structure and content types.

Additionally, Drupal is not built in an object-oriented manner, instead using functions in the global namespace. This clearly does not reflect the state-of-the-art of web application development anymore and limits its extensibility.

[12] http://drupal.org/

Furthermore, as Drupal is a CMS and no generic web application framework, it does not provide state-of-the-art abstractions for arbitary web applications, such as a Model-View-Controller implementation.

However, of all analyzed frameworks in this work, Drupal is clearly the one with the most seamless and advanced support for Semantic Web standards and technologies.

3.2.3 Loomp

Loomp, as introduced in [heese2010], is a content creation and management system with a focus on semantic annotations. At the core of the system is an annotation user interface called *one click annotation* (OCA). It focusses on the casual user, who has no knowledge about Semantic Web technologies, and uses a user interface like a word processor for creating annotations.

Currently, Loomp and the OCA can be tested on their online demo[13], but there is no download or public source code repository available at the time of this writing. For this work, Loomp has been tested using the demo system, which might not support all features of the platform.

Annotating content using OCA is a mostly manual process, as it is shown in 3.1: While the user is editing the text, he can select the words he wants to annotate (Step 1), and then use a toolbar to select the ontology he wants to link the words to (Steps 2 and 3). After that, a popup is shown where the user selects the entity which should be linked to. This can be seen in Figure 3.2.

Generally, the OCA can be used to annotate arbitrary text as RDFa, but there is still quite a lot of manual work involved. Furthermore, the user interface is not as intuitive as the authors of Loomp claim it to be; but this could be due to the early prototype status of that work.

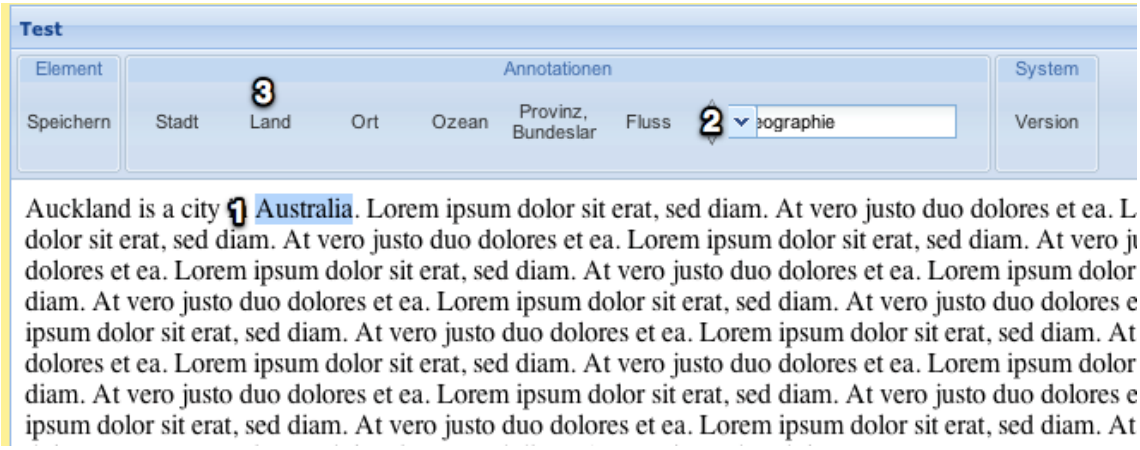

Figure 3.1: Loomp Annotation Workflow

When analyzing the generated RDFa after the annotation process, it seems that unique URIs are assigned to the tagged concepts (like `http://www.loomp.org/dic/pi/0.1/QCIOELD2KZNZDFK7P092R6CZCMV000KD`). However, these URIs are not dereferenceable using HTTP, which is against the established best practices of Linked Data publishing (as outlined in Chapter 2.1.5) and prevents automated spiders to use the information in the most efficient way.

[13] http://demo.loomp.org/oca

Figure 3.2: Selecting an annotation

In the publications about Loomp, it is explained that annotations can also be fetched from other Linked Data sources like DBPedia. However, in the demo instance, this functionality could not be found.

Furthermore, it is stated that web services like OpenCalais could be used to automate the annotation process, but this functionality was also not implemented in the demo system.

Clearly, there is some overlap between Loomp and this work: Both attempt to provide an end-to-end view for managing and annotating content using the principles of Linked Data, and both contain a user interface for annotation.

However, Loomp is mainly built around manual annotations, while the system developed in this work generates annotation suggestions automatically, providing a user interface for reviewing these.

As it is hard to judge on the current state of Loomp, and it is not downloadable, it could neither be extended in this work nor analyzed on a technical level.

3.2.4 RDFaCE

RDFaCE, introduced in [khalili2011], is a content annotation system which enables to add semantic annotations during the writing process. Is is built around a web-based rich text editor (TinyMCE), and extends it with RDFa annotation capabilities.

Besides making it possible to annotate a text manually, RDFaCE provides a supervised *automatic annotation service*: They extract entity types using NER web services such as OpenCalais, and then try to find good URIs for these entities using Sindice. In order to improve extraction performance, they are able to combine multiple NER components, only adding an annotation when a configurable number of NERs agree.

The system is purely implemented in JavaScript inside the browser, which makes it very easy to integrate with existing systems which already use a rich text editor. On the other hand, this makes it impossible to store persistent information on the server side. Thus, the system is not able to adapt to the user's choices, as it cannot learn from past decisions of the user.

3.3 Named Entity Recognition

Here, we will look at solutions which involve *Named Entity Recognition (NER)* — both web services and stand-alone components.

These services analyze continuous text, and detect entities like people, places or companies inside. Some also provide Linked Data URIs for the found entities.

3.3.1 OpenCalais

OpenCalais[14] is a NER web service provided by Thomson Reuters. One can send a human-readable text to the web service, which then finds entities in the text. Besides supporting common entity types like people, places and companies, the system has also support for other types like currencies, music albums or medical treatments. In total, the system supports about 115 different entity types.

Observation shows that OpenCalais has a high quality of recognition, leading to very few false positives. However, while it supports several languages, it does not support German for example.

Furthermore, OpenCalais tries to provide linkification for the found entities, linking them to DBpedia and other trustworthy data sources. However, it does not provide linkification for entities not being found in these trustworthy data sources. This means it would not find the URI `http://sebastian.kurfuerst.eu/` for the person `Sebastian Kurfürst`.

There are some competitors to OpenCalais, like *Alchemy API*[15], *Evri*[16] or *DBpedia Spotlight*[17]. All of these focus on entity type recognition and not linkification, and are built like OpenCalais.

3.3.2 Palladian

Palladian[18] is a framework for Internet Information Retrieval (IIR), developed at the University of Dresden. It hosts a collection of text processing algorithms, and reuses other projects whenever possible.

For this work, we use algorithms for Named Entity Recognition and Language Detection. As Palladian contains a unified API for OpenCalais, Alchemy API and other web services, we use this unified API to call backend NER and language detection services.

Furthermore, Palladian implements a custom Named Entity Recognition system, which is also able to be trained with *partial knowledge*. When a NER is trained, it receives sentences like the following:

> <Person>Sebastian Kurfürst</Person> is a student at TU Dresden.

From this sentence, a NER would learn that `Sebastian Kurfürst` is an instance of `Person`. Furthermore, a standard NERs will also learn that everything else is *not tagged*, i.e. it will learn that `student` and `TU Dresden` is neither a place, a person or an organization.

[14] http://www.opencalais.com/
[15] http://www.alchemyapi.com/
[16] http://www.evri.com/
[17] http://dbpedia.org/spotlight
[18] http://palladian.ws/

Palladian is different in this regard: It can be configured whether the negative information should be learned or not. This *partial learning* fits a lot better to the way the Semantic Web works, as it cannot be inferred that a particular statement does *not* hold when it is not given (as explained in Chapter 2.1.4).

Furthermore, end-users would not understand the implications of having something *not* tagged, as people tend to only tag the most relevant information, depending on the context. Thus, in order to provide a predictable user experience, it is important to not learn negative facts.

In this work, the NER API of Palladian is exposed through a web service, and can be used by user interface components.

3.3.3 Wikipedia Linkification

Some research focusses on cross-linking entities in arbitrary texts with Wikipedia articles ([csomai08], [milne2008], [kulkarni2009], [ferragina2010]. [hachey2011]). Because DBpedia has a deterministic mapping[19] from Wikipedia page titles to Resource URIs, this research can be directly used to generate references to DBpedia articles.

All these approaches analyze the link relations between Wikipedia pages, and the context in which the links appear. Because Wikipedia contains a very high number of real-world concepts and relevant links between them, this is a very promising approach for linking common knowledge. Furthermore, the above approaches use Wikipedia to disambigurate concepts with the same name depending on the context.

The research in the beginning, namely [csomai08], [milne2008], and [kulkarni2009], has focussed on adding relevant cross-references to longer texts such as articles and blogs. [milne2008] specially devised an algorithm which uses un-ambiguous terms to resolve ambiguous ones, greatly improving precision.

Recently, [ferragina2010] introduced a system which is also capable of linking short texts like twitter messages (which lacks a lot of context information, often lacking un-ambiguous terms), while keeping real-world performance requirements in mind. As a result, their publicly available service[20] can also be used for finding entities in texts and cross-linking them with Wikipedia articles / DBpedia resources. Right now, the service is not open source, but API access is granted for research purposes without any fee.

While it is possible to integrate Linkification based on Wikipedia in the framework developed for this work, it has not yet been done because we first wanted to focus on linking to arbitrary data sources in the Semantic Web, not tieing the system to Wikipedia. In the future, it is certainly desirable to integrate the above frameworks.

3.3.4 Apache Stanbol

Apache Stanbol[21] is not only a NER system, but defines itself as a *"software stack ... for semantic content management"*, available via REST services. It will provide technologies in the following areas:

[19] http://wiki.dbpedia.org/URIencoding
[20] http://tagme.di.unipi.it/
[21] http://incubator.apache.org/stanbol/

- Storing and searching semantic information
- Enhancing non-semantic texts with semantic information
- Reasoning over the stored semantic information
- Generate user interfaces for semantic interaction

Apache Stanbol is a very promising project, especially because they are approaching semantic content management with a *holistic* approach — providing high-level APIs and tools which closely work together.

Still, Apache Stanbol is not yet stable or production-ready; and it is difficult to judge from the outside when it will reach a state where it can be used in production. Right now, it has only very limited documentation and examples, not yet explaining how exactly Apache Stanbol will reach the goals outlined above.

Some parts of Apache Stanbol will be very similar to the *Semantifier* web service developed during this work. However, the Semantifier takes a much more pragmatic approach than Stanbol, reusing many pre-existing solutions and plugging them together in a lightweight manner using the *Grails* web application framework, whereas Stanbol is built on top of OSGI.

Apache Stanbol is definitely a project worth following regularily; and once the above points are solved, it might even be a replacement for the Semantifier developed in this work.

3.4 Web Application Frameworks

Apart from the frameworks listed in this chapter, the following web development frameworks have been investigated:

- Ruby on Rails[22]
- Symfony 2[23]
- Zend Framework[24]
- CakePHP[25]
- Yii Framework[26]

None of the above frameworks support Semantic Web technologies like RDF or RDFa in a stable way at the time of this writing.

[22] http://rubyonrails.org/
[23] http://symfony.com/
[24] http://framework.zend.com/
[25] http://cakephp.org/
[26] http://www.yiiframework.com/

3.5 Conclusion

We have been looking at three different types of applications in this chapter:

- RDF Exporters
- Generation of RDFa
- Named Entity Recognition and Linkification research

Each of the pre-existing applications in these areas has their unique strenghts and weaknesses. In this work, we attempt to combine the strengths of all of the above applications into one coherent and powerful Semantic Web framework.

Where possible, we re-use existing frameworks. Our work builds on OpenCalais, Sindice and Palladian as main components. When it was not possible to directly use a framework, we tried to cherry-pick their strenghts, as it was done with the Grails plugin (Domain Model to RDF mapping), Stanbol (Web Service API) and D2R (exporting all data as RDF, with easy setup).

CHAPTER

FOUR

CONCEPTS

In this chapter, we will explain the concepts developed for a full RDF and RDFa framework. This starts with the export of Domain Models to RDF, continuing with RDFa support, and ends with cross-linking arbitrary other Linked Data URIs.

4.1 Exporting Domain Models as RDF

In the first step, we want to expose all data of the system available for external consumption. While it would be possible to develop a proprietary API for it, we decided to use the standards of Linked Data (namely RDF) as the target format. As RDF is a widely accepted *universal information description language*, there exist tools and techniques for accessing this data in a flexible way.

Thus, by exporting potentially all information as RDF, it is possible for any application to consume the data we manage in the source application, and build new applications on top.

In the following sections, we will explain the conceptual details of the RDF generation.

4.1.1 Using Domain Models

We are focussing on web applications built around a *Domain Model*, that is, a class representation of the relevant business concepts and their interactions. All data is (logically) encapsulated in objects, and can be accessed using public properties or through method invocations on these objects.

Technically, however, many frameworks (including FLOW3) do not store these objects directly to disk, but instead use a relational database for persistent storage. To bridge the gap between the object-oriented world and the relational database world, *Object-Relational Mappers (ORM)* are used.

Still, while it would be possible to directly convert the database contents to RDF, we would lose much metadata which is available on the object level, such as the domain-specific data types, inheritance relations, complex validation rules or computed properties.

Because of these reasons, it is a natural choice to use objects as the basis for conversion.

Furthermore, as objects can have relations with each other, we can see the objects as *directed graph*. As RDF is also graph-based, it is possible to map the object graph to an RDF graph.

Basically, every object will become a node in the RDF graph, and each property will be an edge. If the target property is a simple type (like a `String`), then a new RDF literal will be created, as it can be seen in Figure 4.1.

Now, in order to generate the RDF graph from Figure 4.1, some kind of *mapping* is needed, which converts `title` to `dcterms:title` and `text` to `sioc:content`. Furthermore, it has to be specified that all objects of class `Article` should be represented as `sioc:Post`. This is explained in the next section.

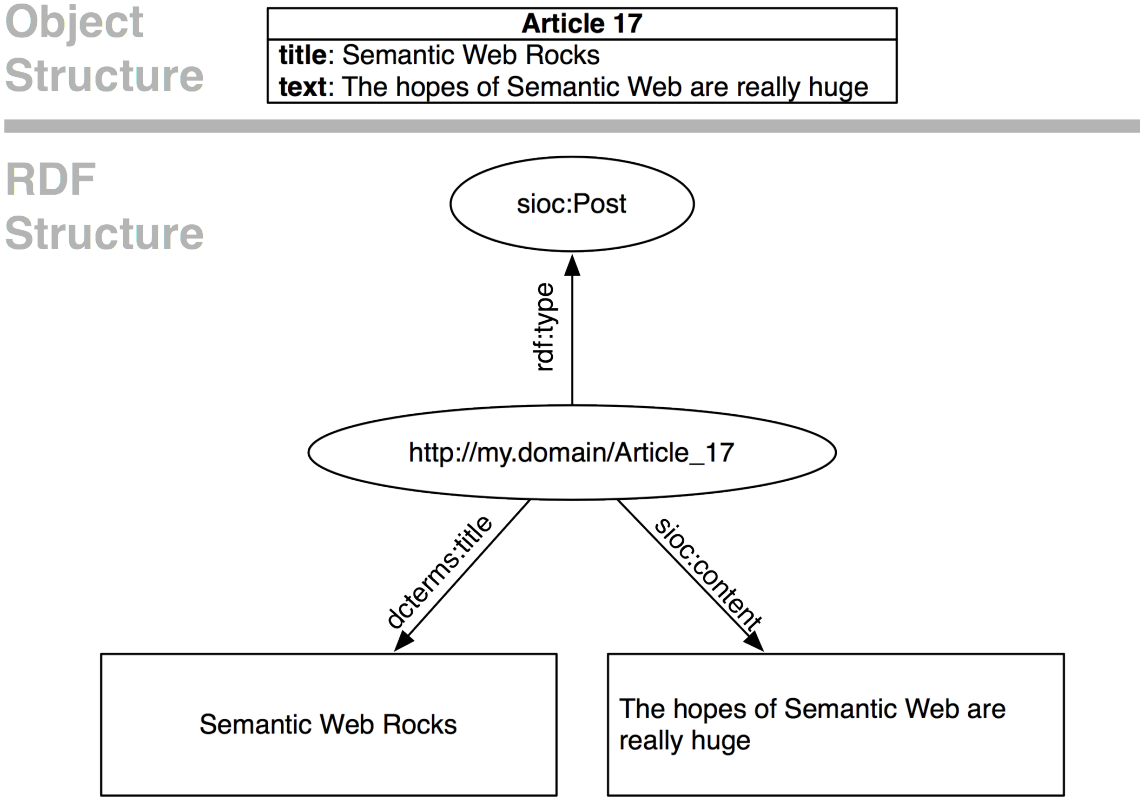

Figure 4.1: Mapping a single object to RDF is straight-forward. Object properties are converted to literals.

4.1.2 Schema Mapping

Every Domain Model is composed of some classes. For converting it to RDF, we are particularily interested in the structure of the model, which is expressed in a so-called *class schema*. This class schema contains all "meta-information", for example the class names, a list of properties for each class, and the type of each property.

We extend this schema to also contain information needed for the conversion to RDF. For the conversion process shown in Figure 4.1, we'd need the schema mapping from Listing 13.

Based on the `rdfType` which is specified in the mapping, the correct RDF can then be generated for an object.

```
Article
  rdfType sioc:Post
  -------------------------
  title
    var string
    rdfType dcterms:title
  text
    var string
    rdfType sioc:content
```

Listing 13: Schema mapping for the conversion shown in Figure 4.1

So far, we have only transformed a single object to RDF. Now, we will look at a case where an object has relations to other objects. We will extend the above example: An `Article` should allow references to related `Articles`, and it will have an `Author`. As an example, we will take the object structure depicted in Figure 4.2.

The needed schema mapping is shown in Listing 14.

```
1  Article
2    rdfType sioc:Post
3    -------------------------
4    title
5      var string
6      rdfType dcterms:title
7    text
8      var string
9      rdfType sioc:content
10   author
11     var Author
12     rdfType foaf:maker
13   related
14     var array<Article>
15     rdfType sioc:relatedTo
16
17 Author
18   rdfType foaf:Person
19   -------------------------
20   name
21     var string
22     rdfType foaf:name
```

Listing 14: Schema for mapping multiple objects with relations (Figure 4.2)

1:1 relationships to other objects are expressed in the Domain Model just using an association to another object. Because of that, the type of the `author` property is `Author` (line 11). As for all other properties which should be mapped to RDF, we also need to specify the name of the RDF predicate to be used, in this case `foaf:maker` (line 12).

We also see a 1:n relationship in the above example, as one `Article` can have multiple `related` articles. These related articles are just defined as a collection. Again, the `rdfType` annotation specifies the RDF predicate to be used (line 15).

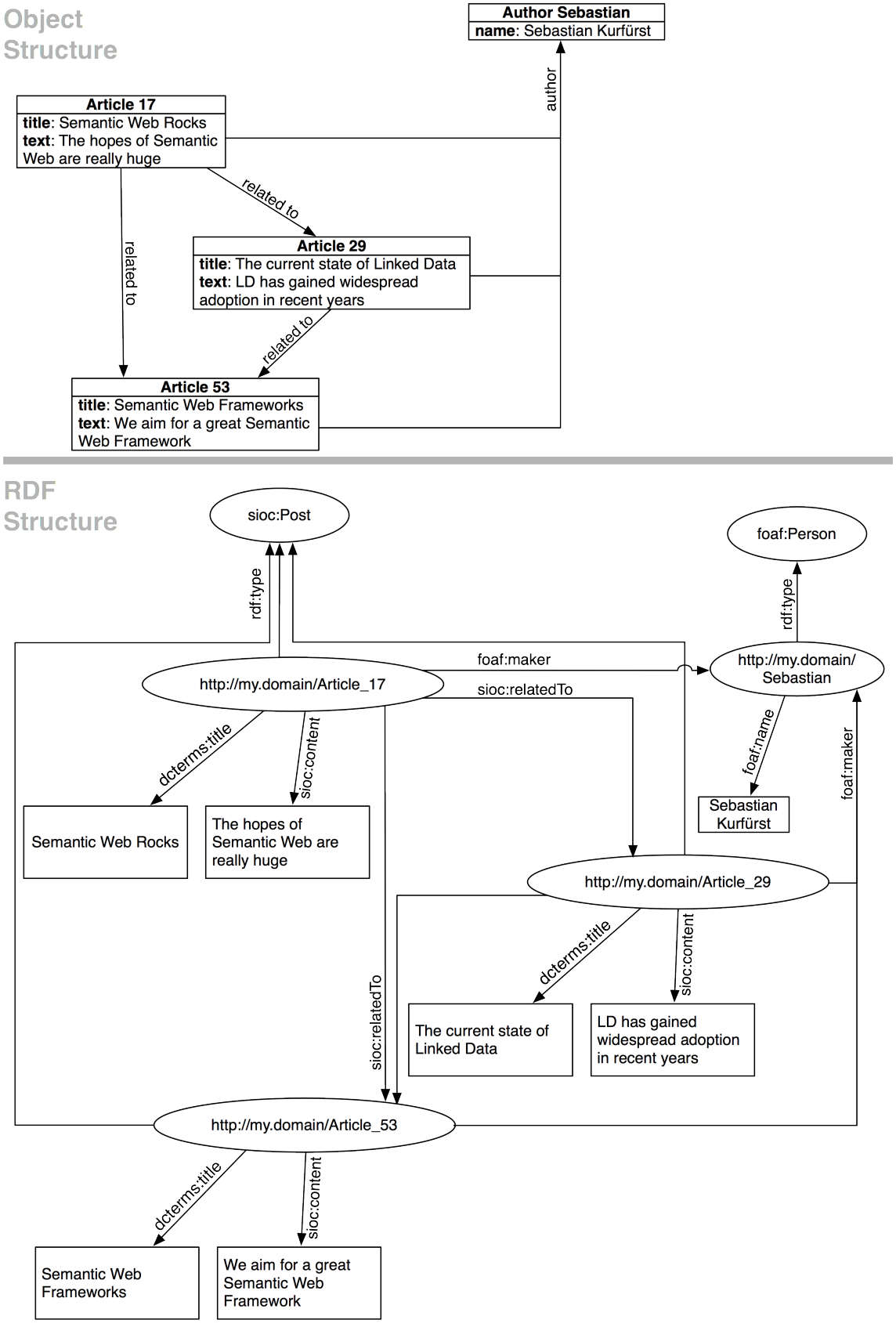

Figure 4.2: Mapping objects with relations is still straight-forward in principle, but the resulting graphs can become quite large.

For 1:n relationships, there are multiple possibilities of expressing this in RDF. We decide for the most straightforward solution: For each element of the relation, a triple is created which connects the relation source and the target object using the given predicate. In the example, this generates the triples in Listing 15.

Alternatively, when the order of the list matters, one would need to use *RDF Collections*[1], containers, or other ordered list ontologies[2]. Because they are difficult to query using SPARQL (see point 7. in Chapter 2.1.5), we did not implement them.

```
<http://my.domain/Article_17>
  sioc:relatedTo <http://my.domain/Article_29>.
<http://my.domain/Article_17>
  sioc:relatedTo <http://my.domain/Article_53>.
<http://my.domain/Article_29>
  sioc:relatedTo <http://my.domain/Article_53>.
```

Listing 15: m:n relationships are expressed by adding multiple triples with same subject/predicate

4.1.3 Defining a Resource Identity

So far, we have explained how the overall mapping between objects and RDF works, and illustrated this with an example. However, in the example, we also quietly generated *URIs* as identification for resources – for example `http://my.domain/Article_17`.

We will now explain the concept of generating these identifiers in detail.

Linked Data defines several best-practices for generating resource URIs, which should be met:

1. Resource URIs should not change during the lifetime of the resource.
2. All generated resource URIs must be *dereferenceable* using HTTP.
3. We need to distinguish between the URI of the *resource* and the URI of the *document where the resource is described*.

Although our Domain Objects do not necessarily have a public identifier, the frameworks which support Domain-Driven Design need to track object identity. That's why these frameworks generally assign a unique identifier or UUID for each object, which will never change during its lifetime.

Thus, by incorporating the object identifier in its resource URI, these will become immutable.

In this work, we are using resource URIs of the following form:

```
http://[domain]/rdf/id/[dataType]/[uuid]
```

Although the data type does not need to be included in the URI, it increases the readability for humans, and is the foundation for possible performance optimizations.

In order to fulfil best practice 2, the developed framework contains a HTTP endpoint, delivering the RDF representation of the current resource in NTriples format. To fulfil best-practice 3, the HTTP endpoint listens to two URIs:

[1] http://www.w3.org/TR/2004/REC-rdf-primer-20040210/#collections
[2] http://smiy.sourceforge.net/olo/spec/orderedlistontology.html

```
http://[domain]/rdf/id/[dataType]/[uuid]
http://[domain]/rdf/data/[dataType]/[uuid]
```

The first URI is used as URI of the *resource*. When accessing it, a HTTP `303 Redirect` to the second URI is sent back to the client.

The second URI identifies the *document* where information about the resource can be found, and it then returns the RDF representation of the resource in NTriples format.

4.1.4 Supporting Value Objects

So far, we are able to convert arbitrary objects with an identity to RDF. In this section, we want to focus specifically on the requirements for converting *Domain Objects* in the sense of *Domain-Driven Design* to RDF.

We need to deal specifically with the *value object* concept from Domain-Driven-Design, as this which will influence the mapping. A *value object* does not have an identity at all, but is identified by all its values.

As value objects cannot live on its own but are always attached to an entity, we export it when the entity is requested, and we represent the value object using a *blank RDF node*.

How does this look in practice? An example in Figure 4.3 shall illustrate this. We'll say that a `Person` object can have one or multiple `Address` objects attached. The `Person` is an entity, and the `Address` is just a value object, consisting of city, postcode and street.

In Figure 4.3, you see that there are two *blank nodes* inside, one for each address. When dereferencing `http://my.domain/Sebastian`, the triples for the `Person` as well as the triples for both `Address` objects are returned.

4.1.5 Sophisticated Queries Using SPARQL

So far, it is possible to export all information stored in domain objects as RDF, and it is possible to link to this information from the outside.

However, it is not yet possible to run more sophisticated queries, f.e. to answer questions such as *"Which articles are written by a particular author?"*, or *"What are the top 10 most commented articles?"*

The above questions can be formulated as SPARQL query. Thus, it would be very helpful to also allow SPARQL queries in our framework. There are several possibilities implementing that:

- Transliterate the SPARQL query to an SQL query, and run this one on the database.
- Use an SQL-based triple store which can also answer SPARQL queries.
- Use a store which is specialized in just storing RDF triples and answering SPARQL queries.

The first option has been discarded because due to the transliteration, it would be very hard to support querying properties which are computed in the Domain Model. Furthermore, we would then work on both the abstraction levels of the database *and* the Domain Model.

While the second option is usable in small projects, it does not scale as well as native triple stores.

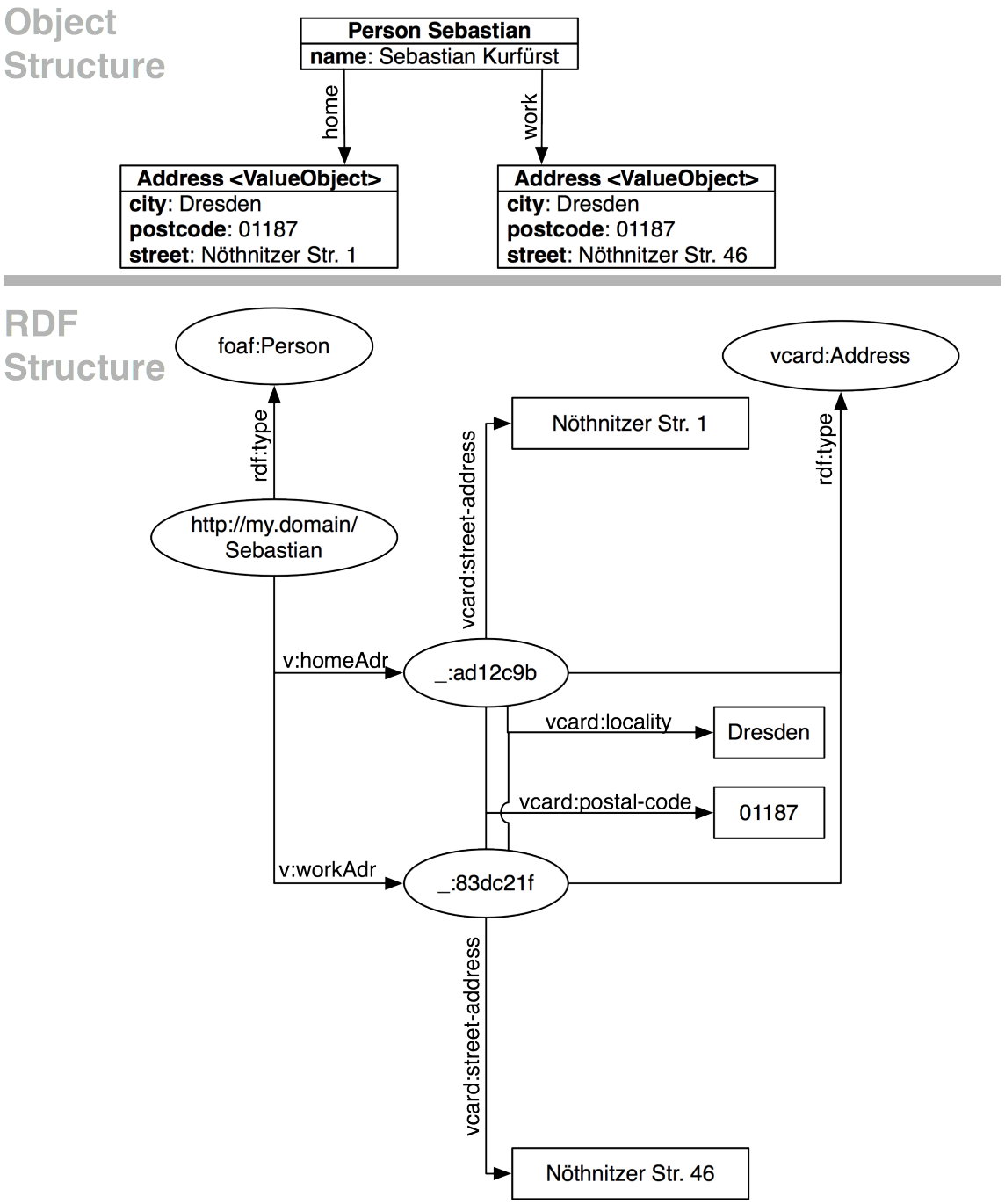

Figure 4.3: Value Objects are represented using RDF Blank Nodes

For this work, the last option has been implemented, because the native triple stores are generally faster and more scalable than a relational database solution.

Most triple stores have a very simple API to the outside: They provide one possibility to add / modify / delete triples, and one way to run SPARQL queries.

Thus, our framework needs to be able to detect changes in the Domain Model, and *push* the updated RDF triples to the store. Fortunately, many object/relational mappers provide notifications for added, modified and deleted objects; and also Doctrine (which is the O/R mapper used here) provides this.

We simply listen to model changes, and then regenerate the RDF of the model, which is then pushed to the triple store.

This works fine for adding new properties, but can impose problems for modifying and deleting them, as a simple example shall illustrate. In the initial situation, the fact *Sebastian knows Max* should be expressed. The object structure of the Domain Model is shown in Figure 4.4. The generated RDF is shown in Listing 16.

```
1 <http://sebastian.kurfuerst.eu/> foaf:knows <http://max-schulze.de/>.
```

Listing 16: Initial conversion result

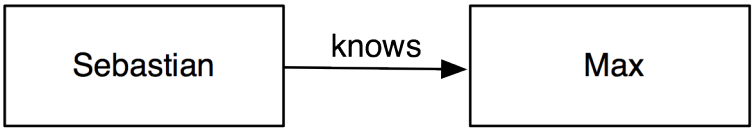

Figure 4.4: Initial runtime structure of the Domain Model

Now, imagine that Max and Sebastian have a big argument, and after that decide not to stay in contact. Instead, Sebastian finds a new friend John. He would then modify the Domain Model as follows to reflect the updated situation, such that it looks as in Figure 4.5. The RDF being generated from the updated Domain Model is displayed in Listing 17.

```
1 <http://sebastian.kurfuerst.eu/> foaf:knows <http://john-hess.de/>.
```

Listing 17: Updated Domain Model: Conversion result

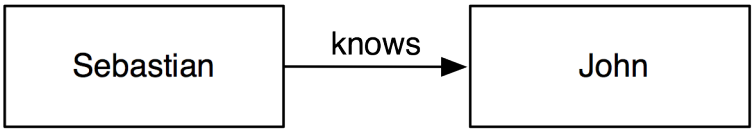

Figure 4.5: Updated runtime structure of the Domain Model

When the updated triple is sent to the store, it at once contains *both* triples (because we did not delete the old triple). Thus, queries asking for all friends of `Sebastian` would return both `Max` and `John`. As this is unexpected behavior, we need to avoid this case.

When the object-relational mapper notifies the framework that an object has changed, it often also transmits the initial state, or information which properties changed (including the old value). We could re-use this knowledge and *delete* the old triples explicitely from the triple store. Still, this would cause problems when *two applications* send the *same triple* to the store, as *both triples* would then be deleted. This is because triples are stored as *sets*.

Fortunately, most triple stores do not only store each *triple*, but also an additional field which contains the *provenance* of the triple. Because of this, they are also called *quad stores*. The additionally stored field is called *document*.

Using a quad store, solving the update problem is a lot easier, as a whole *document* with all its triples can be atomically updated. We only need to generate a stable URI by which the document can be identified.

In this work, we use the URIs of *entities* as the document URI.

4.2 RDFa Support

So far, all the RDF data is completely independent from the web application itself. There are no links from the web application to the RDF data; so it is not possible for a user who is interacting with the web application to detect that information is also available via RDF.

RDFa closes this gap between the document-based world and the RDF world, by providing a way to embed RDF snippets into HTML.

So, what does a web application developer have to implement in order to support RDFa? We aim for a solution which is as transparent and easy-to-use to the developer as possible: Because of the powerful schema mapping, the developer does not need to configure anything else for RDFa generation.

In this work, we are building a framework integrated with the Model View Controller architecture; so we need to modify the *view*. We especially support *template based* views, i.e. which are backed by a template engine.

A template engine usually has a very simple interface: There is some way of selecting which template to use, and then this template is *rendered* using a set of *bound variables*. During the rendering process, each pointer to a *bound variable* is replaced by the passed value.

In this work, we use the template notation of *Fluid*[3], which is a PHP based template engine. We introduce the relevant syntax using a short example in Listing 18.

```
1 <h1>Welcome {user.name}</h1>
2
3 Here is a list of your most recent articles:
4
5 <f:for each="{articles}" as="article">
6     - {article.title}
7 </f:for>
```

Listing 18: An exemplary Fluid Template

[3] http://forge.typo3.org/projects/show/package-typo3-fluid

- `{user.name}` (in line 1) is an *Object Accessor* and outputs a bound variable. By the dot-notation (`foo.bar`) it is possible to traverse the object graph.
- `<f:for>` (line 5) is a so-called *View Helper*, containing rendering logic. View Helpers can, amongst other things, modify the *bound variables*. The `<f:for>` ViewHelper iterates over a given collection of items and adds a new bound variable while rendering its contents.
- everything else is treated as plain text.

For supporting RDFa, the framework needs to intercept every object access, and wrap it with RDFa tags if necessary. For each intercepted *Object Accessor*, the following happens:

1. Extract the base object name and the property from the path. `{user.name}` has the base object name `user` and the property `name`.
2. Fetch the object from the bound variables. In our example, we fetch the `User` object.
3. Check if there exists a schema mapping for the given object and property. If not, do nothing.
4. If a schema mapping exists, generate the RDF triple for the property.
5. Output a wrapping `span` tag with the correct RDFa markup.

For Listing 18, applying the RDFa generation algorithm yields the HTML shown in Listing 19.

```
1  <h1>Welcome <span xmlns:foaf="[namespace import]"
2                    about="[URI for Sebastian]"
3                    property="foaf:name">
4                        Sebastian Kurfuerst</span>
5  </h1>
6
7  Here is a list of your most recent articles:
8
9      - <span xmlns:dcterms="[namespace import]" about="[URI for Article]"
10           property="dcterms:title">Semantic Web Rocks
11       </span>
12     - <span xmlns:dcterms="[namespace import]" about="[URI for Article]"
13           property="dcterms:title">The current state of Linked Data
14       </span>
```

Listing 19: Rendered Fluid Template with automatic RDFa generation applied

4.2.1 Adjusting RDFa output

To adjust the RDFa output, a developer needs to modify the schema mapping. Furthermore, there exists a View Helper to disable RDFa generation in portions of the page.

4.3 Linkification and Named Entity Recognition

So far, our framework can export arbitrary Domain Models as RDF, following many best practices of Linked Data publishing: The framework provides stable, dereferenceable URIs for resources, makes it possible to query them using SPARQL, and cross-references the objects from the existing web application using RDFa.

Essentially, we can expose *all information* stored in our system to the Linked Data Cloud, and other participants can link to our data, as Figure 4.6 shall illustrate.

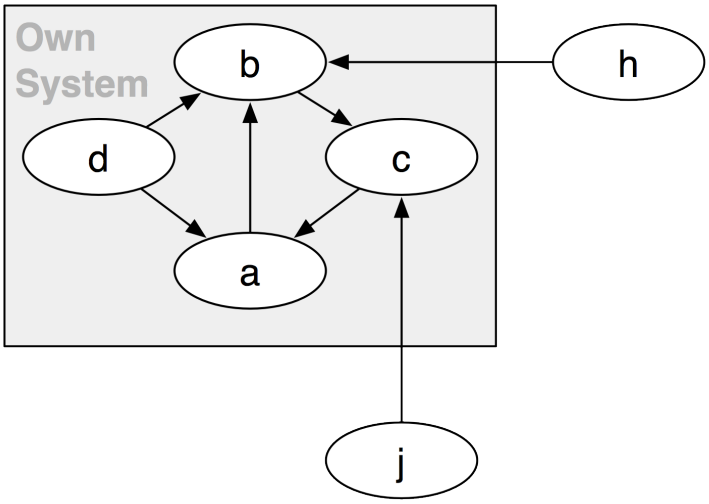

Figure 4.6: While our data can be referenced from other Linked Data sources, it is not yet possible to link our data to other Linked Data sources.

Other participants (`h` and `j`) can link to our data (`a`, `b`, `c`, `d`), but we do not support yet to link from *our data* to other data sets in the LOD cloud. That is a severe limitation, as Linked Data is most useful when it is cross-linked to other data sets.

In this section, we explain a possible solution to this problem.

4.3.1 The Goal

We want to enable interlinking of our data set with arbitrary other resources. More specifically, we need *meaningful* links to these other sources. If we just had an algorithm which enriches our RDF data with foreign links, we would never know if the results of the algorithm are correct or not.

On the other hand, a fully manual linking (where the user would need to add every link target by hand) would generate meaningful links but is not pragmatic enough. As users do not gain any immediate benefits by adding cross-links to their texts, they will probably skip this step in many cases.

Our proposed solution mixes the above two approaches: *We want to support the user by suggesting possibly relevant links to other Linked Data sources, but the user needs to verify them.*

Thus, we gain the benefits of both approaches: The user needs to work a lot less than with manual tagging, but we still can assume that the links are meaningful as the user has acknowledged them.

Besides implementing an algorithm for generating link suggestions, we also need to develop a user interface for acknowledging the proposed links.

In this work, we call the process of finding URIs for entities *Linkification*. It is also known as *Named Entity Linking*.

Furthermore, we need to distinguish two different use cases, which we introduce using an example:

1. *Linkification of properties.* If the Domain Model contains a property where the name of a town should be inserted, the linkification algorithm should use the full contents of the property and the *type* to find a resource URI.

   The algorithm is invoked as shown in Listing 20.

2. *Linkification of continuous text.* If the Domain Model contains a property with continuous text, such as an article text, or a product description, we first have to find the *entities* in the text and then run linkification on every one of them.

   In this case, the algorithm is invoked as shown in Listing 21.

The main difference between both use cases is that in the first case, the *type* is known beforehand, and can thus be used to find better resource URIs. In the second case, the entities and their types are *not known* a priori, and have to be extracted from the text.

```
semantifier.findSuggestions(name, type)
# returns array of Linked Data URI suggestions

# Example:
semantifier.findSuggestions("Berlin", "Town")
# returns:
# - Berlin, Berlin, Deutschland: http://dbpedia.org/resource/Berlin
# - Berlin, Usulutan, El Salvador: http://sws.geonames.org/3587266/
```

Listing 20: Linkification of properties receives a predetermined data type as input.

```
semantifier.annotate(text)
# returns annotated text with Linked Data URI suggestions

# Example:
semantifier.annotate("Sebastian Kurfuerst is a student at TU Dresden.")
# returns:
#   (1)<Person>Sebastian Kurfuerst</Person> is a student at
#   (2)<Organization>TU Dresden</Organization>
#
#   (1) - Sebastian Kurfuerst, Dresden
#         http://sebastian.kurfuerst.eu/
#       - Sebastian Kurfuerst Presentations
#         http://slideshare.net/skurfuerst/
#   (2) - Dresden University of Technology
#         http://data.semanticweb.org/organization/tu-dresden
#       - TU Dresden
#         http://www.aifb.kit.edu/id/TU_Dresden
```

Listing 21: Linkification of properties receives a predetermined data type as input.

4.3.2 General Idea

We want do develop a loosely coupled system, enabling re-use of individual components. That's why the linkification framework contains three parts:

- A component for named entity recognition (NER) and providing linkification suggestions

- A user interface for choosing amongst these suggestions
- A persistent storage mechanism, fully integrated with the RDF/RDFa framework

The general control flow is illustrated in Figure 4.7. For linkification of properties (1), the *linkification* component is triggered, the candidate results are displayed in the user interface and the user's choice is stored in the *annotation storage*. For linkification of continuous text, there is one more step at the beginning of the chain, the named entity recognition (NER).

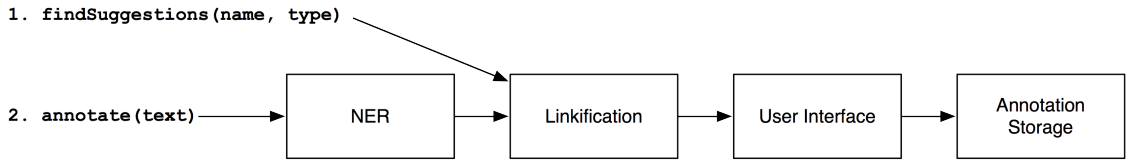

Figure 4.7: Control flow for linkification of properties and continuous texts

Each of these components can be exchanged separately; and the NER and linkification components each consist of several sub-components.

For named entity recognition, we mainly query existing web services, among them *OpenCalais* and *Alchemy API*. For linkification, we query *DBPedia* and *Sindice*.

The user interface presenting the different choices is implemented as an embeddable JavaScript component.

4.3.3 Defining a Common Vocabulary

Before focussing on each of the components in detail, we still need to talk about the *common interface* between the NER and Linkification component.

As we have seen above, the linkification component needs an *entity type* as input. Thus, we need some *common types* which are recognized by the NER, and which the linkification component gets as input.

We are using an entity type system which is roughly oriented on the names used by OpenCalais. It is extensible: By some configuration, new entity types can be added.

4.3.4 Named Entity Recognition

For Named Entity Recognition, we use foreign web services as primary resource, as these are trained with a lot of high-quality data. Thus, they have quite a high detection rate.

However, not all services can work with all languages. OpenCalais for instance is trained for English, French and some other languages, but not for German. Thus, we first do a language detection on the written text, and then choose the NER service depending on the language.

After the results from the NER are received, we need to post-process them, converting the entity types to our universal type system, such that they can be recognized by the linkifier.

4.3.5 Linkification

As our goal is to link to other resources in the semantic web, recognizing the entity type is only the first step. For each found entity, we need to find resource URIs which can be dereferenced and queried for more information.

The Linkification component receives the entity title (such as `Berlin`), and the entity type (`city`) – and based on this, should return a list of Linked Data URIs together with human-readable names (which can then be displayed).

Internally, the Linkification component queries several web services in parallel and then returns the combined result set. Right now, DBPedia and Sindice are queried, though it is very easy to implement custom queriers. DBPedia was chosen because it contains curated, high-quality data, and is the central Linked Data hub so far (as it can be seen on Figure 1.1). Sindice is being used for some alternative results not being on DBPedia; and as Sindice also crawls RDFa on web pages and microformats, we get results for these as well.

For DBPedia, a mapping is configured from the universal entity type to the DBPedia Ontology. Based on this, we do a prefix search for the given string, together with the expected DBPedia Ontology type.

On Sindice, we search for the entity title using a plain-text search. Furthermore, there is a mapping configured which maps the universal entity types to RDF types. If such a mapping is found for the given entity type, such as `Person` to `http://xmlns.com/foaf/0.1/Person`, the RDF type is also added to the search string. This gives more high-quality search results, as the results are filtered by RDF type. Still, using that only makes sense if there is a de-facto standard ontology to describe certain entity types, like it is the case with the FOAF ontology.

Furthermore, there is another issue with Sindice: It returns the *documents* in which the triples were found, not the *resource URI* itself. A small example shall illustrate this: When you search for `Sebastian Kurfürst` Sindice returns for example the result `http://sebastian.kurfuerst.eu/index.rdf`, which is the URI for my FOAF profile document. However, the URI for describing *me* is `http://sebastian.kurfuerst.eu/` – and this is also the URI we want to link to. When the data is published according to best practices, you can dereference `http://sebastian.kurfuerst.eu/` and get a 303 redirect to the document where more information can be found – in this case `.../index.rdf`.

Because Sindice returns the document URI instead of the *resource URI*, we retrieve all triples of the document from the Sindice cache, and use a heuristic to find the actual resource URI.

Right now, the heuristic looks for resource URIs which match the intended target type, and amongst them, uses the most-referenced one.

It is another area of research to fine-tune these heuristics.

4.3.6 Linkification User Interface

The Linkification User Interface is the main component the user interacts with during his daily work. He gets linkification suggestions presented, can choose amongst them, or add custom Linked Data URIs if he is an expert user.

As we are developing a generic framework to be used with any web application (built on FLOW3), we cannot assume a certain user interface structure, and must find a way to present the user's choices in a way which is independent of the overall web application layout.

Generally, we need to extend the *editing* and *creation* user interface. When the end-user creates a new object or edits it, he should also be able to add or modify the linkification information. In the vast majority of web applications, texts are entered using text fields or text areas.

Because of that, why we will extend the text fields which should be enriched by a *popover*, which is only shown when the user activates the text box. Figure 4.8 shows how the popover should appear next to the form input element as soon as it is activated. Inside the popover, the user can choose between Linked Data URIs which have been found for the given entity, or add a new Linked Data URI if the correct URI has not been found.

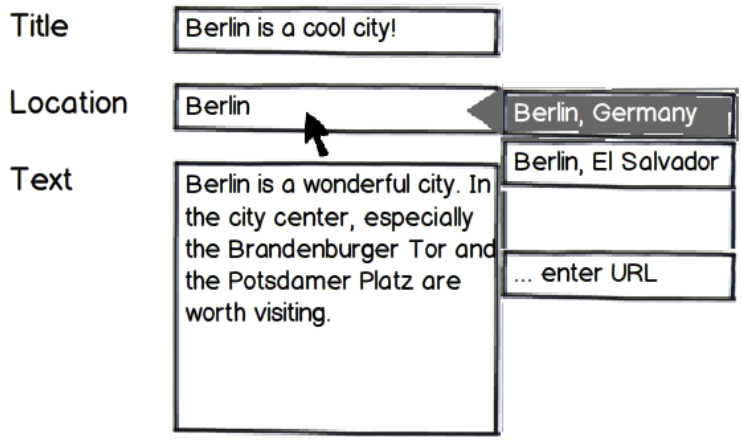

Figure 4.8: Mockup of the unobstrusive linkification interface for single properties

For enriching continuous texts, the user enters the contents in a text field, and then switches to *enrichment* mode. There, he cannot edit the text anymore, but sees a highlighting of the found entities (Figure 4.9). When he clicks on such an entity, the popover is shown next to the entity, and the user can choose the Linked Data URI (Figure 4.10).

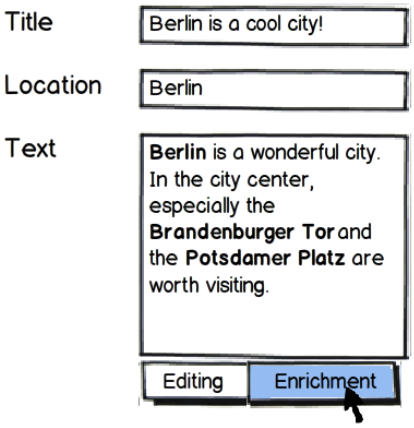

Figure 4.9: The user can toggle between enrichment and editing mode

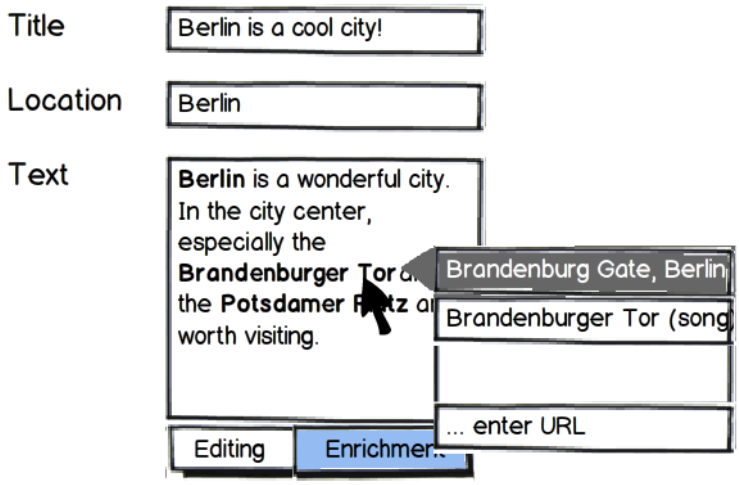

Figure 4.10: When clicking a found entity, the popover shows the possible Linked Data targets

The system also takes special care to maintain already chosen annotations when *editing text*.

4.3.7 Learning

So far, the system can handle public data, and possibly find Linked Data URIs for this kind of data. However, when being in an enterprise environment, much data is stored in *internal systems*, not public to the outside, and thus also not retrievable through Sindice for example.

Imagine a company uses an internal issue tracking system. Whenever somebody uses an internal web application, and writes something about a particular issue, the system should suggest a cross-link to this issue.

This means the annotation from Listing 22 should be successfully detected.

```
1 semantifier.annotate("Remember to check bug #1223, and fix it now.")
2
3 # returns:
4 # Remember to check (1)<Issue>bug #1223</Issue>
5 #
6 # (1) - Fix RDFa export: http://my.bugtracker/issue/1223
```

Listing 22: Domain-specific annotations should be annotated through the use of learning

So, how can we make this functionality possible? We will implement *learning*, and thus adopt the behavior of the system to the environment.

The learning process should be as simple as possible for the user: He should mark the entity in the text, add a Linked Data URI for this entity and finally select the entity type if it has not been detected correctly.

As the text enrichment is split into two phases (NER and Linkification), the learning process is also split into two parts: We need to learn the *type of the entity* for the NER phase, and then we need to learn the Linked Data URI for the *linkification* phase.

Now, we will walk through the complete learning process as an example, which is also shown in Figure 4.11.

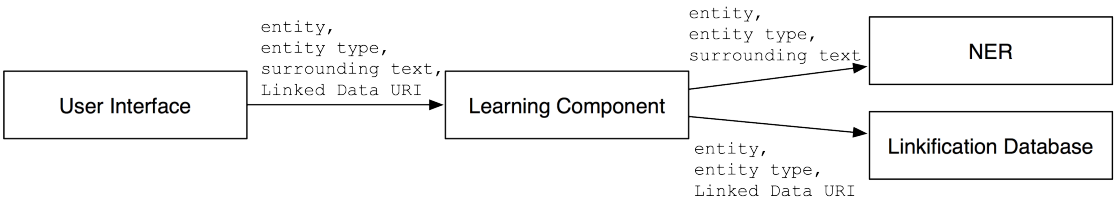

Figure 4.11: Component interactions during the entity learning process

1. *User Interface*: The user selects a text, and then a popover is shown where the user can input a Linked Data URI and select the entity type. Then, the selected text, its surroundings, the entity type and the Linked Data URI is sent to the learning component:

   ```
   "Remember to check *bug #1223*", Issue, http://my.bugtracker/issue/1223
   ```

2. The learning component now sends the tuple `(text, selected entity, entity type)` to the *NER component*, which can then learn this information.

3. Furthermore, the tuple `(selected entity, entity type, Linked Data URI)` is sent to the *linkiacficiation* component for learning.

Palladian as NER

When a NER is trained, it receives tagged input texts. Normal NERs learn both *positive* and *negative* examples, as the following example illustrates. Imagine a NER receives the following input sentence:

```
<Person>Sebastian Kurfürst</Person> is living in Dresden.
```

From this sentence, it will, as one would expect, learn, that the string `Sebastian Kurfürst` is a person. However, it will also learn that `Dresden` is *no entity at all*, i.e. no person, no place, etc.

This has two implications: Learning also negative facts greatly improves recognition performance. However, the training data for NERs needs to be highly optimized and consistent.

In our case, this is approach is not feasible: It would be very hard to tell the user that if he forgets to tag a certain entity type at some place, the system would learn the *negative information*. For an end-user, this is a very unexpected behavior.

Luckily, Palladian NER also contains a learning mode in which *only positive facts* are being learned. While this reduces precision and learning speed, it is certainly preferrable in our case, as it will make the whole system a lot more predictable to the user.

The *only-positive* learning process used by this work is also much more in line with the *open world assumption*, onto which the Semantic Web is built.

Now, the trained NER will also be queried when a text needs to be annotated, and the results will be merged together with the results of OpenCalais or Alchemy API. This way, custom results are returned together with the results of the NER web services.

Learning Linkification

The learning linkifcation engine gets a tuple `(selected entity, entity type, Linked Data URI)` and should learn it. The next time a query `(selected entity, entity type)` is received, the system should return the Linked Data URI for this selected entity. So, essentially Linkification is a string matching problem: We look for the Linked Data URI for which the entity string is most common with the input string, and the entity type matches.

Now, we still need to define *commonality*. We could use exact string matching, but a more promising approach would be to use *approximate string matching*, f.e. using Levenshtein distance or matching using n-grams.

In practice, we store the tuples to be learned in a database, and query this database to find the most fitting URI.

There are a number of *linkification matchers* which perform the matching and return the most fitting Linked Data URI. The most simple matcher, albeit the worst one, simply compares the entity strings. Only if an entity is spelled exactly the same, its Linked Data URI will be found.

A matcher which works better in practice uses a Lucene fulltext index. Because of this, it can cope with slight spelling variations. For example, if an entity string `Sebastian Kurfürst` has been learned, it will also match for `Sebastian Kurfuerst`.

Furthermore, more specific matchers can be implemented. For example, many issue-tracking or helpdesk systems use URIs similar to *http://some.uri/issue/[issueNumber]*. We could implement a matcher which checks if the entity string contains a number, and this number is found inside the URI as well. Then, when the system was trained with the tuple `("Bug #1221", Issue, http://some.uri/issue/1221)`, it can also guess the URI for the entity `Bug #4257`.

4.3.8 Implementation Architecture

Because our continuous text enrichment component uses *learning*, this has profound influence on the whole implementation infrastructure. Without learning, our component would be completely stateless; and the component could be embedded in other applications.

Thus, the first possibility would be that each application has a separate linkification component, as shown in Figure 4.12. This makes the whole system more decoupled and resilient to failures.

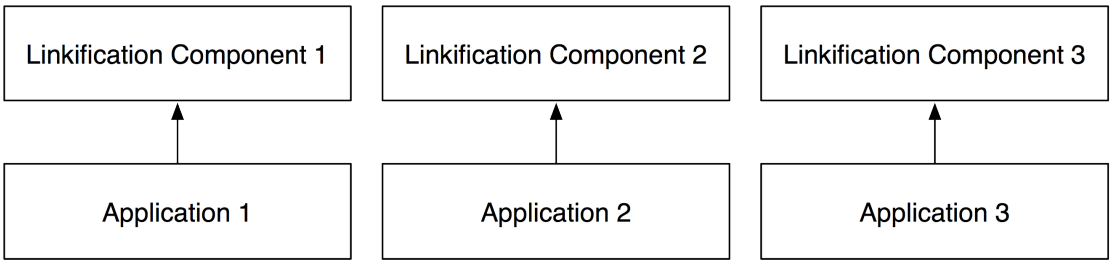

Figure 4.12: If the continuous text enrichment component is stateless, it is embeddable into applications.

However, because the linkification component *accumulates knowledge* over time, it would be beneficial to maintain just one of these services per organization. This way, all the knowledge

which is available implicitly in a company would be bundled at the linkification component over time.

Thus, the suggested architecture for a company is shown in Figure 4.13, just using one linkification component per company.

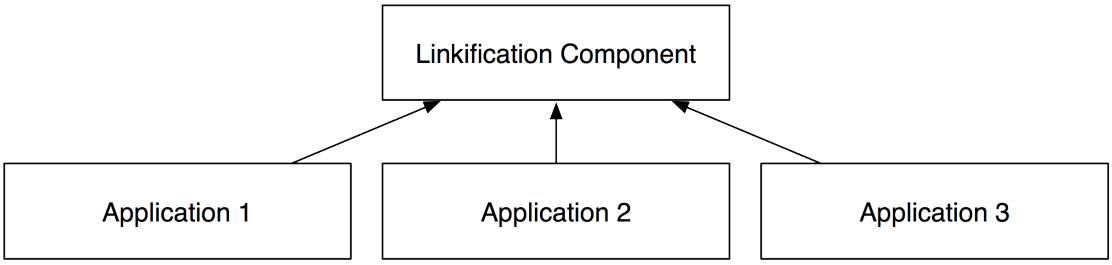

Figure 4.13: As the linkification component accumulates knowledge over time, there should be just one central linkification component per company.

Altough the linkification component is now a single point of failure, cross-linking between applications will become more efficient: If somebody manually adds a Linked Data URI, this will be available throughout all of the connected applications.

Thus, we will write the continuous text enrichment and linkification component as a *web service*, and all services of the company will use a single semantic enrichment service.

4.3.9 Interactions of Enriched Continuous Text with the RDF Generation Framework

The result of the linkification and NER process is a text with annotated Linked Data URIs. This is stored in our web application, and we will use it in various points to provide maximum business value: Both in the RDF generation and in the RDFa output, we will make the annotated texts available.

We will use the annotated Domain Model shown in Figure 4.14 as an example. There, the string `Sebastian Kurfuerst` in the blog post has been enriched by the Linked Data URI `http://sebastian.kurfuerst.eu/`.

Then, the generated RDF also contains the information about all annotations found in the text. We add the information twice to account for different kinds of queries:

- First, we insert a relation `Subject <sioc:about> AnnotatedEntity`, which makes easy to f.e. query for all articles where a given entity appears, i.e. "Find all articles which are about Sebastian Kurfürst".
- Because the first variant looses information, we need another way of querying *all* information. For that, we introduced a custom *annotation ontology*, which contains the source entity, the target entity, the property of the source entity onto which this annotation applies to, and the offset/length of the annotation. This way, all annotation information is exported losslessly as RDF triples.

  This can be used to answer questions such as: Which entities are annotated in the first 200 characters of a text?

As all this is also pushed to the triple store, we can efficiently answer the above queries using SPARQL.

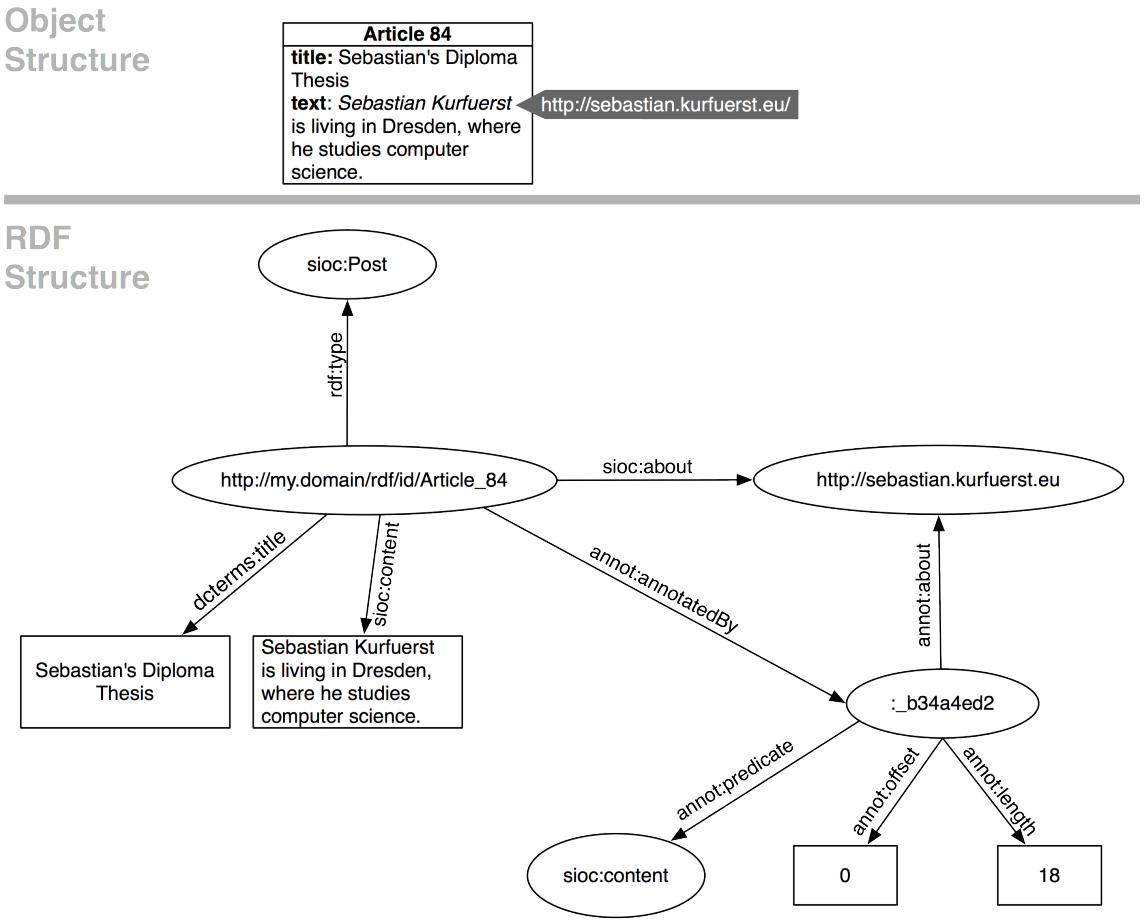

Figure 4.14: Annotations are made explicitely available as RDF, so it is queryable via SPARQL.

Besides adding the annotations to the generated RDF output, we also directly render it as RDFa as soon as the text is displayed in a template. This enables RDFa-aware browsers to gain a deeper insight about the structure of the text.

4.4 Conclusion

In this chapter, we have developed the main concepts of this work. First, we have introduced how we can expose all information which is stored inside Domain Models as RDF, using a definable *schema mapping*. We have taken special care to export a subset of RDF which is well-queryable using SPARQL.

Second, we use the already defined schema mapping to generate RDFa, mainly being meant as an entry point for RDFa aware browsers. We have provided a short algorithm which extracts the needed information for RDFa generation from the surrounding template, freeing the developer from thinking about RDFa generation.

Third, we have tackled the problem of references towards external Linked Data sources, where we distinguish between linkification of properties where the target type is known, and linkification of continuous texts. As we discovered that a fully automatic linkification solution cannot generate additional information, we have developed a user interface concept for validating the external links. This UI concept is embeddable across a wide range of web applications with very little work.

Last, we have shown that a learning process is needed for accomodating domain-specific entity types, and have shown how an open-world learning approach can be used to learn from user input. We have furthermore discussed the profound impact of having a stateful learning component on the system infrastructure.

In the next chapter, we will focus on implementation details. After an overview of the system architecture, we will give a walkthrough based on a real-world example. This will showcase all features of the developed framework.

CHAPTER

FIVE

IMPLEMENTATION DETAILS

Now, after introducing the theoretical foundations of this work, we will show how it has been implemented in practice. We first focus on the implementation architecture, and then providing a walkthrough of the framework from an implementor's perspective.

5.1 General Architecture

Figure 5.1 shows the overall architecture. The framework consists of two major parts. The first part is written in PHP, and implements the mapping of Domain Models to RDF. We will call this part *FLOW3 Semantic Framework*. The second part, which we call *Semantifier*, performs the NER and Linkification tasks. It is written as a web service, and by building it upon the Grails framework, we can use numerous Java-based language processing frameworks.

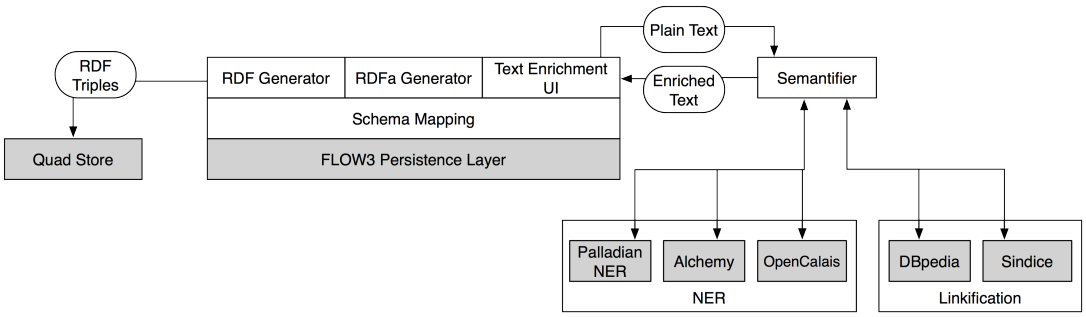

Figure 5.1: Implementation Architecture of the Semantic Framework. All components with grey background are external components.

5.1.1 FLOW3 Semantic Framework

Let's first focus on the FLOW3 Semantic Framework. At its core, it implements a mapping from Domain Models to ontologies, the so-called *schema mapping*. We implemented multiple *schema providers*, which provide configuration for the RDF generation: One for YAML[1] configuration files, and another one for source code annotations, and it is also possible to provide custom ones. There is a *chain* of schema providers active, where each schema provider can add parts to the final

[1] http://www.yaml.org/

schema, or modify the intermediate schema of the previous schema providers. This means one can use annotations such as `@rdfType foaf:Person`, or a YAML configuration like `type: foaf:Person` for configuration.

Because schema providers can override each other, it can be hard to debug the final schema. Because of this, a *schema debugger* has been implemented, showing and validating the final schema. As there are no usable validation tools for PHP arrays available as of now, we convert it to JSON and then use a *JSON Schema*[2] validator for checking the generated schema. This is a very effective means for implementors to find errors in their configuration.

The schema mapping is used by the components which provide the functionality for the end-user: The first component provides some controllers which generate RDF for a given entity in NTriple format. Furthermore, it sets up the 303 redirect scheme for distinguishing between the resource itself and the document describing it. This component also hooks into the persistence framework, being notified each time an entity changes. When an entity change occurs, it re-generates the RDF for it and sends it to an external RDF store, such that it can be queried using SPARQL. We implemented support for 4store, but because a clean API is used, other triple stores can be easily supported.

Second, we generate RDFa each time an entity is displayed. For that, we have hooked into FLOW3's templating engine *Fluid*. Fluid works in two phases: First, the template source is parsed into a syntax tree, and then this syntax tree is rendered. Furthermore, Fluid automatically caches the intermediate syntax tree as PHP files, such that the parsing does not have to occur at every stage.

Fluid provides an API for manipulating the intermediate syntax tree of the template. We use this API to intercept all object accesses which are displayed in the template, and wrap some code around it which generates the RDFa.

The third component is responsible for implementing the semantic enrichment and linkification user interface. For this, we again hook into the Fluid templating engine, intercepting every form input element, and then deciding if the enrichment and linkification user interface needs to be shown, depending on the configuration. The linkification and enrichment UI is then implemented in JavaScript using jQuery.

The enrichment and linkification itself is happening in the *Semantifier* web service which is called through cross-domain AJAX requests. The linkification and enrichment results are stored as hidden fields using a special naming convention in the form, appending `_metadata` to fields which are annotated. Thus, the data being sent back to the server looks similar to Listing 23.

As the semantic enrichment takes some seconds, this is done in the background while the user types his text. By that, we are able to directly show found entities as soon as the user switches to enrichment mode. FLOW3 then reads this data and converts it back to an object using the so-called *property mapper*[3]. This is extensible, so we hook into it and save the metadata separately from the objects themselves. When the object is then rendered as RDF or RDFa, we use the linkification information if available.

[2] http://tools.ietf.org/html/draft-zyp-json-schema-03

[3] http://flow3.typo3.org/documentation/guide/partiii/propertymapping.html

```
article[author] = Sebastian Kurfuerst
article[location] = Berlin

# added by the linkification user interface:
article[location_metadata] = http://dbpedia.org/resource/Berlin
article[text] = ....

# added by the linkification user interface:
# annotations encoded as JSON; added by the linkification UI
article[text_metadata] = ...
```

Listing 23: Enriched data which is sent back to the server on a form submission

5.1.2 Semantifier

The semantifier is a web service responsible for linkification of entities and continuous text. It is implemented in Groovy[4] using the Grails framework[5].

It consists of three parts. First, it uses a NER web service for finding entities in a text. Second, it tries to find Linked Data URIs for entities using numerous search services. Third, it provides functionality for learning new entities, both for named entity recognition and linkification.

For named entity recognition, the source language of the input is determined. After that, the configured NER component for the given language is used. By default, we use OpenCalais[6] for English texts and Alchemy API[7] for German texts; but this mapping is again configurable.

Parallel to querying the NER web service, the local NER *Palladian*[8], which contains the custom-learned entities, is also triggered, and their results are merged together.

In the linkification step, multiple *linkification services* are requested to find a Linked Data URI for found entities. Here, we do multiple parallel requests to DBPedia, Sindice and the local database of learned entities, and merge the results together.

For finding results in the local database of learned entities, we use a Lucene[9] index to find entity URIs also when they do not fully match, i.e. when the string `Kurfürst` is stored in the database, but the user searches for `Kurfuerst`.

In order to improve response times of the Semantifier, we parallelize the requests to the backend web services, aborting them if no reply was received within a short timeout.

5.2 A Complete Walkthrough

Here, we want to give a complete walkthrough from an implementors perspective, showing the advanced features of the framework.

[4] http://groovy.org
[5] http://grails.org
[6] http://opencalais.com
[7] http://alchemyapi.com
[8] http://palladian.ws
[9] http://lucene.apache.org

5.2.1 Example Package

Our example is the `SandstormMedia.MicroBlog` package which has been created specifically to show how the semantic framework can be used.

The main Domain Model is the `BlogEntry` class. Each blog entry has a `title`, `location`, `creationDate`, `teaser`, `text` and `relatedPosts`. In PHP, it looks as shown in Listing 24. We will now explain how we can use this work to export the data into the Semantic Web.

5.2.2 Installing The Semantic Package

First, we need to install the `Semantic` package. For that, we check it out from the repository, and place it in the `Packages/Application/SandstormMedia.Semantic` folder of our FLOW3 distribution.

Then, we need to activate the package using `./flow3 package:activate SandstormMedia.Semantic`.

Now, we need to import the routes from the package into our global routes. For that, we need to edit `Settings/Routes.yaml` and insert the YAML configuration from Listing 25 as the first route. This instructs FLOW3 to redirect every URI starting with `rdf/id/` and `rdf/data/` to the controllers of the `Semantic` package.

```
-
  name: 'Semantic'
  uriPattern: '<SemanticSubroutes>'
  subRoutes:
        SemanticSubroutes:
          package: SandstormMedia.Semantic
```

Listing 25: This configuration must be added to the global routes.

As a last step, we need to configure the base URI of the FLOW3 instance in the settings. For that, open up `Configuration/Settings.yaml` in your FLOW3 distribution and add the code from Listing 26.

```
SandstormMedia:
  Semantic:
    baseUri: 'http://blog.local' # adjust to your needs
```

Listing 26: This configuration must be added to the global settings.

Now, the package is set up and can be used.

5.2.3 Finding The Right Ontologies

For exporting the `BlogEntry` as RDF, we need to provide a mapping to an ontology. So, we first need to find suitable ontologies which can act as the mapping target. After some google queries,

```php
/**
 * @entity
 */
class BlogEntry {

  /**
   * @var string
   * @identity
   */
  protected $title;

  /**
   * @var string
   */
  protected $location;

  /**
   * @var \DateTime
   */
  protected $creationDate;

  /**
   * @Column(type="text")
   * @var string
   */
  protected $teaser;

  /**
   * @Column(type="text")
   * @var string
   */
  protected $text;

  /**
   * @var \Doctrine\Common\Collections\Collection
   *       <SandstormMedia\MicroBlog\Domain\Model\BlogEntry>
   * @ManyToMany
   * @JoinTable(inverseJoinColumns={@joinColumn(name="related_id")})
   */
  protected $relatedPosts;

  public function __construct() {
        $this->creationDate = new \DateTime();
        $this->relatedPosts =
          new \Doctrine\Common\Collections\ArrayCollection();
  }

  // ... getters and setters for every property ...
}
```

Listing 24: A simple FLOW3 Domain Model

we find the *Semantically Interlinked Online Communities (SIOC)* Ontology[10], which seems well-suited for our blog.

After some reading in the ontology specification, we find that the blog post could be represented by the type `http://rdfs.org/sioc/types#BlogPost`. Furthermore, we find an example on how a blog post is mapped in the ontology specification (shown in Listing 27).

```
<sioc:Post rdf:about="http://johnbreslin.com/blog/2006/sioc-clouds/">
    <dcterms:title>
        Creating connections between discussion clouds with SIOC
    </dcterms:title>
    <dcterms:created>2006-09-07T09:33:30Z</dcterms:created>
    <sioc:content>... blog text here ...</sioc:content>
    <!-- the example has been shortened for better readability. -->
</sioc:Post>
```

Listing 27: An example from the SIOC specification, demonstrating a possible use for the ontology.

By reading this example, we learn the following things:

- The specification encourages the use of *Dublin Core* for the title and creation date.
- There is a `sioc:content` which seems to be useful for the full content of the post.

Furthermore, we would like to add the mapping of related posts. By reading the SIOC ontology, we find *sioc:related_to*, which can be used to express relations towards other posts – exactly our use case.

Thus, we can use the mapping depicted in Listing 28. While the mapping process still involves quite a bit of manual work, a wizard could be created at a later stage which suggests mapping for the most well-known ontologies.

```
BlogPost -> sioctypes:BlogPost
    title -> dcterms:title
    creationDate -> dcterms:created
    text -> sioc:content
    relatedPosts -> sioc:related_to
```

Listing 28: A schema mapping from the Blog Domain Model to Ontologies

5.2.4 Adding Ontology Mappings

Now that we have thought of the mapping between Domain Models and ontologies, we can configure the framework accordingly. We open up our `BlogEntry` Domain Model and add some `@rdfType` annotations to the class and the properties we want to map, yielding the Domain Model in Listing 29. Now, we can use the *schema debugger* to see if we configured our mapping correctly. For that, open `http://blog.local/rdf/debug` in your browser (replace *blog.local* by the hostname to your FLOW3 installation). When all the mappings are green, there is no error, as shown in Figure 5.2.

[10] http://sioc-project.org/

```php
/**
 * @entity
 * @rdfType sioctypes:BlogPost
 */
class BlogEntry {

        /**
         * @var string
         * @identity
         * @rdfType dcterms:title
         */
        protected $title;

        /**
         * @var string
         */
        protected $location;

        /**
         * @var \DateTime
         * @rdfType dcterms:created
         */
        protected $creationDate;

        /**
         * @Column(type="text")
         * @var string
         */
        protected $teaser;

        /**
         * @Column(type="text")
         * @var string
         * @rdfType sioc:content
         */
        protected $text;

        /**
         * @var \Doctrine\Common\Collections\Collection
                  <SandstormMedia\MicroBlog\Domain\Model\BlogEntry>
         * @ManyToMany
         * @JoinTable(inverseJoinColumns={@joinColumn(name="related_id")})
         * @rdfType sioc:related_to
         */
        protected $relatedPosts;

        // ... getters and setters for every property ...
}
```

Listing 29: Domain Model with RDF mapping annotations

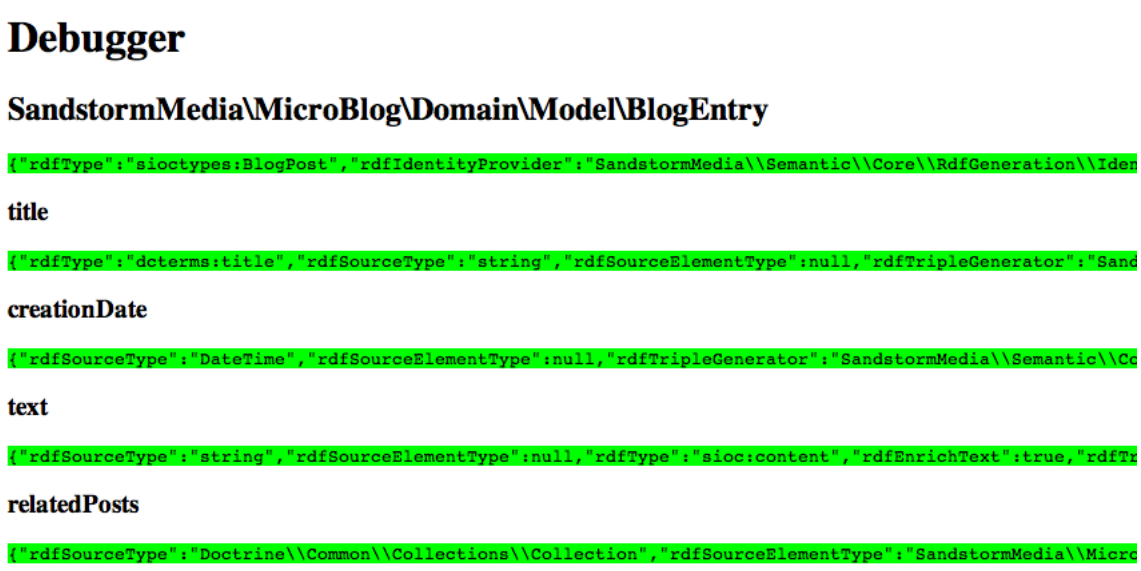

Figure 5.2: Schema Mapping Debugger

We could have also used a YAML configuration file to define the schema mapping. However, this is only suggested when you have no control over the source code of the package, for instance because it is maintained by a third party.

5.2.5 Browsing RDF and RDFa Data

Now, we can browse our RDF and RDFa data already. When we open the web application, we will see that the values we have output are automatically enriched by RDFa tags (Listing 30).

```
<h2><a href="http://blog.local/microblog/berlin-is-a-cool-city">
    <span about="http://blog.local/rdf/id/...4167b5e5d2d4"
          xmlns:dcterms="http://purl.org/dc/terms/"
          property="dcterms:title">
                Berlin is a cool city
    </span>
</a></h2>
```

Listing 30: HTML with embedded RDFa

If you want to globally disable RDFa generation, you can do that in your `Settings.yaml` file, by setting `SandstormMedia: Semantic: rdfa: enable: false`.

When we now dereference the URI given in the `about` attribute of the `span` tag, we will get redirected from `/rdf/id/[dataType]/[uuid]` to `/rdf/data/[dataType]/[uuid]` using a 303 redirect, as it is suggested by the Linked Data publishing guidelines. The second URI then returns the RDF for the given entity in NTriples format, as it is shown in 31.

```
// shortened for better readability
<http://blog.local/id/...4167b5e5d2d4>
    rdf:type
    <http://rdfs.org/sioc/types#BlogPost>.
<http://blog.local/id/...4167b5e5d2d4>
    dcterms:title
    "Berlin is a cool city".
<http://blog.local/id/...4167b5e5d2d4>
    dcterms:created
    "2011-10-05T12:40:48+02:00"
    ^^<http://www.w3.org/2001/XMLSchema#dateTime>.
<http://blog.local/id/...4167b5e5d2d4>
    sioc:content
    "...".
<http://blog.local/id/...4167b5e5d2d4>
    sioc:related_to
    <http://blog.local/id/...1577a5e5f23a>.
```

Listing 31: All URIs which are generated are automatically dereferenceable and contain the RDF data for the entity.

5.2.6 Enabling 4Store

After downloading 4store[11], we need to initialize its storage backend and then start an HTTP server which can answer SPARQL queries. For that, we run the commands shown in Listing 32.

```
4s-backend-setup test # creates the storage backend
4s-backend test       # start the storage backend
4s-httpd test         # start the HTTP server
```

Listing 32: Initialization of 4store

Go to http://localhost:8080, and see your local 4store instance running.

Now, you might need to configure your 4Store base URI in `Settings.yaml`, by configuring `SandstormMedia: Semantic: 4Store: baseUri`.

Now, we only need to do an initial import of our data to the quad store. That is possible using the supplied FLOW3 command:

```
./flow3 triplestore:import
```

Now, when you visit `http://localhost:8080/status/` you will see how many triples have been imported.

5.2.7 Running a SPARQL query

We can now for example query our data store for the titles and creation dates of all blog posts which have a related blog post. For that, we can use the SPARQL query shown in Listing 33.

[11] http://4store.org

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX sioc: <http://rdfs.org/sioc/ns#>

SELECT DISTINCT ?created, ?title WHERE {
  ?s dcterms:created ?created.
  ?s dcterms:title ?title.
  ?s sioc:related_to ?y.
}
```

Listing 33: SPARQL query which shows the titles and creation dates for blog posts with relations

This will result for example in the following result listing (formatted as table):

?created	?title
2011-10-05T12:40:48+02:00	Berlin is a cool city

As soon as multiple applications post their data to one triple store, we can formulate queries spanning multiple systems. Thus, we can aggregate information across system boundaries.

By now, interlinked RDF and RDFa is generated, following all the best practices. However, there are no outgoing links to other Linked Data sources such as DBPedia yet. We will look into this in the following sections.

5.2.8 Enabling Linkification for Properties

Because the linkification component augments the user interface using JavaScript, we have to include some CSS and JavaScript files in our web application (Listing 34).

Then, the linkification can again be enabled by adding some annotations. In our case, we want to suggest cities when the user edits the `$location` field of the `BlogPost`. The configuration shown in Listing 35 enables this.

```
<!-- only include jQuery if you did not include it before -->
<script src="...jquery/1.6.2/jquery.min.js"></script>
<script src="{f:uri.resource(package: 'SandstormMedia.Semantic',
              path:'jquery-popover/jquery.popover.js')}"></script>
<script src="{f:uri.resource(package: 'SandstormMedia.Semantic',
          path:'widgets.js')}"></script>
<link rel="stylesheet"
       href="{f:uri.resource(package: 'SandstormMedia.Semantic',
          path:'jquery-popover/jquery.popover.css')}" />
<link rel="stylesheet"
       href="{f:uri.resource(package: 'SandstormMedia.Semantic',
          path:'style.css')}" />
```

Listing 34: Required JavaScript and CSS include files for linkification

```
1 /**
2  * @var string
3  * @rdfType foaf:based_near
4  * @rdfLinkify true
5  * @rdfLinkificationType City
6  */
7 protected $location;
```

Listing 35: Configuration of property linkification

The `@rdfLinkify` annotation (line 4) enables the linkification features, and `@rdfLinkificationType` (line 5) specifies the expected type. We use the types from OpenCalais[12] as a starting point, but it is also easily possible to define custom ones.

Now, you need to start the *Semantifier* web service, and then you can use the linkification. When creating or editing a `BlogPost` and modifying the location, it is automatically searched using DBpedia and Sindice, looking like Figure 5.3.

Figure 5.3: Linkification of single properties

[12] http://www.opencalais.com/documentation/calais-web-service-api/api-metadata/entity-index-and-definitions

After selecting an entry, the underlying Linked Data URI will be stored in the system. When RDF or RDFa for a location is generated, it does not anymore contain the string `Berlin` but the resource URI `http://dbpedia.org/resource/Berlin` (depending on what has been chosen by the user). This is also sent to the triple store.

Now, our microblog example application is a fully-compliant member of the Linked Data space: All data is exported as RDF and RDFa, is queryable using SPARQL, has persistent and dereferenceable URIs and can contain links to other Linked Data sources.

We have now made all *explicit information* available to the outside as Linked Data. However, there is also information stored *implicitely* in continuous texts. Now, we will look at extracting this information.

5.2.9 Enabling Continuous Text Enrichment

Now, we want to extract the information stored inside the contents of our blog post. As we already included the necessary JavaScript and CSS libraries above, we only need to adjust our schema mapping. By annotating `@rdfEnrichText true` we enable these features, as shown in Listing 36.

```
1 /**
2  * @var string
3  * @Column(type="text")
4  * @rdfType sioc:content
5  * @rdfEnrichText true
6  */
7 protected $text;
```

Listing 36: Configuration of continuous text enrichment

Now, the user interface is again modified, showing an *enrich* button underneath the form input element which triggers the text enrichment. This can be seen in Figure 5.4.

When the user switches to enrichment mode by pressing the *enrich* button, the text area is replaced by a widget which shows the possible annotations in the text, as it can be seen in Figure 5.5. There are three types of annotations:

- Green annotations are already fully tagged by the user, i.e. they have a *confirmed* Linked Data URI.
- Yellow annotations are not yet confirmed by the user, but there are some candidate Linked Data URIs found.
- Red annotations are not yet confirmed by the user. Furthermore, the user needs to manually search for a Linked Data URI as the system was not able to find one.

All annotations can be clicked, showing the already-known Linked Data URI chooser (Figure 5.6).

As the annotation process takes some time (between 3 and 10 seconds usually), it is automatically triggered while the user adds or modifies text. This greatly reduces the time the user needs to wait for the enrichment process, as it is continuously done. In order to provide the user some indication if a background action is happening, a progress indicator is shown when an action takes place.

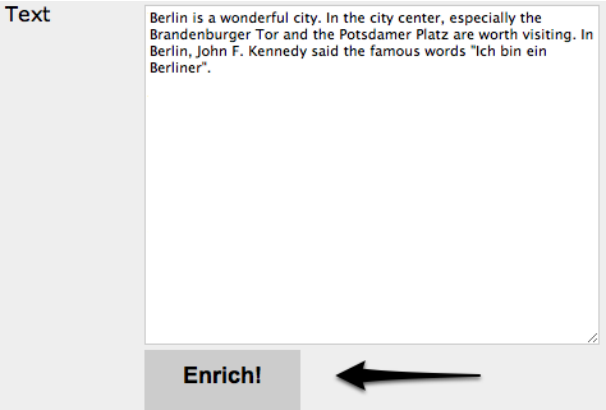

Figure 5.4: The enrichment mode switcher is automatically inserted into the user interface

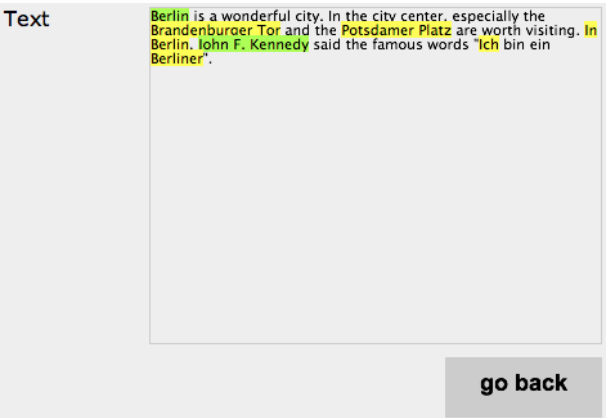

Figure 5.5: In enrichment mode, found entities are highlighted.

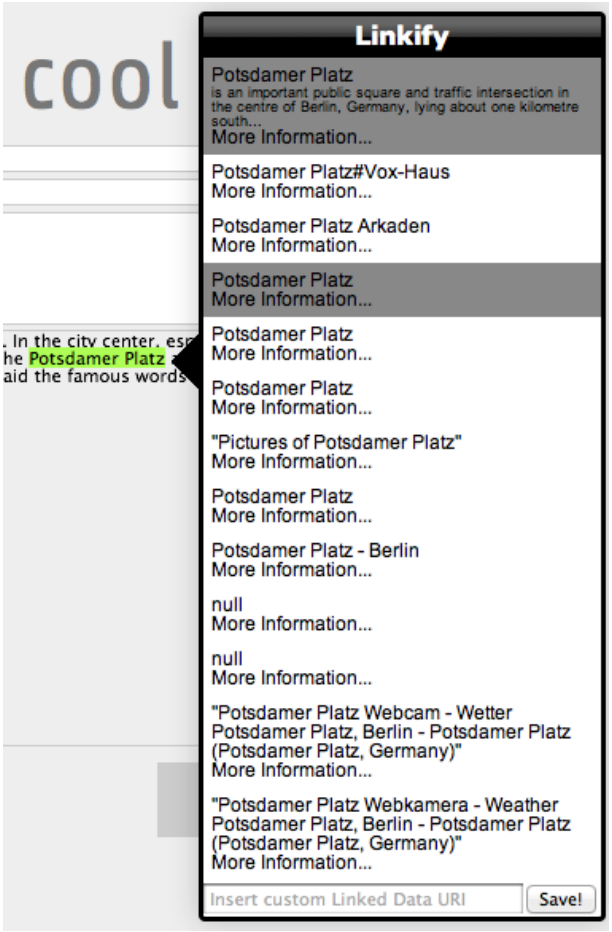

Figure 5.6: Clicking an annotation shows the already-known Linked Data URI chooser.

5.2.10 Querying Continuous Texts

As the chosen annotations are also exported into the triple store, we can formulate a query like "Find the titles of all blog posts which have some annotated content" (Listing 37).

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX sioc: <http://rdfs.org/sioc/ns#>
PREFIX sioctypes: <http://rdfs.org/sioc/types#>

SELECT ?title ?o WHERE {
   ?s rdf:type sioctypes:BlogPost.
   ?s dcterms:title ?title.
   ?s sioc:about ?o.
}
```

Listing 37: Finding all blog posts which have some annotated content

Depending on the annotations you did, this could return the following results:

?title	?o
Berlin is a cool city	http://dbpedia.org/resource/Brandenburg_Gate
Berlin is a cool city	http://dbpedia.org/resource/Potsdamer_Platz

As the annotations are also stored in a more verbose format, we can also answer queries like "Return only annotations which occur in the first 200 characters of all texts", shown in Listing 38.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX sioc: <http://rdfs.org/sioc/ns#>
PREFIX sioctypes: <http://rdfs.org/sioc/types#>
PREFIX annot: <http://sandstorm-media.de/ns/2011/annotation#>

SELECT ?title ?o WHERE {
   ?s rdf:type sioctypes:BlogPost.
   ?s dcterms:title ?title.

   ?s annot:annotatedBy ?annotation.

   ?annotation annot:about ?o.
   ?annotation annot:offset ?annotationStartOffset.
   FILTER (?annotationStartOffset < 200).
}
```

Listing 38: Finding all annotations occuring in the first 200 characters of the text

This would then return only the results at the beginning of the text:

?title	?o
Berlin is a cool city	http://dbpedia.org/resource/Brandenburg_Gate

5.2.11 Using Learning

So far, we have only shown how the user can work with the system when the system provides the correct suggestions. However, there are cases when this automatic suggestion does not work, and we distinguish two different kinds of errors: First, the entity type was determined correctly, but no Linked Data URI could be found for it. Second, not even the entity was found in the text, so no Linked Data URI was found for it.

To solve the first case, the Linked Data URI chooser has a free-form text field where the user can enter an arbitary Linked Data URI and an entity type. Together with the surrounding text, this is then sent to the *Semantifier*, which learns this Linked Data URI for future use.

For solving the second case, the user can select an arbitrary text, specifying the entity type and the Linked Data URI. This is again sent to Semantifier, which learns this information then. While this is already implemented on the server side, it is not yet done on the client side because of time constraints.

5.3 Conclusion

This chapter explained the implementation of this work on a more fine-grained level. We first introduced the software components which have been developed for this work, focussing on implementation details.

After that, we have given a hands-on walkthrough, showcasing all parts of the developed framework. For implementors following these guidelines, it should be easily possible to transfer his own FLOW3 application into the Semantic Web.

In the next chapter, we focus on evaluating the developed framework, comparing it with alternative implementations and testing it against its requirements.

CHAPTER

SIX

EVALUATION

In this chapter, we will evaluate this work. As it consists of three major components, we will evaluate these components one-by-one. Depending on the component, we will use different evaluation techniques.

6.1 Exporting Domain Models to RDF

We automatically want to generate RDF which adheres to the current best practices of Linked Data. In Chapter 2.1.5, we have listed eight best practices for publishing Linked Data. We will now go through this list and check if we were able to fulfil these requirements. A ✓indicates that we fulfill this requirement, ⇛indicates possible improvements and ✗indicates when we do not fulfill the requirements.

1. *Use URIs as names for things.*

   ✓Every Domain Object gets a URI `http://[host]/rdf/id/[type]/[uuid]` assigned by the framework. Because it contains the UUID of the object, the URI is guaranteed to be the same throughout the whole object life cycle.

2. *Use HTTP URIs, so that people can look up those names.*

   ✓We use HTTP URIs, as FLOW3 is a web framework.

3. *When someone looks up a URI, provide useful information, using the standards.*

   ✓When the URI of an object is requested, the RDF triples of the object are returned in NTriples format. This includes the type and all its exported properties.

   ⇛We always return NTriples currently. It would be nice if a browser which does not support NTriples got a HTML representation of the triples. This could be done using content negotiation.

4. *Include links to other URIs, so that they can discover more things.*

   ✓We include links to referenced entities. When the developer has enabled the *linkification* and *continuous text enrichment* features, we also include links to arbitrary other Linked Data URIs.

5. *Use 303 redirects or hash URIs for distinguishing between real-world objects and documents describing them.*

✓We generally use 303 redirects, redirecting from real-world objects to the document which describes it.

6. *Avoid RDF reification.*

   ✓We do not support RDF reification.

7. *Avoid RDF Collections and RDF Containers as long as the order of items does not matter.*

   ✓We currently do not support RDF Collections or Containers.

   ➠When the order matters, the developer should be able to use RDF Collections or Containers using a simple annotation. This could be easily implemented if needed.

8. *Avoid using blank nodes as much as possible.*

   ✓We do not use blank nodes, except when representing *value objects*. In this case, using blank nodes makes sense, as a value object is only represented by its values and does not have an external identity. Hence, it should also not be referenceable using a persistent URI.

While we fulfill all of the requirements defined in Chapter 2.1.5, there is still some room for future improvement. However, this is easily possible because of the extensible concepts used.

6.2 RDFa Generation

In Chapter 2.1.8, we have outlined how well-formed RDFa should look like. In Chapter 5.2.5, there is an example for the automatically generated RDF. We will now check this against our requirements:

1. *Validity*

   ✓The RDF we generate is syntactically valid, and the RDF triples extracted from the RDFa tags match the triples inside the RDF representation.

2. *Structural View*

   ✗As we do not know if the developer who wrote the template intended a certain mapping between document structure and semantical structure, it is more safe to not infer this information. That is why we do not move RDFa `about` properties to parent nodes in the document structure.

   ➠As soon as browsers support visualizing RDFa information to the user, the structural view should nevertheless be implemented. This could be done either by a heuristic which analyzes the document structure, or by some helpers which the developer of the webpage could use to provide structural information.

In the area of RDFa generation, the system fulfills the mandatory requirement, but not the structural one. As the generated RDFa should serve only as an entry point to the Linked Data cloud, this is acceptable as of now. In the future, there are ways to improve this.

6.3 External References

For evaluating the external references to foreign Linked Data sources, we take a two approaches: First, we compare the built-up solution with other frameworks and user interfaces in terms of

functionality. We use a weblog about cars as an example web application (introduced below), and analyze the requirements of this towards such a referencing solution.

Then, we analyze one part in detail: The entity type learning system. This is particularily important as we use an *open world* approach there, which is not common to NER systems.

6.3.1 Introducing The Car Blog Use Case

Here, we introduce the example we use for evaluation; and based on this example we derive requirements for our work. Still, although we do an evaluation based on a certain use case, this is easily applicable to other real-world examples where freeform text is entered at some point into a web based system. This includes other blogs, shop systems, news management systems, forums and knowledge databases. But let us get back to our example:

Imagine you write a weblog about cars, their manufacturing process, technical details or the recent motor shows. You want to enrich your blog posts with more semantic information, as this makes your blog posts more easily searchable and linkable. However, as you are a non-technical writer, you do not want to get in contact with any of the underlying technologies of the Semantic Web. You should not need to learn about ontologies or RDF.

1. The technical underpinnings of the Semantic Web should be hidden.

Furthermore, when you write a blog post, you are focussed on the contents of this blog post, not the semantic enrichment. Some users prefer a rich text editor, while others want to have more control over the generated HTML, thus they prefer to write their blog posts in a simple text area. From that stem two requirements:

2. Semantic enrichment must be integratable with simple text areas.
3. Semantic enrichment must be integratable with rich text editors.

As enriching and tagging your blog posts is an additional effort you have to make, it must be easy and quick to do so.

4. Semantic enrichment must be easy and quick.

Thus, you prefer automatic solutions over fully manual tagging. Still, you understand that an unsupervised automatic solution does not add any informational value to the posts, as you do not know how much this information can be trusted. That is why you want to supervise the enrichment process.

5. Semantic enrichment should be automatic, but supervised.

While researching on the topic of semantic enrichment, you realize that certain entity types are detected in most systems, such as companies, people, or events. For your blog, you also want to tag cars, motor shows, and racing competitions. The system should be trainable to also cope with these entity types. This training should happen in an intuitive way.

> *Technical Sidenote:* “Intuitive training” in this context means an open world learning approach as introduced in Chapter 4.3.7 in section *Palladian as NER*.

6. The enrichment system must be trainable towards new entity types.
7. Only positive facts should be learned, in line with the open world assumption.

Furthermore, we have a special feature as we are often blogging live from certain events: These blog posts have a "location" field which is set to the city the event takes place in, to make location-based searches possible (i.e. *Show me all events 200 km around my hometown*). You as a user only want to type the name of the city into an input field, and then the enrichment should also be done in an unobstrusive way.

8. Enrichment on single input fields, where the entity type is known beforehand, should be possible.

If you are a power user, you might want to add custom data sources like your personalized del.icio.us link list or flickr images, which should also be taken as sources for enrichment.

9. Custom enrichment sources should be possible.

Based on the above requirements, we will compare different text enrichment systems in the next section.

6.3.2 Feature Comparison

We will now check each evaluated system against all requirements, and summarize the results in a feature matrix shown in Table 6.1. After the tabular overview, the comparison results are explained in detail.

	Loomp	RDFaCE	FLOW3 Semantic Framework
1. Hide technical underpinnings of Semantic Web	✓	✗	✓
2. Integration into standard text areas	✗	✗	✓
3. Integration into WYSIWYG Editors	✗	✓	✗
4. Usable Semantic Enrichment Process	✗	✓	✓
5. Enrichment is automatic, but supervised by the user	✗	(✓)	✓
6. Training of custom entity types possible	✗	✗	✓
7. Positive facts learned, negative ones ignored	✗	✗	✓
8. Facts with known entity type are linkable	✗	✗	✓
9. Integration of custom enrichment sources	✗	✓	✗

Table 6.1: Comparison between different linkification user interfaces

Loomp

Loomp has been evaluated in October 2011 based on the online demo[1], as it was not possible to download the system and test it locally.

1. ✓Loomp hides the technical details, the user only works with concepts such as people, organizations or events.
2. ✗It is not easily possible to integrate Loomp in a foreign web application. It needs its own widgets for text writing, which does not integrate into the look-and-feel of the web application.
3. ✗, same reasons as 2.
4. ✗Enrichment must be done manually, by selecting entities in the text and then finding the right category. The user interface is not intuitive for that.
5. ✗While automatic enrichment is planned, it could not be found in the demo application.
6. ✗, because of 4.
7. ✗, because of 4.
8. ✗, because it provides its own heavy-weight widget, and no public API
9. ✗, because of 4.

RDFaCE

RDFaCE has been evaluated in October 2011 based on the online demo[2].

1. ✗Users get in contact with ontologies directly, and at least need a rough understanding of it to use the tool.
2. ✗RDFaCE can not easily replace a simple text area, as it is built upon TinyMCE[3].
3. ✓RDFaCE is (in itself) a plugin to a rich text editor, so editing annotations and formatting the text is no problem.
4. ✓The user can first concentrate on writing his text, and then enrich it whenever he finds the time to do so.
5. (✓) While RDFaCE supports automatic enrichment, all found annotations are automatically added to the text. There is no guidance in telling the user which annotations still need to be checked.
6. ✗RDFaCE is a purely client-side solution, so it does not contain any NER which could be trained.
7. ✗, because of 6.
8. ✗, only works with full-blown Rich Text Editors.
9. ✓, as it is purely client-side, it is possible to add custom enrichment sources based on the current user.

[1] http://demo.loomp.org/oca/
[2] http://rdface.aksw.org/test/tinymce/examples/rdfaDemo.html
[3] http://www.tinymce.com/

FLOW3 Semantic Framework

1. ✓Users do not get in touch with ontologies, but only get human-readable strings and descriptions to choose from.
2. ✓The framework is built to enhance a simple text area.
3. ✗It is not easily possible to combine this enhancement with a rich text editor right now.
4. ✓Enrichment is a separate step the user can take after writing his text. Also, it is possible to switch between enrichment and writing mode.
5. ✓Enrichment suggests entity types to the user, but he has to choose which one to take.
6. ✓Training of new entity types is possible, as the system wraps the Palladian NER as web service.
7. ✓Only positive facts are learned.
8. ✓using the same un-obtrusive user interface.
9. ✗Not easily possible because of the server-side enrichment. However, could be implemented by custom authorization and user-defined sources in the Semantifier web service.

6.3.3 Evaluating The Learning Process

We will analyze the learning process of Palladian for new entity types, particularily under the open world assumption. For this, we used a real-world car blog[4]. We extracted the plain texts of the last 50 blog posts using boilerpipe[5], and then manually tagged the following entity types:

- *Car:* car names, such as `Honda Civic` or `VW Golf`
- *Organization:* car manufacturers such as `Honda` or `Volkswagen`, other companies and organizations like `ADAC`
- *Event:* trade shows and racing events, such as `Frankfurt International Motor Show`
- *Place:* countries, cities, such as `Frankfurt` and `Germany`, `Europe`
- *Person:* persons, company managers, race drivers
- *O:* other entities. We filter them out from the evaluation, as they are not shown to the user.

We deliberately distinguished between *Car* and *Organization*, although it requires a lot of context knowledge to decide if e.g. `Honda` is a reference to a particular car or a manufacturing company, as that is a requirement from this use case.

During the manual tagging process, leftover boilerplate markup and wrongly detected articles have been removed. 50 documents have been fully tagged. 25 of these documents are used as testing set. We execute different runs, increasing the number of training documents from 1 to 25.

Figure 6.1 shows how recognition performance improves with the number of documents used for training. On the x axis, the number of training documents is shown, while on the y axis, various performance scores can be found.

[4] http://theblogaboutcars.com/
[5] http://code.google.com/p/boilerpipe/

We distinguish between *exact match* and *MUC* performance. In *exact match* scores, an entity is only counted as correct match if the *entity type and the boundary* have been correctly determined by the NER. In *MUC* scores, the entity type and the boundary is assessed individually. This distinguishes between the following cases:

- no match at all
- correct boundaries but wrong entity type
- wrong bondaries but correct entity type
- correct boundaries and correct entity type

Both *exact match* and *muc* performance are measured by three values: precision, recall and F1 measure. *Precision* is calculated by dividing the number of correctly found entities through the number of all found entities, effectively stating how relevant the found entities are. *Recall* is calculated by dividing the number of all entities in the text through the number of correctly found entities in the text. This expresses how many of the possible entities in the text have been found. The *F1 measure* is the *harmonic mean* of precision and recall.

In Figure 6.1, it can be seen that all measures increase with the number of training documents. We will now focus on the case with 25 training documents: 55 % of all possible entities are found fully correctly (exact match recall), and 74 % are found which have at least a correct boundary or entity type (MUC recall).

However, there are many false positives found: The precision has a value of 36 % (exact match precision) respectively 49 % (MUC precision), meaning that more than half of all found entities are not relevant. That again shows how important it is to have a *manual confirmation step* by the end-user, who can easily eliminate many false positives.

In Table 6.2, the *confusion matrix* for 25 training and 25 testing documents can be found. It shows the expected entities on the x axis (*real*), and the predicted entities on the y axis. As an example, a *Place* was identified correctly in 21 cases, while in 4 cases a *Car* has been identified wrongly as *Place*. The last column (*OTHER*) shows how often an entity was found whereas no entity was tagged. For example, the system characterized 14 entities as *Place* which were not tagged at all in the test set.

When the NER works perfectly, the prediction always matches the real entity type. Thus, the matrix would consist of all zeroes except the diagonal axis which would contain all values.

In our case, it can be seen that the prediction for *Place, Event, Organization* and *Person* is often correct, while it is hard for the system to predict *Car* correctly. In 29 cases, the NER tags organizations as cars, while in 11 cases, it tags people as cars. This shows that the system has difficulties in distinguishing cars and organizations, which is in line with our expectations. Furthermore, the system has a high false-positive rate while detecting cars: 57 times, it found a car where nothing was tagged. This happens because car names are very diverse, and this makes it difficult to distinguish them from other proper names.

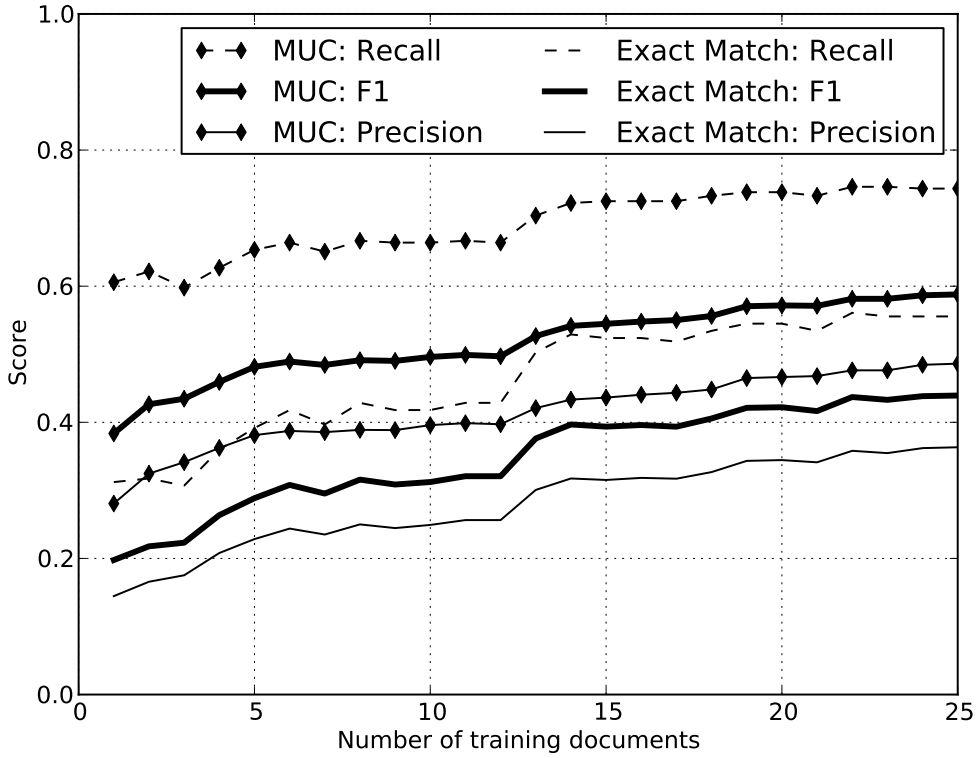

Figure 6.1: Recognition performance improves with an increasing number of training documents.

		real					
		Place	**Car**	**Event**	**Organization**	**Person**	**OTHER**
predicted	**Place**	21	4	1	3	3	14
	Car	2	56	1	29	11	57
	Event	0	0	7	0	0	5
	Organization	1	6	0	36	4	16
	Person	0	0	0	1	6	5

Table 6.2: Confusion Matrix for 25 training documents

6.4 Conclusion

In this chapter, we have evaluated the frameworks which have been developed during this thesis. As the framework is comprised of smaller individual and separate units, we evaluated these one-by-one. The RDF and RDFa functionality was checked against requirements. Furthermore, the external reference service was evaluated in two pieces. First, its functionality was compared with alternative products based on a use-case study. Second, an evaluation of the *open world* NER Palladian has been done, as this is a key component of the infrastructure.

We have shown that most principles we set out to follow are fulfilled. Additionally, this evaluation has unveiled some weak spots, and thus possible areas of future research.

CHAPTER

SEVEN

FUTURE WORK

In this chapter, we outline possible future improvements and research topics. Additionally, we show how the big vision of Linked Data in a corporate environment could become reality.

7.1 Implementation Improvements

While RDF generation is extensible and flexible, it is currently lacking generators for ordered lists (RDF sequences and collections). When this becomes widespread in practice, that feature shall be implemented.

Furthermore, as soon as web browsers extract and show post-processed RDFa data to end-users, it might become necessary to revisit the RDFa output and implementing a heuristic approach for adding the RDFa subjects to the correct nodes in the document tree.

The named entity recognition process can also be improved: It would be very interesting to extract the "Wikipedia NER" into a web service, and use this for improving detection accuracy. Because there is a deterministic one-to-one mapping between DBpedia resources and Wikipedia pages, we can directly take the output of the Wikipedia NER, and add Linked Data URIs to the found entities. This would merge NER and linkification steps, doing the same with potentially less network requests, enabling a faster response time and maybe even a higher detection accuracy. Of course the above would need to be evaluated, and until then remains an open question.

As the system shall be used by developers who are not proficient in technologies of the Semantic Web, it should be able to catch beginner's errors and inconsistencies. For that, the Semantifier should be extended to check the generated RDF for consistency and typing errors based on the Eyeball[1] framework.

7.2 Linkification Process Evaluation

The linkification process can still be adjusted and fine-tuned, adding more data sources like GeoNames[2] or Freebase[3]. In recent days, other projects have also implemented linkification using a number of different data sources. For example RDFaCE[4] is only adding a URI when m out of n

[1] http://openjena.org/Eyeball/
[2] http://geonames.org
[3] http://freebase.com
[4] http://rdface.aksw.org

data sources agree on a specific URI for a certain entity. Thus, it makes sense to evaluate these different linkification approaches.

While there exists research for Linkification against Wikipedia (and thus also DBpedia), we have found surprisingly little published information on the topic of generic linkification against arbitrary Linked Data sources. Next to NER systems, we believe that such algorithms are a key building block for enriching non-semantic data and integrating it with the Semantic Web.

7.3 Extracting Relations from Continuous Texts

Right now, we can detect entities of various types inside texts, as it is shown with the following example sentence:

> While visiting Paris, Barack Obama met with the French Prime Minister.

Through the NER and linkification process, the system is able to find the entities `Paris`, `Barack Obama`, `French Prime Minister` and resolve URIs for them.

However, there is still more knowledge embedded in the text, putting *entities into relation* with each other: It would be nice if the following facts could be derived from the text:

1. Barack Obama visited Paris.
2. Barack Obama had a meeting with the French Prime Minister. They met in Paris.

If the system was able to extract the entity relations from the text, we could query them using RDF as well – potentially giving completely new insights into data.

One possibility to implement this would be to query a large common-world knowledge database like DBpedia or Freebase for every matching entity pair, asking for RDF predicates which connect both entities together, as it is done in the *RelFinder*[5]. Furthermore, the sentence structure of the incoming text needs to be analyzed, using natural language processing algorithms. When the predicate has been identified in the sentence which connects the two entities, some heuristic has to calculate the best-matching RDF predicate from that. With such an algorithm, simple relations (like fact 1 in the example above) could be learned.

Another more complex scenario appears in fact 2, where the system would need to identify an *intermediate entity*, in this case a `Meeting`. After that, it needs to connect the various entities to the intermediate entity. How to implement this remains an open question.

7.4 The Big Picture - Data-Driven Companies

Now, after we have shown in this thesis which steps we can take to export data in a universal format, we want to give some vision on what could be done with this information. Instead of doing that on a web scale, we will show how companies of all sizes can benefit from a semantic information architecture.

[5] http://www.visualdataweb.org/relfinder.php

7.4.1 Information Fragmentation Is A Problem

Every company which works in the IT sector uses a multitude of different tools in their processes. Usually, there exists an e-mail system, a shared calendar and address book. Furthermore, many companies run issue trackers, customer relationship management (CRM) software and use specialized invoicing tools. Each tool is highly specialized and stores a certain part of the company's data.

This has two effects: First, the company employees have to use all kinds of different tools to get a certain task done. Second, the fragmentation immensely complicated to get a *global view*, aggregating information across tool boundaries.

Because of the above drawbacks, people often want to use systems which integrate all kinds of different tools into one big tool (we will call that a *mega-tool*). The SAP suite is such an example of a mega-tool providing many of the above tools. However, these solutions often are very expensive, and it often turns out that a specialized product is much better in solving the task it is supposed to solve than a mega-tool which also contains this functionality.

That is understandable when thinking about the development process of a product: If a product is developed to solve just one problem, more time can be spent about how to solve this problem effectively. On the other hand, if a product is developed which contains thousands of features, less time usually is spent in optimizing each feature.

7.4.2 The Vision

Our vision is a third alternative to either using many small tools, suffering from data fragmentation, or a mega-tool, suffering from lack of usability: *Integration of data from multiple specialized tools using the Semantic Web technologies.*

Because the Semantic Web technology stack already defines how information is modelled and stored in a distributed way, this frees us from many decisions: We do not need to define a common data integration format which is extensible and distributed, we only need to embrace RDF. We do not need to define a query language for our distributed data, as we can use SPARQL.

Our vision is shown in Figure 7.1: At the center of the IT infrastructure of a company is a *triple store* which stores the aggregated information from the different systems as RDF. This triple store should also support *fulltext searching*, in order to make freeform search queries across all the information of a company possible. A triple store containing such a full-text index is BigData[6].

This triple store is filled by the different applications in the company, each being specialized on a particular problem domain. That is where the work of this diploma thesis is embedded: Through the FLOW3 Semantic Framework, sending all the information as RDF to the triple store becomes really easy.

By using the *Semantifier* service, it is even possible to cross-link information with other data sources in a way which is transparent to the FLOW3 application.

Other applications, which are not based on FLOW3, need other techniques in sending their data to the triple store. Some tools which export relational database contents (such as D2R) might be helpful in this context.

[6] http://www.bigdata.com

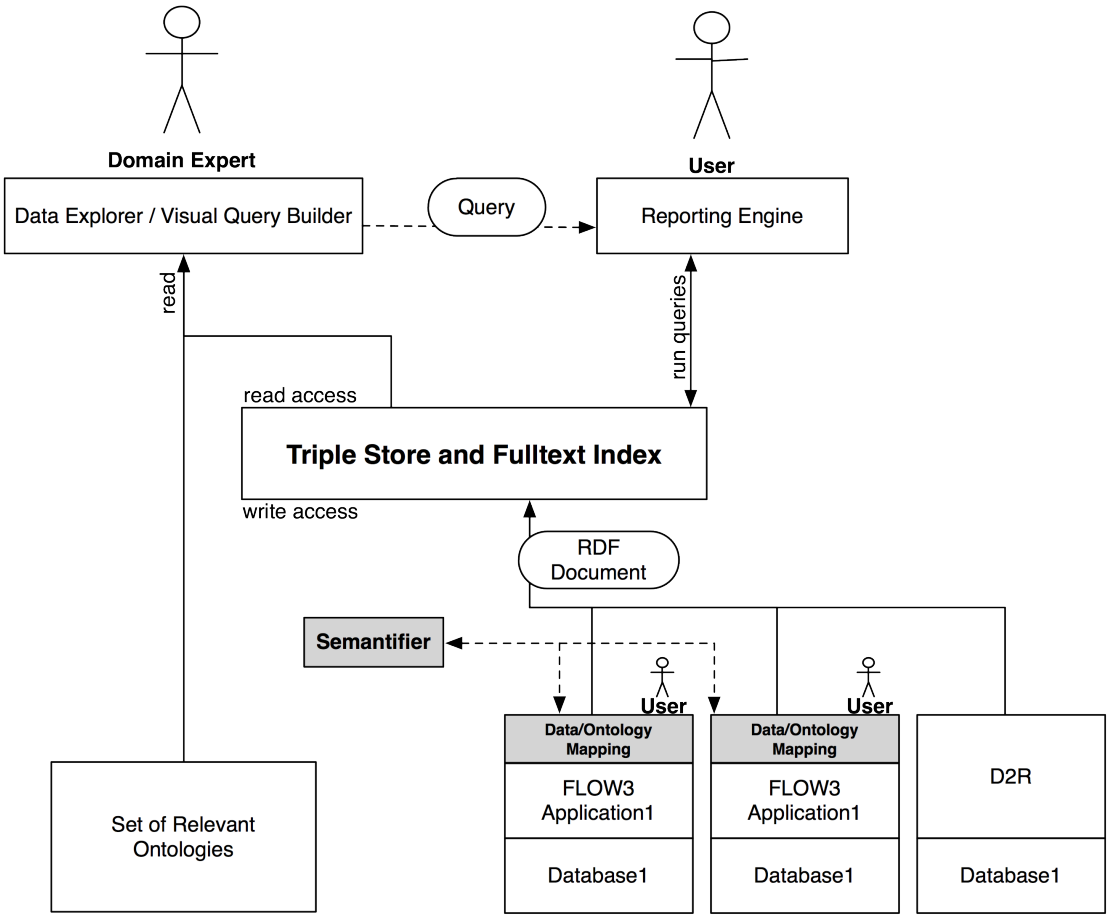

Figure 7.1: Vision of Enterprise Data Integration. The components with the gray background have been developed for this thesis.

As soon as most of the information is inside the triple store, we can start formulating SPARQL queries across tool boundaries.

This brings us to the question on how we can effectively write SPARQL queries. When having a detailed knowledge on the structure of the data in the store, it is easy to write a query which returns the desired information. However, when this detailed knowledge is missing, it is very hard to build a SPARQL query which returns the desired information.

Because we integrate data from many different applications, we can assume that a lot of different ontologies are used to express information. That makes it hard to reason about information structure, and we might need some different tool which guides us in formulating SPARQL queries: *The Data Explorer* or *Visual Query Builder*.

The *Data Explorer* should analyze both the ontologies and the data inside the triple store. It should present a web-based graphical interface which helps to formulate SPARQL queries.

While some queries shoud be only run once (*ad-hoc queries*), others shall be repeated regularily (*scheduled queries*). For the latter, a *Reporting Engine* should run queries in a defined interval, generating HTML or PDF reports and sending these to the relevant stakeholders.

7.4.3 Predictable Behavior

In order to archieve predictable behavior when so many different tools are involved, it is important to define some guidelines which shall never be violated:

- The authoritative data source is *always* the specialized application. It must be possible to re-fill the triple store from scratch without loss of information.
- As a consequence, data aggregation applications are not allowed to write to the triple store.

7.4.4 Many Open Questions

While the general vision is clear, there are of course still many open questions which have to be investigated further:

- It remains an open question how the *Visual Query Builder* should look like.
- How to connect SPARQL queries with freeform text queries?
- When some information shall only be visible for certain people, where shall this access control be implemented?

Despite the many open questions, an important step has already been solved: How to export the data from many proprietary systems into one unified RDF triple store? This question has been answered by this work.

7.5 Conclusion

In this chapter, we wanted to show possible directions for future research. Next to building upon the foundations laid in this work, we introduced our vision of companies which solve their enterprise data integration tasks by Semantic Web technologies.

We hope that this is an important step to unleashing the full power of Linked Data, finally bringing this technology to the masses.

CHAPTER

EIGHT

SUMMARY

We started this work by discussing the benefits of Linked Data and the Semantic Web, discovering that these technologies can solve the information integration problem very effectively. As Linked Data frees developers from thinking about syntax, they can fully concentrate on building a semantically rich language for the particular problem they set out to solve. Through the consistent use of URIs to identify resouces on all levels of the Semantic Web technology stack, this effectively creates a *universal language* for information.

Without a doubt, the vision of an Internet connected through Linked Data is very promising. However, up to now, only very few data sources in the Internet participate in this Linked Data Cloud. While there are millions of websites which publish information, only a few hundred participate in the Semantic Web.

In our opinion, this huge discrepancy appeared because participation in the Linked Data movement is far from easy: Developers need to learn a lot of new technologies and have to take many decisions along their way. We believe that by enabling *web application developers* to participate easily in the Linked Data cloud, the amount of data available as Linked Data can literally explode. When the amount of data grows quickly, completely new applications based on this data will appear, effectively starting a new era of the web.

In this work, we have taken a state-of-the-art web development framework based around the concepts of Domain-Driven Design, and added all the needed capabilities to export semantically enriched data. Especially we focussed on great usability for the developer and the end-user, making it possible to publish standards-compliant semantic data with very little knowledge. We will now focus on the steps we needed to take in order to archive this goal.

First, we made it possible to export all data inside the framework as RDF, by only specifying a mapping between properties of the model and ontologies. By that, we are already able to follow most of the conventions and best-practices for Linked Data publishing (shown in Chapter 6.1).

Second, we realized that an entry point for browsers (such as Semantic Radar for Firefox[1]) is needed: They must be able to detect that a page contains semantic information. Thus, it was a natural step to implement RDFa support, linking the document-based internet together with the Semantic Web. It turned out that we were able to re-use the information available from the RDF generation: This feature does not need any specific settings or configuration, making it very convenient for the developer.

Third, we wanted to be able to add *outgoing links* to other Linked Data participants. We argued that in order to change an entity string such as `Paris` into a Linked Data URI, we have to *add*

[1] https://addons.mozilla.org/en-US/firefox/addon/semantic-radar/

information to the system, as the ambiguity of the entity string has to be resolved. This discovery had far-reaching consequences: We needed a *user interface component* which helps the end-user to disambigurate an entity. Furthermore, we wanted to support the user whereever possible, making good suggestions about the disambiguration choices the user can take. This also includes that the framework needs to learn from the user's behavior everytime it has done a wrong choice.

While the first two parts are rather stable, we are leaving the established paths with the 3rd part of the framework. Because of this, we will highlight the results especially in this area.

8.1 Linkification of Entities and Continuous Texts

While finding entity types in a text is a well-researched topic (called *Named Entity Recognition*), finding URIs for entities is not. We have built a framework which combines NER systems with heuristics to *linkify* the found entities with Linked Data URIs: The *Semantifier* Web Service.

Besides finding entities in a continuous text, we discovered another relevant use-case for linkification: In web applications, it is often known beforehand that a certain property contains a specific entity type such as a person's name or a location. These properties are often stored as simple strings, as they are not needed for further processing in many cases. We wanted to linkify these properties as well: With the Semantifier, we are able to find Linked Data URIs for such entity types, effectively adding information which can be used for more advanced data processing lateron.

On the client side, we have developed a *user interface component* which is used to disambigurate the possible meanings of an entity. As we cannot forsee which web applications will embed this component, we implemented it unobstrusively, not modifying the user interface at all. Only when the user activates the relevant text field, we show the disambiguration controls.

The last functionality we want to highlight in this area is the *learning* subcomponent. Because we wanted to adapt the system to learn new entity types and new linkification URIs, we embedded the Palladian NER into the Semantifier. Palladian has a unique learning mechanism which is very much in sync with the open world assumption, only learning positive facts. In Chapter 6.3.3 we have evaluated the learning performance for a specific real-world use case.

The learning subcomponent also has profound implications on the implementation architecture: We have shown the benefits of using a centralized *Semantifier* web service in an organization instead of embedding it directly inside each application.

8.2 Final Remarks

Besides developing the concepts and the implementation for easy Linked Data publishing, we also thought a lot about the long-term vision of the Semantic Web. Besides its usage on a global scale, we also developed a vision for using Linked Data inside organizations to solve the data integration problem on a fundamental level (Chapter 7.4).

We hope that this work helps in spreading the word about Linked Data, making participation for web developers a lot easier and fostering the Linked Data ecosystem.

APPENDIX

A

PREFIX MAPPINGS USED

The following table shows the prefix mappings used in this work.

Table A.1: Prefix Mappings

Prefix Mapping	URI
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
foaf	http://xmlns.com/foaf/0.1/
purl	http://purl.org/
geonames	http://sws.geonames.org/
cv	http://captsolo.net/semweb/resume/cv.rdfs
geo	http://www.w3.org/2003/01/geo/wgs84_pos#
sioc	http://rdfs.org/sioc/ns#
sioctypes	http://rdfs.org/sioc/types#
dcterms	http://purl.org/dc/terms/
annot	http://sandstorm-media.de/ns/2011/annotation#
vcard	http://www.w3.org/2006/vcard/ns#
v	http://nwalsh.com/rdf/vCard#

BIBLIOGRAPHY

[RDFpub08] W3C Working Group Note: Best Practice Recipes for Publishing RDF Vocabularies, 28 August 2008, http://www.w3.org/TR/swbp-vocab-pub/

[saiedian99] Saiedian, R. Dale. 1999. Requirements engineering: making the connection between the software developer and customer. Information and Software Technology, Volume 42, Issue 6, Pages 419-428.

[rau10] Jochen Rau, Sebastian Kurfürst. 2010. Zukunftssichere TYPO3-Extensions mit Extbase und Fluid. O'Reilly. ISBN 978-3-89721-965-6. Kapitel 2.2

[evans04] Eric Evans. 2004. Domain-Driven Design. Tackling Complexity in the Heart of Software. Addison-Wesley. ISBN 0-321-12521-5

[leibnitz08] Stanford Encyclobedia of Philosophy, Fall 2008 Edition: The Identity of Indiscernibles, http://plato.stanford.edu/archives/fall2008/entries/identity-indiscernible/

[curie10] W3C Working Group Note: CURIE Syntax 1.0, 16 December 2010, http://www.w3.org/TR/curie/

[heath11] Tom Heath, Christian Bizer. 2011. Linked Data: Evolving the Web into a Global Data Space. Morgan & Claypool. ISBN 978-1-60845-430-3. http://linkeddatabook.com/editions/1.0/ - Chapter 2.3: Making URIs Dereferenceable

[csomai08] Csomai, Mihalcea. 2008. Linking Documents to Encyclopedic Knowledge. In IEEE Intelligent Systems 23 (September): 34-41. doi:10.1109/MIS.2008.86.

[milne2008] David Milne, Ian H. Witten. 2008. Learning to link with wikipedia. In CIKM '08 Proceeding of the 17th ACM conference on Information and knowledge management. ACM Press. doi:10.1145/1458082.1458150

[kulkarni2009] Sayali Kulkarni, Amit Singh, Ganesh Ramakrishnan, Soumen Chakrabarti. 2009. Collective annotation of Wikipedia entities in web text. In KDD '09 Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM Press. doi:10.1145/1557019.1557073.

[ferragina2010] Paolo Ferragina, Ugo Scaiella. 2010. TAGME: on-the-fly annotation of short text fragments (by wikipedia entities). In CIKM '10 Proceedings of the 19th ACM international conference on Information and knowledge management, 1625. ACM Press. doi:10.1145/1871437.1871689.

[hachey2011] Ben Hachey, Will Radford, James R. Curran. 2011. Graph-based Named Entity Linking with Wikipedia. In The 12th International Conference on Web Information System Engineering, Sydney, NSW, Australia.

[heese2010] Ralf Heese, Markus Luczak-Rösch, Adrian Paschke, Radoslaw Oldakowski and Olga Streibel, “One Click Annotation”, 6th Workshop on Scripting and Development for the Semantic Web, colocated with ESWC 2010, Crete, Greece, May 31, 2010. http://loomp.org.

[khalili2011] Ali Khalili and Sören Auer. The RDFa Content Editor –From WYSIWYG to WYSIWYM. 2011. Universität Leipzig, Institut für Informatik, AKSW.