UNIVERSITY OF OSLO
Department of Informatics

A JavaScript API for accessing Semantic Web

Master Thesis

Arne Hassel

Spring 2012

Acknowledgments

I want to thank Kjetil Kjernsmo and Martin Giese, my supervisors who are both part of the research group Logic and Intelligent Data. They have contributed with invaluable advice along my study, and with great enthusiasm for the research field.

I also want to thank my family, especially my parents, who have supported me all my life, and patiently allowed me to take my time. I hope I make you proud.

And finally, I owe a great deal to my beloved Veronika, with whom I share love eternal.

Abstract

The thesis describes and discusses the development of the framework GraphiteJS (Graphite), an implementation of a JavaScript (JS) Application Programming Interface (API) for accessing Semantic Web (SW). It outlines the necessary background in terms of technology, standards and tools, and how this becomes a part of the framework. Software Design patterns (SDPs) are a central tool to help its design, and emphasis is put on splitting the functionality into separate modules that can be reused by other works within JS. I conclude that modularization is a necessary feature to support in works that try to take on SW, as it requires a lot of components that need to collaborate in a multitude of ways. As such, the prospect of a singular framework being the de facto tool for JS developers wanting to access SW seems dim, and the better approach would be to create modules that can be reused by several frameworks, the result being that developers can pick and choose from a variety of approaches.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The Semantic Web (SW) is a many-faced entity, a colossal structure of standards and resources. It is also an idea shared by a multitude of communities, a concept of structured information, and an abstraction of knowledge. It is a mixture of technologies, created over a decade of work by professionals. Academia researches it, businesses try to create common ground with it, and visionaries preach of its promises: A richer world, where computer-driven agents find, process, and act upon information tailored for our need.

At the center of the SW we have the World Wide Web Consortium (W3C), led by Tim Berners-Lee. Berners-Lee is perhaps more famous for his invention, the World Wide Web (WWW), and he is also the one who coined the phrase Semantic Web. It is in his writings of Design Issues we find the essence of SW, namely the sentence "The Semantic Web is a web of data, in some ways like a global database" [4].

The web of data has been in the making since the late 1990s, but in terms of traction there is still much to be done. Some complain it is still very much an academic affair, while others complain of the lack of interest from the developing community.

This master thesis has taken the approach to look at the gap between SW and the developing community by trying to construct a framework that offers tools to access the SW. It has been written in and for JavaScript (JS), as it is a programming language of the web, and the time seems right.

JS can relate to SWs struggles for traction. For long time it was ridiculed by developers, saying it was a silly language that merely created fancy effects on web pages, but not doing anything useful. Douglas Crockford, an evangelist of JS, has called JS the world's most misunderstood language [11]. And if the name and its syntax was not confusing enough, the browsers with their differing implementations were not making it any easier.

There were, and still are, many reasons to why people get confused by JS. But in the mid-2000s, efforts were made to make JS more accessible to developers. Prototype, MooTools, and jQuery are all frameworks that promises Application Programming Interfaces (APIs) for easier, cross-browser access to the power within JS. And it worked! Readily manipulation of the Document Object Model (DOM), asynchronous fetching of resources with Asynchronous JavaScript and XML (AJAX), and the increasing effort of making JS into a full-fledged server-side programming language, are making JS a powerful and fun tool for developers to work with.

It is this fertile ground the work of this master thesis is trying to tap into. GraphiteJS

(Graphite) is an Asynchronous Module Definition (AMD)-based framework (described in section 2.2.8.3) written in JS that sports a modularized API to fetch resources in the SW, process it and output in useful way for JS-developers. Frameworks typically serve to implement (larger-scale) components, and are implemented using (smaller-scale) classes [25]. This description of frameworks suits my implementation well, as the work in large part will consist of defining smaller components and have them collaborate effectively for a higher-level purpose.

This master thesis will describe the work and choices made during the implementation of Graphite. It is divided into three parts. The first consists of the underlying theory and constraints in technology (chapter 2), how this fits into the scope of this thesis (chapter 3). The second part describes the implementation, and starts by explaining which tools and third party libraries I made use of (chapter 4 and 5 respectively). It continues with an extensive presentation of the framework itself (chapter 6) and a demo I constructed to demonstrate some of the frameworks' capabilities (chapter 7). Finally, in the third part I offer a discussion of the work (chapter 8), and a conclusion of the matter (chapter 9).

I hope to contribute to the developing community of SW and JS in two ways; through the thesis, to showcase what is already available and present some research and thoughts of my own, and through the framework, in the hopes that it contributes to the evolution of handling SW in JS.

Part I

Foundations

Chapter 2

Background

This chapter will describe the technologies, standards, and theories that Graphite has been build upon.

2.1 Semantic Web (SW)

SW represents a multitude of standards and technologies, and seeing the whole picture may not be so easy to grasp. A perhaps fitting metaphor is the story of the elephant and the blind men. It is a story made famous by the poet John Godfrey Saxe, and tells the story of how six men tried to describe an elephant. Depending on which part they touched, each described the elephant differently. One approached its side, and called it a wall. Another touched the tusk, and surely it had to be a spear. The third took hold on the tusk, and spoke of how it resembled a snake. The fourth reached out for its knee, and stated it had to be like a tree. The fifth touched the ear, and meant it had to be like a fan. Finally, the last one had grabbed its tail, and stated how it had to be like a rope [28]. As such, what are some of the descriptions we have of SW?

- A web of data [4].
- An extension of WWW [20].
- A `killer` app [8].
- W3C's vision of the Web of linked data [37].

The list above are some of the descriptions in literature, and they are all true. Other aspects of SW is the set of standards it sports (e.g. Resource Description Framework (RDF), Resource Description Framework Scheme (RDFS), Web Ontology Language (OWL), and SPARQL Protocol and RDF Query Language (SPARQL)), technological foundations (e.g. Linked Data (LD)), applicabilities (e.g. use of Linked Open Data (LOD) amongst governments), social consequences (democratizing data), limitations (e.g. Anyone can say Anything about Anything (AAA)), and more.

2.1.1 Resource Description Framework (RDF)

At the heart of SW lies RDF. It is a formalized data model that asserts information with statements that together naturally form a directed graph. Each statement consists of one subject, one predicate, and one object, and are hence often called a triple. The three elements have meanings that are analogous to their meaning in normal English grammar [19, p. 68-69], i.e. the subject in a statement is the entity which that statement states something about. An example of statements and how they are represented as a graph is showed in figure 2.1. It illustrates that the subject `"Arne"` is related to the object `"Kjetil"` by the predicate `"knows"`, and to the object `"Hassel"` by the predicate `"familyName"`.

- Arne knows Kjetil.
- Arne has last name Hassel.

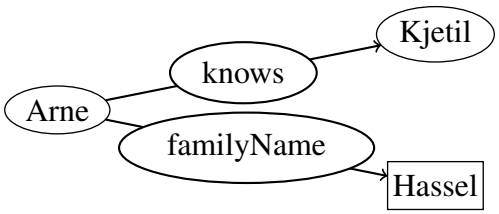

Figure 2.1: A directed graph.

You might have noticed that the two objects have different shapes, one being a circle (like the subject), and the other being a rectangle. That is to show that `"Hassel"` is a literal. Literals are concrete data values, like numbers and strings, and cannot be the subjects of statements, only the objects [19, p. 69].

The circles on the other hand, are known as resources, and can represent anything that can be named. As RDF is optimized for distribution of data on WWW, the resources are represented with Internationalized Resource Identifiifiers (IRIs) (IRI is an extension of Unified Resource Identifier (URI), and is explained in section 2.1.4.2).

IRIs are usually declared into namespaces, to make terms more human-readable (e.g. resources in the namespace `http://example.org/` could be prefixed `ex`). If we look at figure 2.1, we have two resources, namely `Arne` and `Kjetil`. To make these available as LD, we could assign them into the namespace `ex`, writing them respectively as `ex:Arne` and `ex:Kjetil`.

The basic syntax in RDF has a relatively minimal set of terms. It enables typing, reification, various types of containers (bags, sequences, and alternatives), and assigning of language or data type to a literal [2]. Its power lies in its extensibility by URI-based vocabularies [21]. By sharing vocabularies as standards between software applications, you can easier exchange data.

With this in mind, we see that figure 2.1 is faulty, and we turn to figure 2.2 to see a correct representation (using the vocabulary Friend of a Friend (FOAF), prefixed `foaf`, for the properties).

Not all resources are given IRIs though. The exception to the rule are Blank Nodes (BNs), which represent resources that have not any separate form of identification [21], either because they cannot be named, or it is neither possible nor necessary at the time of modeling. These

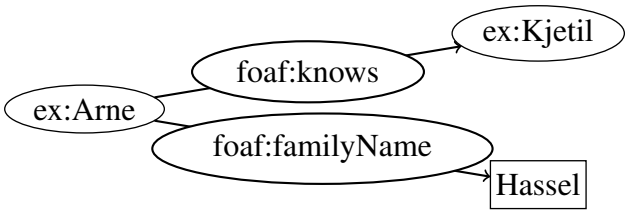

Figure 2.2: Statements from figure 2.1 correctly represented with IRIs.

resources are not designed to link data, but to model relations of resources that are given IRIs.

An example of modeling BN is given in figure 2.3, where I have modeled that `ex:Arne` has a friend, who we do not know anything about except his nicknames, `Bjarne` and `Buddy`.

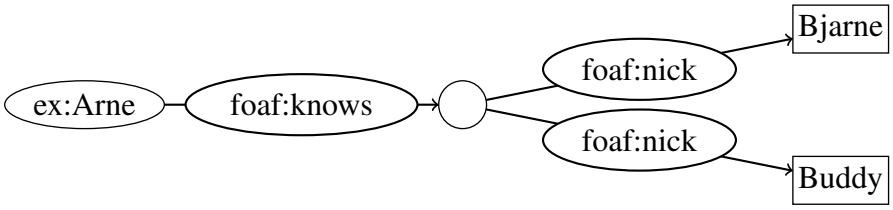

Figure 2.3: A graph containing a BN.

The figures 2.2 and 2.3 are examples of the form of visualization we will have of graphs in RDF.

2.1.2 Resource Description Framework Scheme (RDFS)

RDFS is an extension in form of vocabulary that extends the semantic expressiveness of RDF. But RDFS is not a vocabulary in the traditional sense that it covers any topic-specific domain [20, p. 46]. It is designed to extend the semantic capabilities of RDF, and in that sense it can be regarded as a meta-vocabulary.

The perhaps most important feature of RDFS is its ability to support taxonomies. It empowers the use of `rdf:type` by introducing `rdfs:Class`, in effect enabling classification. The properties `rdfs:range`, `rdfs:domain`, `rdfs:subClassOf`, and `rdfs:subPropertyOf` further extends this feature.

It also builds on the reification-properties of RDF, by instantiating `rdf:Statement` as a `rdfs:Class`. It continues by clarifying the semantics of `rdf:subject`, `rdf:predicate`, and `rdf:object` by instantiating them as `rdf:Property`, and in terms of entailment (explained in section 2.1.7) ties together with `rdfs:range` and `rdfs:domain`.

Another extension is the clarification of containers by introducing the class `rdfs:Container` and the property `rdfs:containerMembershipProperty`, which is an `rdfs:subPropertyOf` of the `rdfs:member` [10].

Finally, it introduces the utility properties `rdfs:seeAlso` and `rdfs:isDefinedBy`. The former represents resources that might provide additional information about the subject resource, while the latter gives the resource which defines a given subject. It also clarifies the use of rdf:value, to encourage its use in common idioms [10].

2.1.3 Web Ontology Language (OWL)

In the same way RDFS is an extension to RDF in order to express richer semantics, OWL is an extension to RDFS to express even richer semantics. It does so by introducing vocabularies that are based on formal logic, and aims to describe relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer type of properties, characteristics of properties (e.g. symmetry), and enumerated classes [31, sec. 1.2].

As of this writing, OWL exists in two versions: The version recommended by W3C in 2004 (often known as OWL 1), and The OWL 2 Web Ontology Language (OWL 2), which became recommended in 2009. OWL 2 is an extension and revision of OWL 1, and is backward compatible for all intents and purposes [33].

OWL 1 features three sublanguages/profiles[1]. These are, with complexity in increasing order (all quoted from OWL Features [31]):

1. OWL Lite: Supports classification hierarchy and simple constraints (e.g. only cardinality values of 0 and 1).

2. OWL Description Logics (OWL DL): Maximum expressiveness while retaining computational completeness and decidability.

3. OWL Full: Maximum expressiveness and the full syntactic freedom of RDF, but with no computational guarantees.

OWL 2 also make a distinction with DL and Full. It does not list a Lite profile, but all OWL Lite ontologies are OWL 2 ontologies, so OWL Lite can be viewed as a profile of OWL 2 [34]. In addition, DL has three sublanguages that are not disjunct, and also does not cover the complete OWL 2 DL. These sublanguages are (all quoted from OWL 2 Profiles [34]):

1. OWL Existential Language (OWL EL): Designed to be used with ontologies that contain very large numbers of either properties or classes.

2. OWL Query Language (OWL QL): Aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task.

3. OWL Rule Language (OWL RL): Aimed at applications that require scalable reasoning without sacrificing too much expressive power.

To go through all differences between OWL 1 and OWL 2 would be beyond the scope of this thesis, but suffice to say is that OWL 2 is designed to be backward compatible with OWL 1, and the sublanguages OWL provides as a whole increases the reasoning capabilities of SW.

[1]This might be wrong: `http://www.w3.org/TR/2004/REC-owl-features-20040210/#s1.3` states that OWL 1 has three sublanguages, while `http://www.w3.org/TR/2009/REC-owl2-new-features-20091027/#Backward_Compatibility` claims that it only has one. But for the purposes of this thesis, I work with three sublanguages.

2.1.4 Linked Data (LD)

A cornerstone of RDF is that all identifications (that is, except BNs) are IRIs. In this way, machines can browse the web for relevant resources, much like you browse the web through hyperlinks. This design feature makes RDF adhere to LD, which is a term that refers to a set of best practices for publishing and connecting structured data on the web [9].

Tim Berners-Lee have in his article about LD[2] outlined four "rules" for publishing data on WWW [6]:

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.

These have become known as the "Linked Data principles", and provide a basic recipe for publishing and connecting data using the infrastructure of the WWW while adhering to its architecture and standards [9].

LD are reliant on two web-technologies, namely IRIs and Hypertext Transfer Protocol (HTTP). Using the two of them you can fetch any resource addressed by an IRI that uses the HTTP-scheme. When combining this with RDF, LD builds on the general architecture of the Web [30].

The Web of Data can therefore be seen as an additional layer that is tightly interwoven with the classic document Web and has many of the same properties [9]:

- The Web of Data is generic and can contain any type of data.
- Anyone can publish data to the Web of Data.
- Data publishers are not constrained in choice of vocabularies with which to represent data.
- Entities are connected by RDF links, creating a global data graph that spans data sources and enables the discovery of new data sources.

2.1.4.1 Linked Open Data (LOD)

Based on the notion of LD, there is a movement to publish data on WWW as LOD. Especially toward governmental institutions there is now an increasing trend of opening data[3].

To encourage this trend, Tim Berners-Lee published a star rating system. On a scale from one to five stars, it rates how well the given dataset is in becoming open. It is incremental,

[2] `http://www.w3.org/DesignIssues/LinkedData.html`

[3] Examples of this are platforms such as the UK initiative to open governmental data (`http://data.gov.uk/`), US' approach of the same (`http://www.data.gov/`), and Norway's parliament opening of its databases through Stortingets datatjeneste (`http://data.stortinget.no/`). You also have other non-governmental organizations, such as Parliamentary Monitoring Organizations (PMOs).

meaning that the dataset needs to be have one star before it can be given two. One star is given if your data is available on WWW with an open license. Two stars means that your data is available in machine-readable structure, and is valid for another star if the structure is a non-proprietary format (e.g. Comma-Separated Values (CSV) instead of Excel). Four stars are given if your the data is identified by using open standards from W3C (e.g. RDF and SPARQL). The last star means that your data also link to other people's data, in order to provide context [6].

Figure 2.4 shows the linking open data cloud diagram. It illustrates to some extent the magnitude of data that are linked as of yet[4].

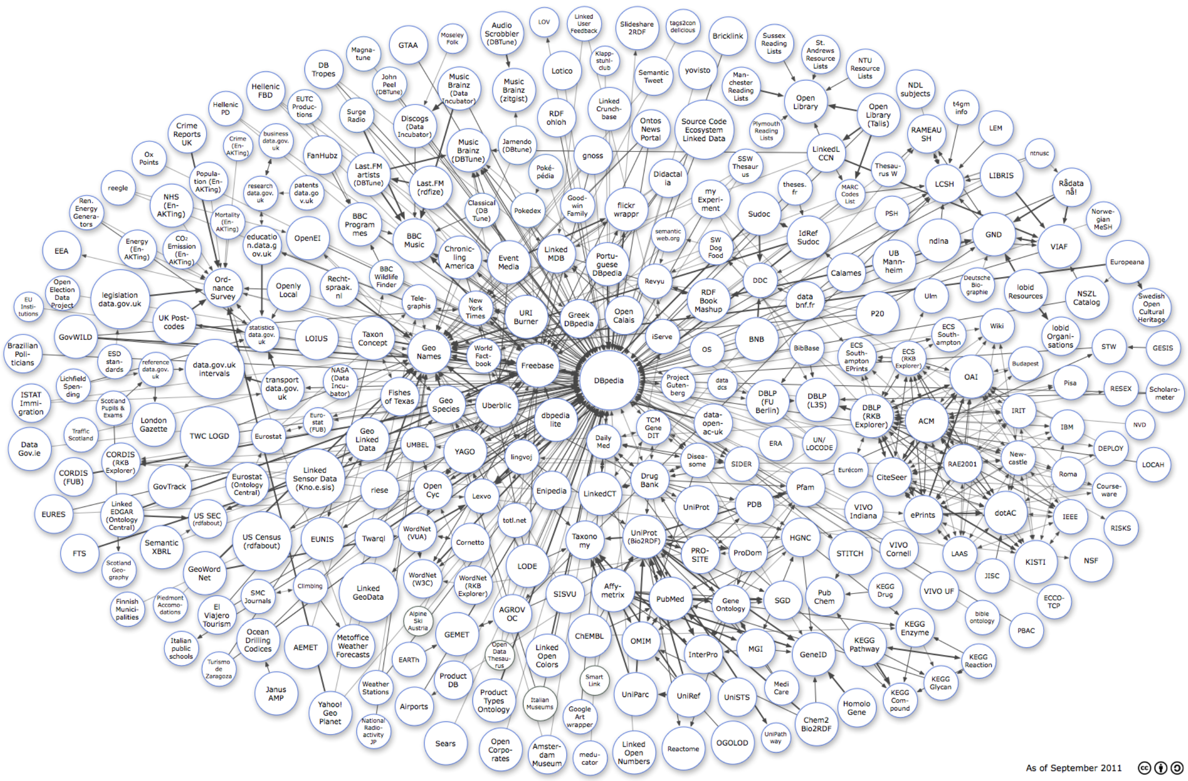

Figure 2.4: Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch. `http://lod-cloud.net/`

2.1.4.2 URL VS URI VS IRI

Throughout this thesis you will read the terms Unified Resource Locators (URLs), URIs, and IRIs being used interchangeably. I strive to use IRI as it is supported by RDF, but in some cases it is more appropriate to use the others because of the texts they reference.

URLs and URIs are the most commonly used terms. The former denotes dereferenceable resources on WWW, while the latter is a generalization that can denote anything, even resources not on WWW. But URIs are limited to the character-encoding scheme American Standard Code for Information Interchange (ASCII), and as such IRI has been introduced to solve this problem.

URI have the form `scheme:[//authority]path[?query][#fragment]`, where the parts in brackets are optional. The list below explains the different terms (shortened versions

[4] Well, as of 19th of September 2011, when the diagram was last updated.

of the ones offered by Hitzler et.al. [20, p. 23]. The explanations are equally valid for URL and IRI.

- `scheme`: The scheme classify the type of URI, and may also provide additional information on how to handle URIs in applications.
- `authority`: An authority is the provider of content, and may provide user and port details (e.g. arne@semanticweb.com, semanticweb.com:80).
- `path`: The path is the main part of many URIs, though it is possible to use empty paths, e.g., in email addresses. Paths can be organized hierarchically using `/` as separator.
- `query`: The query can be recognized with the preceding `?`, and are typically used for providing parameters.
- `fragment`: Fragments provide an additional level of identifying resources, and are recognized by the preceding `#`.

2.1.5 Serializations

RDF in itself offers no serialization of the graph it represents. But there are many serializations available, and more are coming as of this writing.

There are some considerations to take when choosing a serialization for a given project. One consideration is the ease for humans to read the syntax, which is very useful if you want to verify how your data is related. Another is the availability of tools to process the serialization. RDF/XML, for example, is based on Extensible Markup Language (XML), and as such there are many tools that can deserialize it. Terse RDF Triple Language (Turtle) on the other hand is specific for RDF, and may not be as easy to deserialize. But most will agree that the latter is much easier to read and understand than the former.

2.1.5.1 RDF/XML

RDF/XML has been recommended by W3C to represent RDF since the beginning of SW [21, sec. 2.2.4]. As the name suggests, RDF/XML is based on the markup language XML. XML may not be as humanly accessible as some of the other serializations, but it is the most commonly used, probably because of the readily available software to process XML-documents.

XML is tree-based, which means some considerations need to be taken when we serialize graphs. Each statement will have the subject as the root, followed by the predicate, and then the object. As an example of this we have listing 2.1, which shows a serialization of figure 2.2.

Another reason for XML being chosen as the default serialization was that it was readily available at the time RDF was being standardized. Figure 2.5 shows a timeline of the development of XML and SW.

Listing 2.1 shows that we have namespaces in XML through the attribute rdf:xmlns. But we cannot use namespaces in values given to attributes (i.e. we have to write `rdf:about="http://example.org/Arne"` instead of `rdf:about="ex:Arne"`). This adds to the notion that XML-documents are bigger than what we need to serialize RDF.

Listing 2.1: Serialization of figure 2.2 into RDF/XML.

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/">

<rdf:Description rdf:about="http://example.org/Arne">
  <foaf:knows>
    <rdf:Description rdf:about="http://example.org/Kjetil">
    </rdf:Description>
  </foaf:knows>
  <foaf:familyName>Hassel</foaf:familyName>
</rdf:Description>

</rdf:RDF>
```

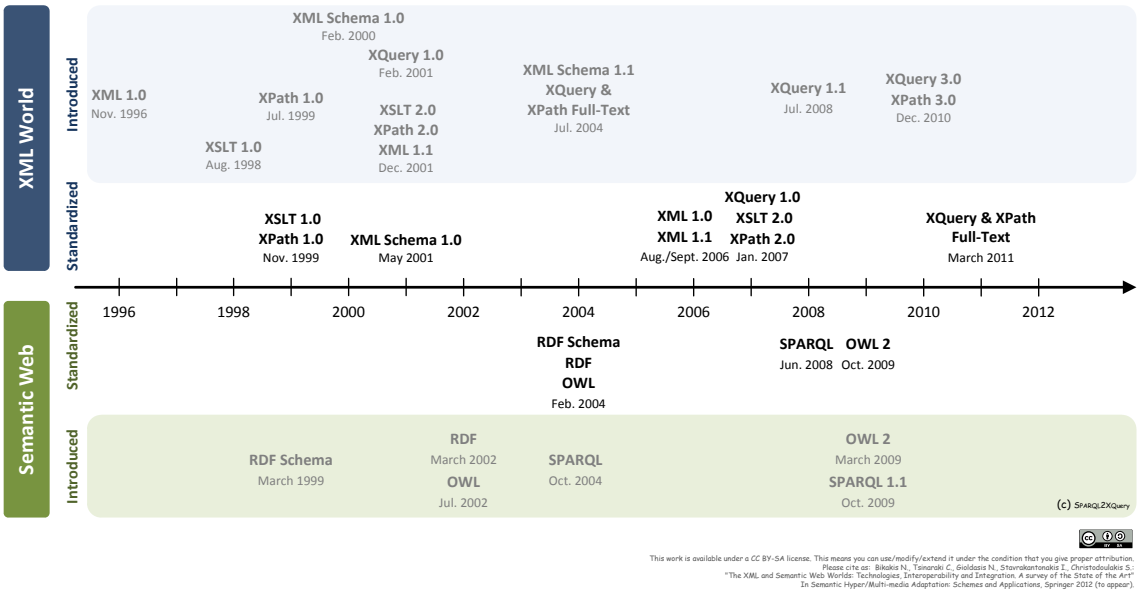

Figure 2.5: XML and Semantic Web W3C Standards Timeline.

2.1.5.2 Terse RDF Triple Language (Turtle)

Turtle defines a textual syntax for RDF that allows RDF graphs to be completely written in compact and natural text form [3]. The latest version was submitted as a W3C Team Submission 28th of March 2011. Listing 2.2 shows the serialized form of figure 2.2.

Listing 2.2: Serialization of figure 2.2 into Turtle.

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:Arne foaf:knows ex:Kjetil ;
    foaf:familyName "Hassel" .
```

We see from the example that IRIs are written with angular brackets, literals with quotation

marks, and statements ends with either a semicolon or a period. The usage of semicolon is a syntactic sugar, and enables writing the following triples without their subject, as they reuse the subject in the first statement. We can also reuse the subject and the predicate in a statement by using the comma, in essence writing a list.

The syntax `@prefix` is also used in the listing. This allows us to introduce namespaces, and abbreviate IRIs by prefixing them (e.g. `http://example.org/Arne` → `ex:Arne`). We also have the term `@base`, which also enables us to abbreviate IRIs, by writing the suffix in angular brackets (e.g. `@base <http://example.org/>` → `<Arne>`).

Turtle also supports BNs by wrapping the statements in square brackets. Listing 2.3 shows all of these syntaxes in use by serializing figure 2.3.

Listing 2.3: Serialization of figure 2.3 into Turtle.

```
@base <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<Arne> foaf:knows [
    foaf:nick "Bjarne" , "Buddy" .
]
```

There is also syntactic sugar for writing collections. This is done by enveloping the resources as a comma-separated list in parentheses. Lastly, Turtle abbreviates common data types, e.g. the number forty two can be written `42`, instead of `"42"^^<http://www.w3.org/2001/XMLSchema#integer>`, and the boolean true can be written `true` instead of `"true"^^<http://www.w3.org/2001/XMLSchema#boolean>`.

Turtle has become popular amongst the academic circles of SW, as it is a valuable educational tool because of its simplicity and readability.

2.1.5.3 Notation3 (N3)

N3 is often presented as a compact and readable alternative to RDF/XML [7], but the syntax supports greater flexibility than the confinements of RDF (e.g. support for calculated entailment with “built-in” functions [5]).

It dates back to 1998 [20, p. 25], and currently holds status as a Team Submission at W3C, last updated 28th of March 2011. Figure 2.2 is serialized as N3 in listing 2.4.

Listing 2.4: Serialization of figure 2.2 into N3.

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:Arne foaf:knows ex:Kjetil ;
    foaf:familyName "Hassel" .
```

N3 shares a lot of the syntax of Turtle, but is an extension in the regard that it has extra syntax (e.g. `@keywords`, `@forAll`, `@forSome`) [3, sec. 9].

2.1.5.4 N-Triples

N-Triples was designed to be a fixed subset of N3 [32, sec. 3]. It is also a subset of Turtle, in that Turtle adds syntax to N-Triples [3, sec. 8]. Serialization of figure 2.2 is given in listing 2.5.

Listing 2.5: Serialization of figure 2.2 into N-Triples.

```
<http://example.org/Arne> <http://xmlns.com/foaf/0.1/knows> <http://example
    .org/Kjetil> .
<http://example.org/Arne> <http://xmlns.com/foaf/0.1/familyName> "Hassel" .
```

One way of looking at N-Triples is to see it as Turtle without the syntactic sugar.

2.1.5.5 RDF JSON

RDF JSON was one of the earliest attempts to make a serialization of RDF in JavaScript Object Notation (JSON). It is designed as part of the Talis Platform[5], and is a simple serialization of RDF into JSON. Figure 2.2 is serialized into RDF JSON in listing 2.6.

Listing 2.6: Serialization of figure 2.2 into RDF JSON.

```
{
    "http://example.org/Arne": {
        "http://xmlns.com/foaf/0.1/knows": [ {
            "value": "http://example.org/Kjetil",
            "type": "uri"
        },
        "http://xmlns.com/foaf/0.1/familyName": [ {
            "value": "Hassel",
            "type": "literal"
        }
    }
}
```

RDF JSON uses the syntax provided by JSON (explained in section 2.2.4). All triples have the form `{ "S": { "P": [ O ] } }`, where `"S"` is the subject, `"P"` is the predicate, and `O` is a JSON object with the following keys:

- `type`, required: either `"uri"`, `"literal"` or `"bnode"`.
- `value`, required: the lexical value of the object.
- `lang`, optional: the language of the literal.
- `datatype`, optional: the data type of the literal.

[5] `http://docs.api.talis.com/platform-api/output-types/rdf-json`

2.1.5.6 JavaScript Object Notation for Linked Data (JSON-LD)

JSON-LD is another JSON based serialization of RDF, and is the newest serialization to be included by W3C. It became a working draft on 12th of July 2012, after being in the works for about a year by the JSON for Linking Data Community Group (JSON-LD CG)[6]. It has been included in the work of the RDF Working Group (RDF WG) in hope that it will become a W3C Recommendation that will be useful to the broader developer community[7].

JSON-LD CG has from the start worked with the concern that RDF may be to complex for the JSON-community[8], and as such has embraced LD rather than RDF. That being said, it is a goal that JSON-LD will serialize a RDF graph, *if* that is what the developer want to do. This is reflected in the current working draft, in that subjects, predicates and objects "*SHOULD* be labeled with an IRI".

Another design goal of JSON-LD is simplicity, meaning that developers only need to know JSON and two keywords (i.e. `@context` and `@id`) to use the basic functionality of JSON-LD [36, sec. 2]. So how do we use these keywords? Lets look at two examples in listings 2.7 and 2.8, which serialize figures 2.2 and 2.3 respectively.

Listing 2.7: Serialization of figure 2.2 into JSON-LD.

```
1 {
2     "@context": {
3         "ex": "http://example.org/",
4         "foaf": "http://xmlns.com/foaf/0.1/"
5     },
6     "@id": "ex:Arne",
7     "foaf:knows": "ex:Kjetil",
8     "foaf:familyName": "Hassel"
9 }
```

Listing 2.8: Serialization of figure 2.3 into JSON-LD.

```
1 {
2     "@context": {
3         "ex": "http://example.org/",
4         "foaf": "http://xmlns.com/foaf/0.1/"
5     },
6     "@id": "ex:Arne",
7     "foaf:knows": {
8         "foaf:nick": [ "Bjarne", "Buddy" ]
9     }
10 }
```

In listing 2.7 we see that prefixing namespaces are featured in line 3 and 4. We also see that the subject are defined by using the proprety `@id`. The absence of `@id` creates a blank node, as shown in listing 2.8.

[6] `http://json-ld.org/`, `http://www.w3.org/community/json-ld/`
[7] `http://www.w3.org/blog/SW/2011/09/13/the-state-of-rdf-and-json/`
[8] Topic: Formal Definition of Linked Data at `http://json-ld.org/minutes/2011-07-04/`

Another design goal of JSON-LD is to provide a mechanism that allow developers to specify context in a way that is out-of-band. The rationale behind is to allow organizations that already have deployed large JSON-based infrastructure to add meaning to their JSON documents that is not disruptive to their day-to-day operations [36]. In practice this will work by having two JSON documents, one being the original JSON document, which is not linked, and another that provide rules as to how terms should be transformed into IRIs. Listing 2.9 shows how a serialization of figure 2.1 could be transformed into the serialization of 2.2.

Listing 2.9: Framing in JSON-LD.

```
// A non-LD JSON object
{
    "Arne": {
        "knows": "Kjetil",
        "lastname": "Hassel"
    }
}
// A JSON-LD object designed to transform the object above into a JSON-LD
   compliant object
{
    "@context": {
        "ex": "http://example.org/",
        "foaf": "http://xmlns.com/foaf/0.1/",
        "Arne": {
            "@id": "ex:Arne"
        },
        "Kjetil": {
            "@id": "ex:Kjetil"
        },
        "knows": "foaf:knows",
        "lastname": "foaf:familyName"
    }
}
```

2.1.5.7 Resource Description Framework in Attributes (RDFa)

RDFa is another serialization that recently got promoted in the W3C-system. As of 7th of June 2012 it is a W3C Recommendation, and offers a range of documents (the RDFa Primer[9], RDFa Core[10], RDFa Lite[11], XHTML+RDFa 1.1[12], and HTML5+RDFa 1.1[13]).

RDFa makes it possible to embed metadata in markup languages (e.g. Hypertext Markup Language (HTML)), so as to make it easier for computers to extract important information. This is in response to the fact that some semantics may not be specific enough. Take the title-tags in HTML, `H1-H6`. Good practices suggest only using `H1` one time, so that it only specifies the most important title for the page. But even so, what does the `H1`-tag specify title for? Is it the page as a whole, or is it the specific article on that page. With RDFa you can specify this.

[9] http://www.w3.org/TR/rdfa-primer/
[10] http://www.w3.org/TR/rdfa-core/
[11] http://www.w3.org/TR/rdfa-lite/
[12] http://www.w3.org/TR/xhtml-rdfa/
[13] http://www.w3.org/TR/rdfa-in-html/

The reasoning is that by making use of independently created vocabularies, the quality of metadata will increase. And by tying it into RDF, you can increase the overall knowledge of WWW.

RDFa has a syntax much to big to describe in detail here, but lets look at an example, by serializing figure 2.2 into a fracture of HTML, given in listing 2.10.

Listing 2.10: Serialization of figure 2.2 in RDFa.

```
<div
  vocab="http://example.org/"
  prefix="foaf: http://xmlns.com/foaf/0.1/"
  about="Arne">Arne knows
  <span
    property="foaf:knows"
    resource="Kjetil">Kjetil</span>
  and has last name <span
    property="foaf:familyName">
    Hassel</span>.</div>
```

Listing 2.10 shows us the use of the attributes `vocab`, `prefix`, `about`, `property`, and `resource`:

- `vocab` defines the usage of a single vocabulary for the nested terms.
- `prefix` allows us to introduce prefixes in case we want to mix in more vocabularies.
- `about` defines the subject in a triple.
- `property` defines the predicate in a triple.
- `resource` may define the object and the subject, depending on context.

2.1.6 Querying

An important feature of structured data is the possibility of querying it. You could have the users scour model in tools like a SW or RDF browser, but this can be a tedious task, and very inefficient for a machine. To query RDF we need a query language that recognizes RDF as the fundamental syntax [19, p. 192] (or rather, as the fundamental model).

2.1.6.1 SPARQL Protocol and RDF Query Language (SPARQL)

SPARQL is the answer to the need for a query language. It exists as version 1.0, which became a W3C Recommendation 15th of January 2008, and as version 1.1, which is a working draft, last updated 5th of January 2012. Version 1.1 builds upon version 1.0, and sports features such as [35]:

- The query forms `SELECT`, `ASK`, `CONSTRUCT`, and `DESCRIBE`,
- Grouping, ordering, and limitation of results fetched,

- Several shortened query forms,
- Aggregation,
- Subqueries,
- Negation,
- Expressions in the `SELECT` clause and Property Paths,
- Assignment, and
- A large list of functions and operators.

As the most powerful version, I will use version 1.1 as the basis for this thesis, and will be the version I refer to when referring to SPARQL.

There are four fundamental forms of read-queries in SPARQL, namely `SELECT`, `ASK`, `CONSTRUCT`, and `DESCRIBE`. The two latter returns new graphs, that can be used as basis for additional queries and manipulations (e.g. merging with other graphs).

The `SELECT` form enables us to query for variables, and return them in tabular form. We can project a specific list of variables we want returned, or just select all variables by using the asterisk sign.

Listing 2.11 shows a very simple example of a `SELECT` query. If we use that query against the model in figure 2.2, we will get the table 2.1 as a result.

Listing 2.11: An example of the `SELECT` form in SPARQL

```
1 SELECT *
2 WHERE { ?subject ?predicate ?object }
```

?subject	?predicate	?object
http://example.org/Arne	http://xmlns.com/foaf/0.1/knows	http://example.org/Kjetil
http://example.org/Arne	http://xmlns.com/foaf/0.1/familyName	“Hassel”

Table 2.1: Result from using query in listing 2.11 on the model in figure 2.2

As we see from table 2.1, the query lists all triples we know in the model.

The `ASK` form enables us to verify whether or not certain query pattern are true or not. We could use it to ask if we know from the model in figure 2.2 whether or not there are an entity which has a given name `"Arne"`. Listing 2.12 shows how this is done.

Listing 2.12: An example of the `ASK` form in SPARQL

```
1 @prefix foaf: <http://xmlns.com/foaf/0.1/>
2 ASK { ?x foaf:givenName "Arne" }
```

In our case the result would be `false`.

Listing 2.13: An example of the CONSTRUCT form in SPARQL

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
CONSTRUCT { ?x foaf:givenName "Arne" }
WHERE { ?x foaf:familyName "Hassel" }
```

The CONSTRUCT form enables us to derive a graph derived from other graphs. Lets look at another example in listing 2.13.

Now, if we were to run the ASK query in listing 2.12 against the new graph, we would get the result true. And if we ran the SELECT query in listing 2.11, we would get the result in table 2.2.

?subject	?predicate	?object
http://example.org/Arne	http://xmlns.com/foaf/0.1/givenName	“Arne”

Table 2.2: Result from using query in listing 2.11 on the graph resulting from the query in listing 2.13 begin executed on the model in figure 2.2.

The DESCRIBE form results in a single RDF graph. It differs from the CONSTRUCT form in that we do not specify which triples we want the new graph to consist of, but rather that the SPARQL query processor determines which triples that are relevant. The relevant triples depend on the data available in the graph(s) queried, but takes basis in the resource(s) identified in the query pattern.

Lets look at the query in listing 2.14, which we apply to the models in figures 2.2 and 2.3, which we have assigned to IRIs http://example.org/GraphA and http://example.org/GraphB respectively. The result could be something like the serialization shown in listing 2.15.

Listing 2.14: An example of the DESCRIBE form in SPARQL

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix ex: <http://example.org/>
CONSTRUCT ?y
FROM <http://example.org/GraphA>
FROM NAMED <http://example.org/GraphB>
WHERE { ?x foaf:knows ?y }
```

Listing 2.15: A possible serialization of the result from the query in listing 2.14

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
[ foaf:nick "Bjarne" , "Buddy" . ]
```

The resulting graph has two triples, namely the one concerning the entity which we known has the nicks "Bjarne" and "Buddy". As there are no triples where http://example.org/Kjetil acts as the subject, we can not describe anything.

I have introduced the token `FROM` in the query. This syntax allows us to specify which RDF Datasets we wish to query. This syntax is optional, as the query processor will use the default graph if nothing is specified. There can be one default graph, whose IRI we override if we specify `FROM` without `NAMED`. A query can take any number (or none) of named graphs, but do not need a default graph if we have one or more named graphs.

SPARQL has a great number of features, and I can not describe them all here[14]. But suffice to say, SPARQL is a powerful language that enables us to ask a variety of questions regarding our data.

2.1.6.2 SPARQL Update Language

The SPARQL 1.1 specification is part of a set of documents, which comprises ten documents. One of these is the document regarding SPARQL Update Language. It introduces an extension of the SPARQL syntax that allow us to update RDF datasets. The tokens are divided into two groups, Graph Update and Graph Management. The former consists of `INSERT DATA`, `DELETE DATA`, `DELETE/INSERT` (with the shortcut form `DELETE WHERE`), `LOAD`, and `CLEAR`. The latter consists of `CREATE`, `DROP`, `COPY`, `MOVE`, and `ADD`.

I will not go into detail, but SPARQL Update Language delivers a great variety of terms that allows us to manipulate our graphs with SPARQL.

2.1.7 Entailment

An important feature of RDF is the ability to infer knowledge from the existing knowledge, i.e. form or entail new conclusions. This is referred to as entailment. There are multiple forms of entailments in RDF, and it supports one form "out-of-the-box". The document "RDF Semantics"[15] gives details about entailment for RDF, RDFS, and D-entailment.

Other regimes are the OWL Direct Semantics[16], which covers OWL DL, OWL EL, and OWL QL. There is also the idea of Rule Interchange Format (RIF), which outlines a core syntax for exchanging rules. The idea is to support multiple rule language, instead of the specific entailment regimes.

As entailment did not become a part of the framework implemented as part of this thesis, I will not go into greater detail at this point. I will return to entailment in section 8.1.3.1, as part of the discussion.

2.2 JavaScript (JS)

JS begins its life in 1995, then named Mocha, created by Brendan Eich at Netscape [13, 22]. It then got rebranded as LiveScript, and later on JavaScript when Netscape and Sun got together. When the standard was written, it was named ECMAScript, but everyone knows it as JavaScript. It quickly gained traction for its easy inclusion into web pages, but was long ridiculed by developers [11].

[14] The SPARQL 1.1 specification numbers almost 100 pages as of this writing

[15] `http://www.w3.org/TR/rdf-mt/`

[16] `http://www.w3.org/TR/owl2-direct-semantics/`

Douglas Crockford states in his article "JavaScript: The World's Most Misunderstood Programming Language" ten reasons for the confusion centering JS, given in the list below:

1. The Name.
2. Lisp in C's clothing.
3. Typecasting.
4. Moving Target.
5. Design Errors.
6. Lousy Implementations.
7. Bad Books.
8. Substandard Standard.
9. Amateurs.
10. Object-Oriented.

Luckily there has been some changes to the list since its conception in 2001.

Point 1-5 is quite valid yet[17], but can be remedied by good and educational resources for learning JS[18].

Point 6 is (mostly[19]) not valid anymore. If the community learned anything from the browser wars, it was to work with the community through the process of standards. Ecma Internationals effort to create a specification based on the de facto standard amongst the browsers has been successful, and groups such as W3Cs HTML Working Group (HTMLWG) and The Web Hypertext Application Technology Working Group (WHATWG) drives the production of standards, and great efforts are made to increase efficiency amongst JS-engines. Another testimony to the fact that implementations are increasingly popular are the efforts to use JS as a programming language outside the browser (described in section 2.2.7).

Point 7 depends on your view of good books, and although there is much left to desire, there are some good books out there[20]. But more importantly, there are several efforts to deliver resources of high quality to educate developers in JS. These resources are increasingly -

[17]Design issues in JS has given rise to many frustrating moments, creating momentum for websites such as `http://wtfjs.com/`, which delivers examples of "weird code".

[18]In this regard, `http://dailyjs.com/` offers a variety of good resources for learning JS, specifically its articles tagged with #beginner (`http://dailyjs.com/tags.html#beginner`).

[19]Considering the slow pace of browser-update in some communities, e.g. usage of Internet Explorer version 6 (IE6) in China, lousy implementations is a thing that is becoming a thing of the past.

[20]Which include, in the authors view:

- JavaScript: The Definitive Guide, 6th edition, by David Flanagan (O'Reilly Media).
- JavaScript: The Good Parts, by Douglas Crockford (O'Reilly Media).

perhaps fittingly - web-based. There is also an increase of interest on conferences that target developers[21].

Point 8 is left to be discussed (I have not read and analyzed the 440 pages that ECMAScript version 3 and 5 consists off), but the implementation of the standards seem to suggest that this point is not so valid anymore.

JS is increasingly becoming part of the professional world, adaptations into conferences being one of the arguments suggesting this trend. You also have examples of major companies either supporting or developing JS-libraries[22]. This would suggest that point 9 is not the case anymore[23].

Point 10 is still valid, as it can be difficult for developers trained in conventional object-oriented languages like Java and C#. Again, as with point 1-5, this is remedied by proper, educational resources, that developers can turn to when puzzled by the intricacies of JS.

JS may be a greatly misunderstood language even today, but it seems to have a lot going for it. The fact that it is the de facto programming language for the web puts it into a position worthy of respect, and should be regarded as a resource which can be used for many great things.

2.2.1 Object-Oriented

JS is fundamentally Object-Oriented (OO) as objects are its fundamental datatype [15, p. 115]. It treats objects different than many other programming languages though, as it does not have classes and class-oriented inheritance. There are fundamentally two ways of building up object systems, namely by prototypical inheritance (explained in section 2.2.1.1) and by aggregation (explained in section 2.2.1.2) [11].

Another design feature is its support of the functional programming style, by treating functions as first-class objects. This feature is explained thoroughly in section 2.2.1.3.

The level of object-orientation in JS is shown in that even literals (i.e. all primitive values except *undefined* and *null*) can be treated as objects. They are, however, immutable, and does not share the dynamic properties that "normal" objects in JS do. JS handles this by wrapping the values into their respectively object-type (e.g. String, Number, and Boolean). An example showing this is shown in listing 2.16.

Listing 2.16: Use of literals in JS

```
1 var stringObject = new String('foo');
2 console.log(stringObject.length); // logs 3
3 var stringLiteral = 'foo';
4 console.log(stringLiteral.length); // logs 3
```

[21] Notably, in Norway you have Web Rebels (`http://webrebels.org/`), and JS has its own session on Norwegian Developers Conference (NDC), with somewhat above 10% of the talks concerning JS.

[22] One example being jQuery, which is shipped with Microsoft's Visual Studio, another being AngularJS, which is an MIT-licensed Model-view-controller (MVC) framework developed by Google.

[23] That being said, percentage-wise it is probable that JS is still written by more amateurs than professional developers. This is not a bad thing though, as it expresses the power of adaptation that JS features, and may be a gateway for developers-to-be. Also, lets not forget that the word amateur means "lover of", and love of computer technologies is something to be embraced.

Other objects that are somewhat different from the norm is the Array- and Math-object, the former representing a list of values and the latter sporting a set of static methods.

Objects in JS do not need classes to be instantiated. But it is possible to emulate classes in JS though, as it helps us use class-depended features (e.g. some Software Design patterns (SDPs)), and an example is shown in listing 2.17.

Listing 2.17: Emulation of classes in JS

```
var MyClass = function () {
    this.myProperty = 42;
    this.myMethod = function (value) {
        return value + this.myProperty;
    };
};

var myObject = new MyClass();
console.log(myObject.myMethod(1295)); // logs 1337
```

2.2.1.1 Prototypical Inheritance

At the heart of all object-handling in JS is `Object`. All objects inherit this object if nothing else is specified, and it is there we find the default properties and methods that are shared by all objects. We can manipulate which object we want our objects to inherit, and as such can create a hierarchy of objects. Listing 2.18 show some examples of inheritance. In it we see how we can initiate objects, and how we can assign them to inherit other objects.

Listing 2.18: Usage of prototype in JS

```
var objectA = {},
    objectB = new Object(),
    objectC = Object.create(objectB)
objectB.__proto__ = objectA;
Object.propA = 42;
objectB.propA = 1337;
console.log(objectA.propA, objectC.propA); // logs 42, 1337
```

The simple secret behind prototypical inheritance is that all objects have the property `__proto__`. When a property or method is called, JS will search for the called element by traversing the objects' properties, and if not found, it will continue with the prototype. We can visualize the structure in listing 2.18 as a tree, and have done so in figure 2.6.

So when we call `objectA.propA`, JS will check if `objectA` has the property `propA`. As it has not it will continue to its prototype, which is `Object`. Now, as `Object` has the property `propA`, JS will return its value, which is `42` in our case. But if we call `objectC.prop`, it will not have to go longer than `objectB` to see that there is a property that matches its search.

A last note is that `Object` also have the property `__proto__`. This can also be manipulated, but JS will take care so that we do not run into an infinite loop when looking for properties that does not exist (it is also considered a bad practice (i.e. an anti-pattern) to manipulate the prototype of `Object`).

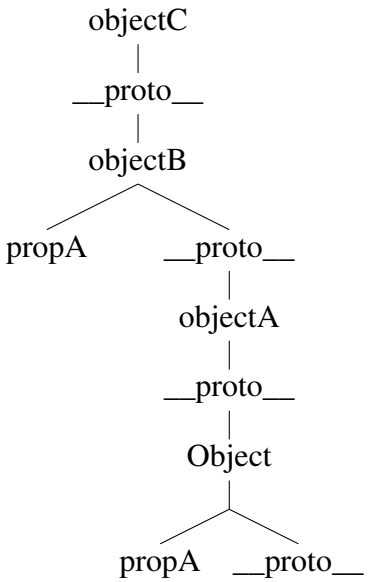

Figure 2.6: Object inheritance created in listing 2.18 visualized as a tree.

2.2.1.2 Dynamic Properties

All mutable objects in JS can be manipulated at run-time. This we also see in listing 2.18, as we add the property `propA` in line 5 and 6. Objects are basically containers for key-value entities, where the key is a string. In this regard, objects in JS can be regarded as maps, or dictionaries.

We can at any time manipulate existing properties by replacing its values or delete the key altogether. We can also manipulate objects that are prototyped, and the objects that inherit will also be affected. How this works is that JS creates a reference in memory for variables that are set as objects. If those variables where to be set to other variables, the reference would be copied, not the values contained within.

A note on mutability and immutability: JS differentiate between primitive values and object. The former are immutable, while the latter is mutable. ECMA-262 5.1 Edition (ECMA5) offers three new functions that alter this behavior, namely the properties `seal`, `freeze`, and `preventExtensions` in `Object` (with the responding `isSealed`, `isFrozen`, and `isExtensible` to test whether or not these are set) [13, p. 114-115]. Explaining how these functions are outside the scope of this thesis, but suffice to say is that ECMA5 adds some spice to the mutable properties of JS-objects.

2.2.1.3 Functional Features

All functions are treated as first-class objects, and as such can be manipulated as any other object. It can also be passed around as variables, and this opens for some nifty features. By passing a function as a parameter, we can call that function whenever we want, e.g. after we have loaded a set of resource. This asynchronous feature is explained in depth in section 2.2.5.

Functions can be instantiated in many ways, as shown in listing 2.19. A function consists of three elements [15, p. 164]:

1. Name: An identifier that names the function (optional in function definition expressions).

2. Parameter(s): A pair of parentheses around a comma-separated list of zero or more identifiers.

3. Body: A pair of curly braces with zero or more JS-statements inside.

Listing 2.19: Instantiating functions in JS

```
1  function functionA (x) { return x; };
2  var functionB = function (x) { return x; },
3      functionC = function functionD (x) { return x; },
4      functionE = new Function("x", "return x;");
5
6  console.log(functionA(42), // logs 42
7              functionB(42), // logs 42
8              functionC(42), // logs 42
9              functionD(42), // throws ReferenceError: functionD is not
                  defined
10             functionE(42)); // logs 42
```

All types in listing 2.19 support these requirements, albeit a little differently. Line 1 shows a named function, while the other two are anonymous. Anonymous functions are called through their reference, i.e. the variables they are set to. Named functions is referable by their names, if not they are set to a variable, in which case it will be referable by the variable (line 9 shows what happens if you call the function by its name when its set to a variable).

Functions of the types listed in line 1-3 can be used as constructors for new objects, while the one in line 4 can be used as a prototype. A simple example of this is shown in listing 2.20. It introduces the use of `this`, which will be explained in section 2.2.2.

Listing 2.20: A simple object in JS

```
1  function ObjectA (x) {
2      this.x = x;
3      this.methodA = function (y) {
4          return this.x + y;
5      };
6  }
7
8  var A = new ObjectA(1300);
9  console.log(A.methodA(37)); // logs 1337
```

2.2.2 Scope in JS

The way JS handles the scope may be confusing to developers coming from class-oriented programming languages. JS does not contain syntax such as private of protected for use with variables, but it supports private variables for objects. It does so in the way it handles the context functions are part of (e.g. the scope).

Functions in JS can be nested within other functions, and they have access to any variables that are in scope where they are defined. This means that JS-functions are closures, and it enables important and powerful programming techniques [15].

If a variable is not set as a property in an object, it will be a part of the global object. The global object in JS depends on which environment it is run in, but in most browsers it is represented by the object `window`. This has some consequences, like the fact that usage of the syntax-element `var` is optional; it will become a key-value entity in the scope in which it is declared, which is the global object if nothing else is specified. This is exemplified in listing 2.21.

Listing 2.21: Examples of scope in JS

```
var x = 42;
y = 42;
window.z = 42;

console.log(x, y, z); // logs 42 42 42
```

2.2.2.1 Closure

Lets review a simple example of closure, given in listing 2.22. In this example we have two functions, one which works as a constructor, and another that merely calls a function it has been given as parameter. When we pass `a.getValue` to `functionA`, JS also include the context which that method runs in, in effect creating a closure.

Listing 2.22: A simple example of closure in JS

```
var ObjectA = function (val) {
            this.val = val;
        this.getValue = function () {
            return this.val;
        }
    },
    functionA = function (getFunc) {
        return getFunc();
    };

var a = new ObjectA(42);
console.log(functionA(a.getValue)); // logs 42
```

This effect is increasingly used in JS-libraries, and is getting a lot of appraise from the community. But it is also a headache for many aspiring JS-developers, as it may be a bit difficult to wrap your head around (and use correctly). Lets look another example of what may go wrong, given in listing 2.23. In this example we try to access `this.val` inside `functionAA`. But as `functionAA` is not part of the scope of `functionA`, and thereby not being a part of the closure given to `functionB`, we fall back to calling on the global object. Since the global object does not have a property named `val`, it will return `undefined`.

Listing 2.23: An example of code gone wrong because of faulty handling of closure

```
1  var functionA = function (val, func) {
2          this.val = val;
3          function functionAA () {
4              return this.val;
5          }
6          return func(functionAA);
7      },
8      function B = function (func) {
9          return func();
10     };
11
12 console.log(functionA(42, functionB)); //logs undefined
```

2.2.3 Static functions

JS supports static functions in that all functions are treated as objects, and by extension can be extended with methods. Listing 2.24 illustrates an example of this.

Listing 2.24: An example of static functions in JS

```
1 var funcA = function (val) { this.val = val; },
2     objA = new funcA(1337);
3 funcA.funcB = function () { return 42; }
4 funcA.funcC = function () { return this.val; }
5 console.log(funcA.funcB()); // logs 42
6 console.log(objA.funcC()); // throws TypeError
7 console.log(funcA.funcC.call(objA); // logs 1337
```

Note that static functions are not accessible as methods in objects constructed with the parenting functions as constructor. But we can manipulate the scope of the functions by overloading `this` (with `call`) to be the object we wish to refer to, as shown on line 7.

2.2.4 JavaScript Object Notation (JSON)

JSON is a lightweight, text-based data interchange format. It is originally based on JS, but is language-independent [12]. It was specified by Douglas Crockford in RFC 4627, and enjoys support in most major programming languages.

JSON consists of literals that are either `false`, `null`, `true`, an object (i.e. collections of key-value pairs), an array (i.e. lists), a number, or a string [12]. Listing 2.25 shows some examples of valid JSON-objects, as well as some structures that are not valid JSON.

JS supports JSON by default (given in ECMAScript Language Definition [13]).

2.2.5 Asynchronous Loading of Resources

Asynchronous loading or resources are common in browsers. Normal HTML documents normally externalize much of its Cascading Style Sheets (CSS) and JS functionality, as dictated by good practices. Those resources are loaded by the browser by default, without too much hassle.

Listing 2.25: Examples of structures in JS that are valid and invalid JSON-objects

```
// valid, can all be parsed by JSON.parse
var goodA = '42',
    goodB = '{ "a": 42 }',
    goodC = '[ 1337,  { "a": 42 }]';
// invalid, will all make JSON.parse throw a SyntaxError
var badA = '', // unexpected end of input
    badB = 'function (x) { return x; }', // unexpected token u
    badC = '{ "a": new Object() }'; // unexpected token e
```

But when it comes to making use of the browsers API (i.e. the ones available to JS) to load resources asynchronously, it becomes another game entirely.

2.2.5.1 Same Origin Policy (SOP)

As with many issues, handling external resources are difficult in JS because of security issues. And justifiable so, as JS becomes an increasingly powerful programming language, so are the possibilities to abuse it. Users of WWW are increasingly used to insert personal information, and if we cannot trust owners of web pages to control what is being run on their site, then there would be a lot of issues with trust on the web[24].

Perhaps the most important security concept within modern browsers is the idea of SOP[25]. Although there is no single SOP governing how browsers implement it, the idea is that resources that do not share the same origin (i.e. having the same scheme, host, and port in the IRI (concepts explained in section 2.1.4.2)) are isolated from each other.

It is possible to circumvent SOP in JS by inserting a script-tag referring to an external file. This technique is used by JSON with padding (JSONP), which allows JSON residing in external files to be loaded during run-time.

2.2.5.2 Content Security Policy (CSP)

Another way of handling security concerning external resources is CSP. CSP is in the works (Working Draft at W3C[26]), and as an incomplete standard it may be prone to changes. But the basic idea is to let developers whitelist external resources. The policy is first and foremost being designed to be part of the HTTP response header, but there is also work on letting it be a part of `HEAD` in a HTML document, as a `META` tag.

2.2.5.3 XMLHttpRequest Level 2 (XHR2)

XMLHttpRequest (XHR) has been part of the world of browsers for a while. It was conceived by Microsoft in their work on Microsoft Exchange Server 2000, and was later ported by Mozilla. It was overlooked for quite a while, until AJAX became a trend, as developers understood the power it had to load resources asynchronously (and synchronously, if needed).

[24] A lot can be said about trust on the web, but that is not the purpose of this thesis.
[25] `http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy`
[26] `http://www.w3.org/TR/CSP/`

XHR2 is a Working Draft as of this writing, but introduces several features requested by the community, allowing cross-domain fetching of resources being one of them. To allow this, it makes use of another standard which is in the making, namely Cross-Origin Resource Sharing (CORS)[27]. This technology is already available in some browsers. But its inherit problem is that it requires domain-owners to add information to their HTTP headers.

Another technology developed to fetch resources across domains are XDomainRequest (XDR). But as it was not included in the framework, I have let it be a part of the discussion in section 8.1.4.1.

2.2.6 CommonJS (CJS)

CJS is a volunteer-driven project[28] aiming to standardize and implement specifications that expand the functionality of JS. Specifications include handling of modules, unit testing, packaging, Input/Output (I/O), handling of binary data, and much more. We have included details concerning three of these specifications (the promise pattern, section 2.2.6.1 and the module patterns AMD and CJS Modules, sections 2.2.8.3 and 2.2.8.4), as they have been included in the framework.

2.2.6.1 Promise Pattern

The promise pattern is titled Promises/A by CJS[29]. It is also referred to as Deferred, and works by having an object represent a promise. The promise consists of a result will be returned at some time in the future, and in the meantime, the run-time will continue evaluating the rest of the sourcecode. This can be set up so that when the result is ready, a function is called with the result sent as parameter. This allows for some proper handling of asynchronous functionality.

Listing 2.26 shows some examples of the API. A central point of these examples are that the functions passed as parameters to the then-function are called as soon as the promise are resolved, i.e. detached from the order in which they were called in the code.

2.2.7 Server-side implementations

As JS has become an increasingly popular programming language, so has its use outside of the browser. One of these branches is the use of JS for server-side web-applications. As part of this thesis I have only used one such implementation as a run-time environment for my Test-driven development (TDD), but a more in-length discussion of the matter can be found in section 8.1.5.

2.2.8 Module Patterns

JS is a flexible language, and one area in which this is very clear is when it comes to module handling. This is not a surprise, as handling variables and ensuring they are not compromised by code elsewhere in the application is harder than you might think. As such, “modules are an

[27] `http://www.w3.org/TR/cors/`
[28] `http://www.commonjs.org/`
[29] `http://wiki.commonjs.org/wiki/Promises/A`

Listing 2.26: Examples of the Promise API

```
// When is available as a global variable
var promiseA = When.defer(),
    promiseB = When.defer();
setTimeout(function () { promiseA.resolve(42); }, 2000));
setTimeout(function () { promiseB.resolve(1337); }, 1000));

// Preparing single promises
promiseA.then(function (result) {
    console.log(result); // logs 42 after 2000 milliseconds
});
promiseB.then(function (result) {
    console.log(result); // logs 1337 after 1000 milliseconds
});

// Preparing multiples promises
When.all([ promiseA, promiseB ], function (results) {
    console.log(results); // logs [ 42, 1337  ] after 2000 milliseconds
});
```

integral piece of any robust application's architecture and typically help in keeping the units of code for a project both cleanly separated and organized" [24].

This section will describe some of the patterns of module handling I have found during my research.

2.2.8.1 Contained Module

The Contained Module pattern is designed to encapsulate private variables and return an explicit object with public methods that can work with the private variables. It was made popular by Douglas Crockford, and is used extensively in smaller libraries. Listing 2.27 shows an example using this pattern.

Listing 2.27: Use of contained modules in JS

```
var myModule = (function () {
  var myPrivateVariable = 42;
  function myFunction () {
    return myPrivateVariable;
  }
  return {
    myPublicFunction: myFunction
  };
})();
console.log(myModule.myPublicFunction); // logs 42
```

The problem with this pattern is that it does not really address how to combine several modules. For that we turn to the other patterns.

2.2.8.2 Namespaces

The simplest way structure several modules is to follow the Namespaces pattern. An example of it can be seen in listing 2.28.

Listing 2.28: Use of namespaces in JS

```
var OurNamespace = {};
// in another file, called after the above code has been evaluated
(function (ns) {
  ns.anotherLevel = {};
})(OurNamespace);
// another file yet again, called after the above code
(function (ns) {
  ns.anotherLevel.ourFunctionalModule = function () { /* ... */ };
})(OurNamespace);
```

It requires the developer to include the modules in correct order, which can be troublesome. The one single argument to use this is that it is supported out-of-the box, as it does not depend on any extra functionality than the one inherent in browsers.

2.2.8.3 Asynchronous Module Definition (AMD)

The AMD pattern is titled Modules/Async/A by CJS[30]. Its overall goal is to provide a solution for modular JS that developers can use today [24]. Essentially it makes use of the functions `define` and `require`. The former defines a module, while the latter enables us to load dependencies that the module requires. Listing 2.29 shows an example.

Listing 2.29: Use of AMD in JS

```
define([
  "dependentModuleA",
  "dependentModuleB"
], function (depModA, depModB) {
  function privateFunction () {
    /* This function is not publicly available by other modules */
  }
  return { /* This object becomes available to other modules */
    myPublicFunction: function () { /* ... */ }
  };
});
```

AMD allows us to split our functionality into modules and easily load components as they are needed, in run-time. This in turn leads do a more decoupled code base, making it easier to make modules reusable. But it may also increase the loading time required, as each module requested fires a HTTP request. Which consequences this has for the framework is further discussed in section 8.1.2.

[30] http://wiki.commonjs.org/wiki/Modules/Async/A

2.2.8.4 CJS Module

Another pattern to emerge from the CJS community the CJS Module pattern. It makes use of the functions `require` and `exports`. An example is given in listing 2.30.

Listing 2.30: Use of CJS Module in JS

```
var moduleDependency = require("moduleWeAreDependentOn");

function privateFunction () {
  /* This function is not publicly available by other modules */
}

exports.myModule = { // this object is available to other modules
  myPublicFunction: function () { /* ... */ }
};
```

CJS also allow modules to be loaded asynchronously[31], and in many regards resembles AMD a lot. AMD and CJS Module differ in which environment they cater to. AMD is mostly being used by client-side projects, while CommonJS Modules is used by server-side projects. That said, both types can be used on either sides, and it becomes merely a question of taste.

2.2.8.5 Harmony

Last, we have the modular pattern that is to be part of the sixth edition of EcmaScript, a.k.a. ES.next, a.k.a. Harmony. This pattern makes use of new syntax, and an example can be seen in listing 2.31.

Listing 2.31: Use of modules in Harmony

```
module moduleA {
  export var functionA = function () { /* ... */ }
  export var objectA = { /* ... */ }
  export var propertyA = 42;
}
module moduleB {
  import functionA, objectA, propertyA from moduleA;
  // equivalent to the above: import * from moduleA;
}
```

This syntax is not available in standard browsers yet, as it is still subject to change, however it is available for experimentation through tools such as traceur-compiler[32] and esprima[33].

2.3 Software Design pattern (SDP)

Patterns were originally conceptualized as an architectural concept by Christopher Alexander, who wrote:

[31] Although some environments, such as Node.js (Node), loads the dependencies synchronously.
[32] `http://code.google.com/p/traceur-compiler/`
[33] `https://code.google.com/p/esprima/`

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [1, p. x].

Alexander's work inspired amongst others Kent Beck and Ward Cunningham, who in 1987 presented the report "Using Pattern Languages for Object-Oriented Programs"[34] on OOPSLA-87. They outline the adaptation from Pattern Language to object-oriented programming, and they summarized a system of five patterns that they had successfully used for designing window-based user interfaces.

SDPs did not become popular before the publication of Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (often known as Gang of Four (GoF)) in 1994. They generalized patterns to have four essential elements:

1. **Pattern name:** A handle which we can use to describe a design problem, its solutions, and consequences in a word or two. A pattern name is useful as a higher level of abstraction, increases our pattern vocabulary, and eases communication in social contexts.

2. **Problem:** Each pattern is designed to handle a specific problem, and this part tells us when it is appropriate to use a specific SDP.

3. **Solution:** This part explains in detail how to solve the given problem by explaining the elements that make up the design, their relationships, responsibilities, and collaborations.

4. **Consequences:** All implementations have consequences, and this part tells us what results and trade-offs we may expect from applying the pattern. Consequences may be how the pattern affects a system's flexibility, extensibility, or portability.

In their book they also design a classification scheme that aims to enable developers to refer to families of SDPs. One categorization is by purpose, which can be either creational, structural, or behavioral. The second categorization is by scope, which can be classes or objects. SDPs that are related to this thesis have been classified in table 2.3.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class		Adapter (section 2.3.1)	Interpreter (section 2.3.7)
	Object	Builder (section 2.3.3) Prototype (section 2.3.9)	Adapter (section 2.3.1) Bridge (section 2.3.2) Composite (section 2.3.4) Decorator (section 2.3.5) Facade (section 2.3.6) Proxy (section 2.3.10)	Observer (section 2.3.8) Strategy (section 2.3.11)

Table 2.3: Categorization of SDPs relevant to this thesis, given in the classification scheme proposed by Erich Gamme, et.al. [18, p. 10].

[34] http://c2.com/doc/oopsla87.html

At this point I need to make two points clear. The first is that as JS is a class-less programming language, the categorization class might be a bit off. But remember that we can emulate classes in JS, and this allows us to make use of the class-categorized patterns. The other point is that JS does not support interfaces. Interfaces can be emulated (at the cost of complexity), but is not anything more than a construct that enforces that a list of properties is set at run-time. Therefore I have excluded the use of interfaces in this thesis, falling back to merely describing the abstractions of participants, and how they are represented in the code samples.

GoF continues to describe a consistent format for describing SDPs, which including Pattern Name and and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, and Related Patterns. Using all of these labels takes a lot of pages, and in this thesis I have limited ourselves to a description of the pattern along with a figure and an example in JS.

2.3.1 Adapter

The Adapter pattern "convert the interface of a class into another interface clients expect" [18]. This pattern is useful when one wishes to make use of third party libraries without modifying them. In its classic form, the Adapter pattern are both a class and an object pattern, where the former makes of subclassing, while the latter forms a reference to the components it adapts, thereby routing requests.

In listing 2.32 I have shown examples of both. Line 4 shows subclassing through Object.create, which enables derivatives of AdapterClass to make use of the methods in the Original object. Line 5 shows the constructor function that returns an object that refers to the Original object, and thereby allows routing of calls.

I have not made use of the Adapter pattern in Graphite, a point I return to in section 8.2.1.1 in the discussion.

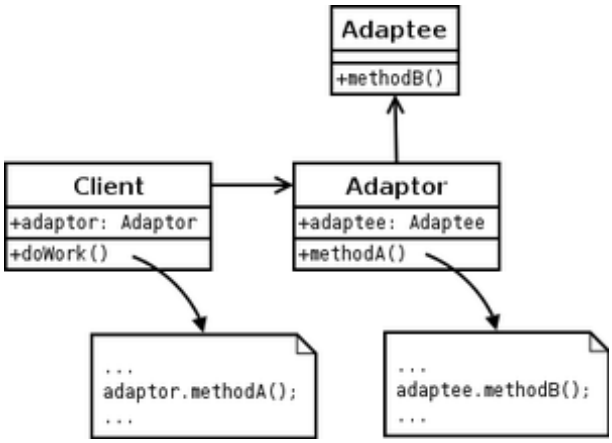

Figure 2.7: Structure of Adapter

2.3.2 Bridge

The Bridge pattern "decouple an abstraction from its implementation so that the two can vary independently" [18]. The pattern is actually often used in JS in terms of event handling, using

Listing 2.32: An example of implementation of Adapter in JS

```
var Original = {
      originalMethod: function (options) { /* ... */ }
    },
    AdapterClass = Object.create(Original).
    AdapterObject = function () {
      return Object.create({
        adapterMethod: function (paramA, paramB) {
          return this.target.originalMethod({ a: paramA, b: paramB });
        }
      }, {
        target: { value : Object.create(Original); }
      });
    };
```

code such as the one in listing 2.33. In that case, the abstraction is that a function is to be called when a specific button is called, the refined abstraction is the actual functions. The implementor on the other hand is a function that takes the id to the button to be handled, and the abstraction that is to be coupled. The concrete implementor is the function `handleClick`, which configures the setup needed.

We could have implemented another abstraction, namely making sure that whatever was passed as `handleClick`'s first parameter was an object that supported the `onclick` property. This way, I could have removed the limitation of sending just strings of ids, e.g. passing the object returned from `document.getElementByClass("buttons")`.

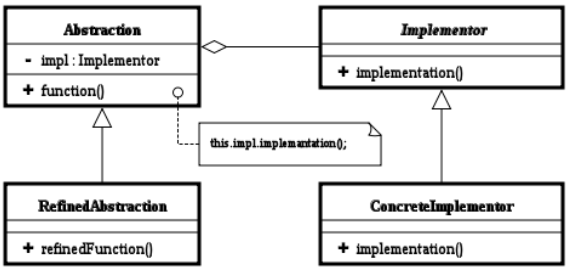

Figure 2.8: Structure of Bridge

2.3.3 Builder

The Builder pattern "separate the construction of a complex object from its representation so that the same construction process can create different representations" [18]. A good example of this is the way jQuery allows us to construct DOM elements (listing 2.34).

These lines should be very easy to read for developers familiar with HTML, and handles a lot of logic that is run behind the scene (e.g. the use `document.createElement`, adding attributes, and text).

Now, lets look at listing 2.35 for my own version of a DOM-builder (a very limited version, i.e. it only support one level of element). I have removed parts of the code, as they unnecessary to understand how the pattern works. The participants are `DOMCreator` (the Director),

Listing 2.33: An example of implementation of Bridge in JS

```
var cancelFunction = function () {
        console.log("Cancel was clicked");
    },
    submitFunction = function () {
        console.log("Submit was clicked");
    },
    handleClick = function (buttonId, func) {
        document.getElementById(buttonId).onclick = function () {
            func();
            return false;
        };
    };
handleClick("CancelButton", cancelFunction); // When clicked, will log "
    Cancel was clicked"
handleClick("SubmitButton", submitFunction); // When clicked, will log "
    Submit was clicked"
```

Listing 2.34: Examples of the Builder pattern in jQuery

```
var paragraph = $("<p>"),
    titleWithText = $("<h1>Our title</h1>"),
    inputWithAttr = $('<input type="password" />');
```

`DOMBuilder` (ConcreteBuilder), and `DOMElement` (the Product). The code works in following steps:

1. We pass to `DOMCreator` the string we want parsed.
2. `DOMCreator` creates an instance of `DOMBuilder`, and passes along the tag.
3. `DOMBuilder` creates an instance of `DOMElement`, and sets the tag.
4. `DOMCreator` parses attributes, if any, and passes them to `DOMBuilder`.
5. `DOMBuilder` adds attributes to the `DOMElement`.
6. `DOMCreator` parses text, if any, and passes it to `DOMBuilder`.
7. `DOMBuilder` adds text.

After these steps, the client can fetch the element by calling `getElement` on `DOMCreator`.

2.3.4 Composite

The Composite pattern "compose objects into tree structures to represent part-whole hierarchies." [18]. This a method of abstracting the types of a complex structure, and streamlining certain procedures. In listing 2.36 I have continued with the DOM, and created a structure that represents DOM elements that can be used to generate HTML.

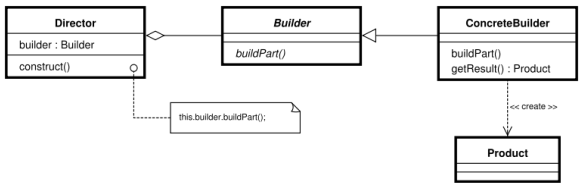

Figure 2.9: Structure of Builder

Listing 2.35: An example of implementation of Builder in JS

```
var DOMElement = {
    attributes = {},
    tag = null,
    text = ""
  },
  DOMBuilder = function (tag) {
    this.element = Object.create(DOMElement);
    this.element.tag = tag;
    this.addAttribute = function (key, value) {
      this.element.attributes[key] = value;
    };
    this.addText = function (text) { this.element.text = text; };
  },
  tokens = {}, // a map of tokens to parse
  fetch = function (str, token) {}, // returns specified type of token
  remove = function (str, token) {}, // removes token, returns modified
     string
  test = function (str, token) {}, // tests for specific token, return
     boolean
  DOMCreator = function (str) {
    var key, tag, text, value;
    // fetches the tag
    this.builder = new DOMBuilder(tag);
    while (test(str, tokens.whitespace)) {
      // fetches key-value pair of attributes, if any
      this.builder.addAttribute(key, value);
    }
    if (!test(str, tokens.slash)) {
      // fetches text, if any
      this.builder.addText(text);
    }
    // We have what we need
  };
DOMCreator.prototype.getElement = function () {
  return this.builder.element;
}
var element = new DOMCreator("<p>42</p>");
console.log(element.getElement()); // logs { attributes: {}, tag: "p", text
    : "42" }
```

In this example we have two Composites (`DOMComposite`, `DOMElement`) and one Leaf (`DOMText`). The client gets the HTML by calling the method `getHtml` on any of the elements desired, and they will take care of producing the result from all nested, if any, elements.

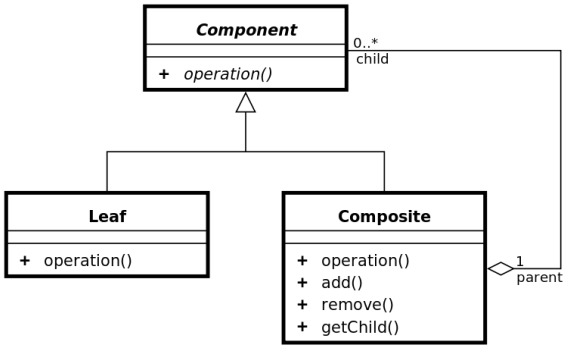

Figure 2.10: Structure of Composite

2.3.5 Decorator

The Decorator pattern "attach additional responsibilities to an object dynamically" [18]. As JS is dynamic in its nature, this is not a very difficult pattern to implement. In listing 2.37, I have simplified the example used by Addy Osmani in his book Learning JavaScript Design Patterns[35].

To use the participants in figure 2.11, we have `PC` as ConcreteComponent, and `addMemory`, `addScreen`, and `addKeyboard` as the ConcreteDecorators.

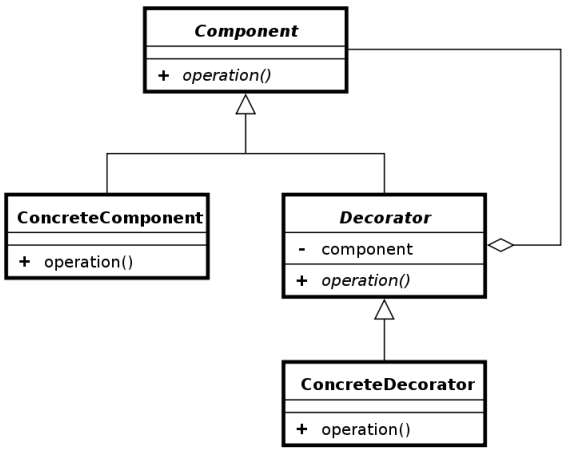

Figure 2.11: Structure of Decorator

2.3.6 Facade

The Facade pattern "provide a unified interface to a set of interfaces in a subsystem" [18]. Again, jQuery shows us an example of design pattern, as the constructor of the jQuery-object applies the Facade pattern. It is usually used to simplify the API the user have to concern himself/herself with, by delivering a subset of methods from underlying modules.

[35] `http://addyosmani.com/resources/essentialjsdesignpatterns/book/#decoratorpatternjavascript`

Listing 2.36: An example of implementation of Composite in JS

```
var DOMComposite = function (children) {
    this.children = children;
  },
  DOMElement = function (tag, content) {
    this.tag = tag;
    this.content = content;
  },
  DOMText = function (text) {
    this.text = text;
    getHtml: function () {
      return this.text;
    }
  };
DOMComposite.prototype = {
  addChild: function (element) {
    this.children.push(element);
  },
  getHtml: function () {
    var child, html = "";
    for (child in this.children) {
      html += child.getHTML();
    }
    return html;
  }
}
DOMElement.prototype.getHtml = function () {
  var text = "<" + this.tag;
  if (this.content) {
    return text + ">" + this.content.getHtml() + "</" + this.tag + ">";
  }
  return text + "␣/>";
};
DOMText.prototype.getHtml = function () {
  return this.text;
}
var text1 = new DOMText("42"),
    text2 = new DOMText("1337"),
    composite1 = new DOMComposite([ text1 ]),
    element1 = new DOMElement("span", text2),
    composite2 = new DOMComposite([ composite1, element1 ]);
console.log(composite2.getHtml()); // logs "42<span>1337</span>"
```

In listing 2.38, I have designed an object that takes the libraries jQuery and when.js (When), and delivers a new interface that taps into some of their functionality. The facade in the example have one method, namely `load`, and promises to load the callback functions in the order they are used (i.e. `http://example.org/1337` will not be loaded before `http://example.org/42` has completed).

Listing 2.37: An example of implementation of Decorator in JS

```
var PC = { cost: function () { return 1000; } },
    addMemory = function (PC) {
      return PC.cost: function () { PC.cost() + 300; };
    },
    addScreen = function (PC) {
      return PC.cost: function () { PC.cost() + 30; };
    },
    addKeyboard = function (PC) {
      return PC.cost: function () { PC.cost() + 7; };
    },
    myPC = Object.create(PC);
addMemory(myPC);
addScreen(myPC);
addKeyboard(myPC);
console.log(myPC.cost()); // logs 1337
```

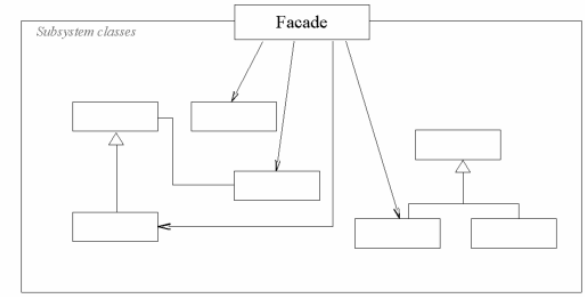

Figure 2.12: Structure of Facade

2.3.7 Interpreter

The Interpreter pattern takes a given language and "define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language" [18].

In my simple example I want to be able to parse simple equations, using the following rules:

- Legals tokens are plus, minus and numbers, and these tokens are represented as expressions.
- It reads the equation from left to right.
- No whitespace allowed.
- The plus and minus expression take the its left expression as parameter, and expects a number to come after it (e.g. "1+44-3" and "-2+3" are both allowed, but "2++3" is not).

The result is an object with a tree-structure consisting of my grammar. E.g. the equation "1+2-3" would look like figure 2.13.

Listing 2.38: An example of implementation of Facade in JS

```
// assumes $ and When are global variables
var facade = (function (jQuery, When) {
  var promise = null;
  function load (uri, callback) {
    promise = When.defer();
    jQuery.get(uri, {}, function () {
      promise.resolve(arguments);
      callback.apply(this, arguments);
    });
  }
  return {
    load: function (uri, callback) {
      if (promise) {
        promise.then(function () {
          load(uri, callback);
        });
      } else {
        load(uri, callback);
      }
    }
  };
}($, When));
facade.load("http://example.org/42", function () {
  console.log(42);
});
facade.load("http://example.org/1337", function () {
  console.log(1337);
});
// Console will always log 42 first, 1337 second
```

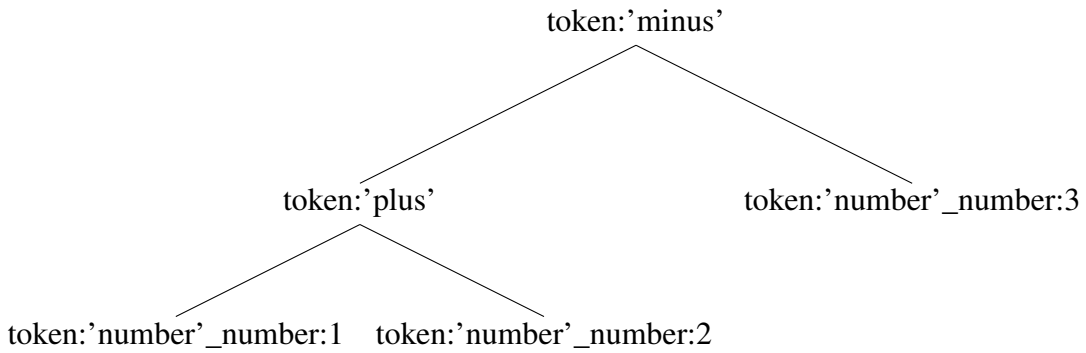

Figure 2.13: A tree-structure representing the equation "1+2-3".

2.3.8 Observer

The Observer pattern "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [18]. This is useful when objects are dependent to know when a dependency changes state, as they will be notified when change happen.

I have included a simple example in listing 2.40. This can be evolved into context-aware notifies, so that observers only are notified when certain things happen. Note that I have made

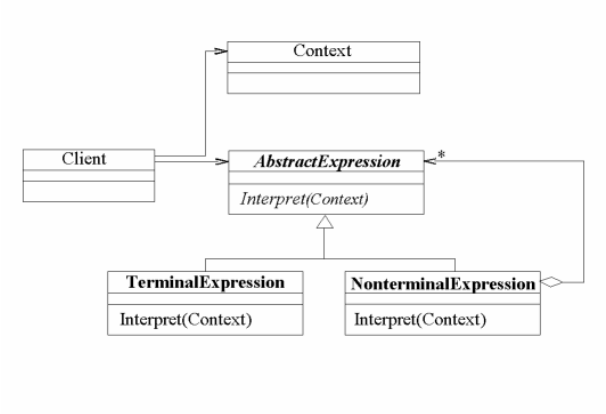

Figure 2.14: Structure of Interpreter

use of closure (section 2.2.2.1) in this example, as the object called upon in line 22 actually is the object first passed in the `obsObject` construct-function (line 19).

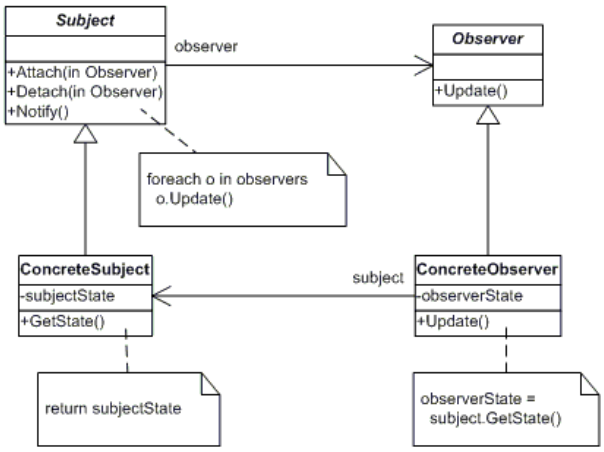

Figure 2.15: Structure of Observer

2.3.9 Prototype

The Prototype pattern "specify the kinds of objects to create using a prototypical instane, and create new objects by copying this prototype" [18]. The variation from how JS handles prototypical inheritance (section 2.2.1.1) is slight, as JS handles prototyping by copying the reference to the prototype object, while the pattern copies the whole thing.

The example given in listing 2.42 shows this in action. Line 18 shows what happens if you compare the prototype object of two objects that have been created using `Object.create` in JS, compared to what happens if you compare the cloned objects. The references are different.

The pattern is not specified in the description of the modules in Graphite, but is included here to show how it differs from prototypical inheritance. It has been used throughout the framework, although as the function named `extend` (that resides in the Utils module). Most often it is used in pair with the parameter option given to functions that offer slight variations from its default behavior. An example is given in listing 2.41.

Listing 2.39: An example of implementation of Interpreter in JS

```
function parseEquation = function (equation) {
  var grammar = {
      minus: function (left, right) { return { token: 'minus', left: left,
          right: right }; },
      number : function (number) { return { token : 'number', number:
          number }; }
      minus: function (left, right) { return { token: 'plus', left: left,
          right: right }; },
    },
    tokens = {
      minus = {
        expression: /^-/,
        evaluate: function (base) {
          var right = tokens.number(base);
          base.eq = base.eq.substring(1);
          return grammar.minus(base.left, right);
        }
      },
      number = {
        expression: /[0-9]+/,
        evaluate: function (base) {
          var value = this.expression.exec(base.eq)[0];
          base.eq = base.eq.substring(value.length);
          return grammar.number(value);
        }
      },
      plus = {
        expression: /^+/,
        evaluate: function (base) {
          var right = tokens.number(base);
          base.eq = base.eq.substring(1);
          return grammar.plus(base.left, right);
        }
      }
    };
  this.left = grammar.number(0);
  this.eq = equation;
  while (this.eq !== "") {
    if (tokens.minus.expression.test(this.eq)) this.left = tokens.minus.
        evaluate(this);
    else if (tokens.number.expression(this.eq)) this.left = tokens.number.
        evaluate(this);
    else if (tokens.plus.expression.test(this.eq)) this.left = tokens.plus.
        evalue(this);
    else throw new Error("No valid expression");
  }
  return this.left;
}
```

2.3.10 Proxy

The Proxy pattern "provide a surrogate or placeholder for another object to control access to it" [18]. There are several kinds of proxies, like the virtual proxy (works like a lazy instantiator,

Listing 2.40: An example of implementation of Observer in JS

```
var obsSubject = function () {
    this.observers = [];
    this.value;
    this.addObserver = function (observer) {
      this.observers.push(observer);
    };
    this.notify = function () {
      var observer;
      for (observer in this.observers) {
        observer.update();
      }
    };
    this.getValue = function () { return this.value; };
    this.setValue = function (val) {
      this.value = val;
      this.notify();
    };
  },
  obsObject = function (subject) {
    subject.addObserver(this);
    this.update = function () {
      console.log("New value: " + subject.getValue());
    }
  },
  mySubject = new obsSubject(),
  myObj1 = new obsObject(mySubject),
  myObj2 = new obsObject(mySubject);
mySubject.setValue(42); // logs "New value: 42" two times
```

Listing 2.41: Altering a functions behavior by extending its configuration with the parameter named option

```
// assume a global function extend that functions like the Prototype
    pattern
var myConfigurableFunction = function (options) {
  var defaultConfig = extend({
    configurationA: true,
    configurationB: 42
  }, options);
  /* Rest of the functions body */
};
myConfigurableFunction({ configurationB: 1337 }); // overwriting the
    default value 42
```

i.e. only creating the proxied object when you need it), remote proxies (proxies an object on a remote destination), and controlling proxies (to handle access), and they may be combined.

In listing 2.43 I have constructed a remote proxy. The object it proxies has one single purpose, which is to fetch the resource located at a given IRI. This may be one way of circumventing SOP (section 2.2.5.1).

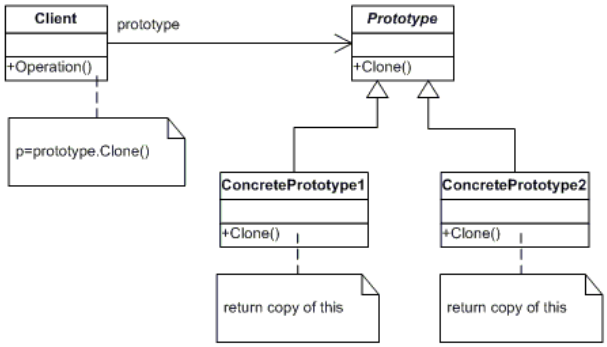

Figure 2.16: Structure of Prototype

Listing 2.42: An example of implementation of Prototype in JS

```
var Prototype = {
  methodA: function () { /* ... */ },
  objectA: { /* ... */ },
  propA: {  /* ... */ }
};
function clone (obj) {
  var o = {}, key;
  for (key in obj) {
    if (typeof obj === "Object") o.__proto__[key] = clone(obj[key]);
    else o.__proto__[key]  = obj[key];
  }
  return o;
}
var a = Object.create(Prototype),
    b = Object.create(Prototype),
    c = clone(Prototype),
    d = clone(Prototype);
console.log(a.__proto__ === b.__proto__); // logs true
console.log(c.__proto__ === d.__proto__); // logs false
```

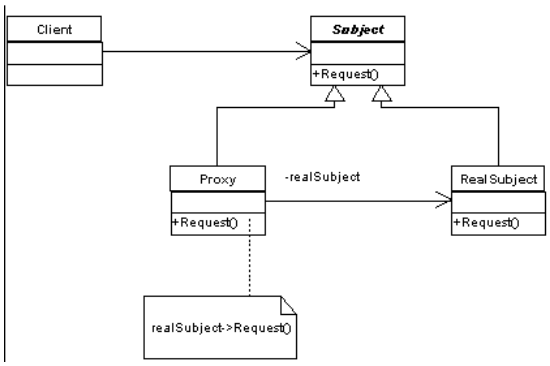

Figure 2.17: Structure of Proxy

2.3.11 Strategy

The Strategy pattern “define a family of algorithms, encapsulate each one, and make them interchangeable” [18]. It relies on a shared interface of properties among objects, and an example

Listing 2.43: An example of implementation of Proxy in JS

```
// assumes function ajax, that acts like $.ajax
var proxy = function (iri, callback) {
  ajax(/localproxy/, {
    data: {
      iri: iri
    },
    method: get,
    success: callback
  });
};

proxy("http://another.domain.com/42", function (data) {
  console.log(data); // logs whatever was fetched from http://another.
      domain.com/42
});
```

is shown in listing 2.44[36].

This pattern streamlines the functionality by eliminating conditional statements.

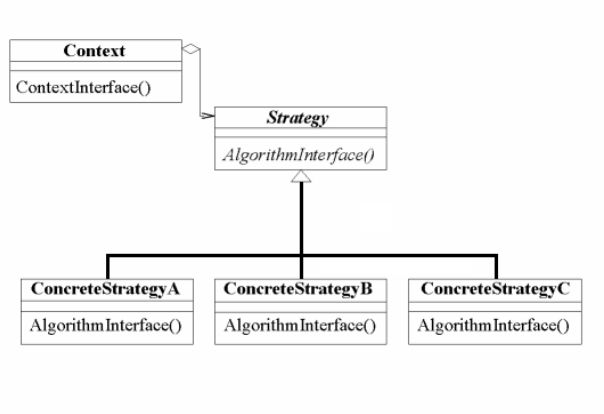

Figure 2.18: Structure of Strategy

2.4 Test-driven development (TDD)

TDD is a development process of software that details how to build your code base. It relies on a iterative cycle of steps that are summarized in figure 2.19. The process start with writing a test that asserts the functionality we wish to implement. It should raise a red flag when first tested (meaning that the functionality is not implemented yet), which in turn leads us to implement the requested feature. When we manage to get a green flag (meaning that the functionality now exists), we can continue to either write a new test for a new functionality, or we can refactor the existing code by making sure it does not raise any red flags (break any tests).

TDD are discussed in sections 8.3 and 8.4, to discuss things I have learned in the progress of implementing the framework.

[36]Inspired by the example given by Mike Pennisi in his blog post at `http://weblog.bocoup.com/the-strategy-pattern-in-javascript/`

Listing 2.44: An example of implementation of Strategy in JS

```
var buttons = [{
      id: "Button42",
      onclick: function () {
        console.log(42);
        return false;
      }
    }, {
      id: "Button1337",
      onclick: function () {
        console.log(1337);
        return false;
      }
    }],
    button;
for (button in buttons) {
  document.getElementById(button.id).onclick(button.onclick);
}
```

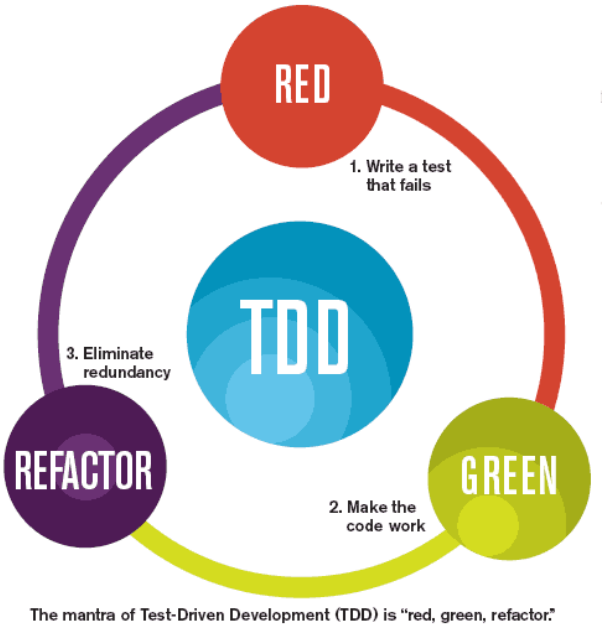

Figure 2.19: An illustration of the TDD-process.

Chapter 3

Problem Description and Requirements

The notion of RDF as a standard for exchanging structured data on WWW is becoming increasingly popular. The technologies of SW are actively developed, and new features, as well as stabilizing old ones, are in the works.

JS has also gotten a lot of attention as an increasingly powerful programming language for WWW. First and foremost as a client-side scripting language, but now also as server-side implementations. Large companies like Google, Microsoft, Mozilla, Apple, and Opera are all putting a lot of effort into increasing the effectiveness of their JS-engines, through implementations and cooperating in evolving the standards.

In this environment, you would think that many developers would try to access SW with a library written in and for JS. While there are projects trying to create frameworks for accessing, query, and manipulating SW, none of them are the defining prototype of a JS-framework for SW as of yet.

3.1 Problem

This thesis seeks to define what is needed in order to have a powerful framework in JS that can access SW. This goal is divided into four subgoals:

1. The first subgoal is to identify the features that a framework accessing SW needs to support. Which technologies needs to be involved, what obstacles do they introduce, and what are the consequences of implementing them?

2. Next, I need to identify the participants and how they should collaborate. I will try using the knowledge of SDP to describe the components in known and widely utilized language. While doing this, I need to explore how JS conform to the patterns of the various SDPs.

3. My third subgoal is to implement a functional framework in JS. In doing this, I need to identify what features JS offers that are relevant for my framework.

4. My final goal is to develop APIs that exposes the functionality of the framework in a way that is easy for developers to get into.

3.2 What are the components required for the framework?

I have identified a lot of the technologies regarding SW in section 2.1. RDF and its serializations need to be a part of the framework. As documents containing RDFS, OWL, and other vocabularies, are subsets of RDF, a representation of RDF should be enough in terms of representing the model.

After I have implemented a model of RDF, I need to implement some way of letting developers browse, or query, the data. SPARQL is a powerful language that handles this purpose, but does it raise the bar for using the framework unnecessarily high? Will developers new to SW want to tackle SPARQL in addition to all the other technologies they need to learn?

Another important feature of the framework will be the APIs. How should I expose the functionality to developers? Should it be available as one monolithic object, or is there a need to divert it into several smaller objects? This problem is further investigated in section 3.5.

3.3 Which Design Patterns are applicable for the components?

I have decided to use SDP to help me decide how I should model the components. This will hopefully help me in identifying participants, collaborations between the participants, and the consequences of implementing them.

Section 2.3 explained in detail the SDPs I think are appropriate for the framework. But these patterns were originally developed for class-oriented programming languages (e.g. C++, which was used to write the sample code used in Design Patterns). Is it appropriate to apply these patterns to a class-less programming language like JS? And if not fully compatible, are there ways to "tweak" the premises, so that we may use the amassed knowledge of patterns to our advantage?

3.4 Which JS-functionalities are of use for the framework?

What are the challenges the framework will have to deal with when implementing the required components? And how does JS align with these problems? I have described JS in section 2.2 and featured some of the functionality that are relevant for my framework.

Serializations will most likely need to be loaded asynchronously. This is handled by the asynchronous loading capacities described in section 2.2.5. I also need certain functions to be called in correct order in response to the asynchronous functionality. By making use of the functional features of JS (section 2.2.1.3), in combination with the promise pattern (section 2.2.6.1), I believe this to be achievable.

3.5 How should the API be designed?

Although SDPs gives a lot of hints as how to design the components, most of the signature of the objects are still up to grab. What are the possibilities we have within the restrains of SW and JS. What is the most effective API to expose to JS-developers that wish to harness the structural data in SW?

How big should it be? How much of the API should be public, i.e. how granular should the functions be? Should it be modularized, or just offer one monolithic API?

These are questions I hope to offer an answer to in my implementation of an actual, working framework.

Part II

Implementation

Chapter 4

Tools

This chapter will describe the software and services we have used in this thesis.

4.1 Buster.JS (Buster)

Buster is a "JavaScript test framework for node and browsers"[1]. It has been in beta for a couple of years, and shows promising results. The lead developers hope to have it ready by the end of this summer.

Some important features of Buster are:

- **Supports tests for Node and browsers:** You can use the same test cases for both environments, but also create dependencies by the use of feature detection.
- **Test utilities:** Assertions, refutations, stubs and spies, expectations, events, properties, and supporting utilities and objects; There are a lot of possible ways to test your code.
- **Flexibility:** There is a lot of public APIs, which can be used to write specific code as needed.
- **Extensibility:** There are already several extensions available, such as buster-amd and buster-lint.
- **Asynchronous testing:** Buster can test resources that only are available in asynchronous functions.
- **Structuring:** Nested test cases, setup, and teardown helps you structure and reuse code easier.
- **Deferred tests:** Easy seclusion of tests that clutters your reports, e.g. while refactoring.
- **Measuring time:** It reports in milliseconds the duration of each test run.

Graphite has used Buster extensible throughout its development, and now sports 33 test cases, with 456 tests, with a total of 1607 assertions. It is all being run in about 10 seconds.

[1] `http://busterjs.org/docs/overview/`

Buster is available at GitHub (GH)[2], and is being led by August Lilleaas and Christian Johansen.

4.1.1 Browsers

Throughout the testing with Buster I have used Chrome for Linux. The last run with Firefox shows that 4 (out of 1607) tests fail in Firefox. I have not tried testing on browsers on other platforms recently, because:

1. Buster is not available for Windows yet (and I have not been able to access a Mac for setting up my tests), and

2. Many of the tests require resources to be loaded with XHR, which makes using Busters test-server to run tests on other computers somewhat hazy.

4.1.2 Node.js (Node)

Buster is dependent on Node, and as such has used it throughout the development. Node is "a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications.

Node supports a great list of features, amongst them:

- **Modules:** A simple module loading system.

- **Networking utilities:** Setting up servers (e.g. with HTTP, Hypertext Transfer Protocol Secure (HTTPS) and net), creating sockets (e.g. with net and User Datagram Protocol (UDP)), lookup Domain Name System (DNS), handling URLs and queries.

- **File System utilities:** File I/O are simplified with modules such as File System, path, and os.

- **Binary Data utilities:** APIs for handling streams and buffers, easing the handling of transmitting binary data on WWW.

- Much, much more.

Node is available at `http://nodejs.org` and latest versions at GH[3]. Originally created by Ryan Dahl in 2009, it is now sponsored by Joyent, his employer, and enjoys contributions of many, many developers.

[2] `https://github.com/busterjs/buster`
[3] `https://github.com/joyent/node`

4.2 RequireJS (Require)

Require is ”a JavaScript file and module loader“. It allows modularization of JS by deploying functionality throughout several files, or modules. It ties it all together by supporting technologies such as AMD (explained in section 2.2.8.3).

Below are some of the features of Require:

- Loading of modules through AMD for browsers, Rhino (an implementation of JS written in Java[4]), and Node.
- Code optimizing.
- Threading by utilizing Web Workers.

Graphite has used Require to include the vast list of modules as they are needed.

4.3 Git

Git is a free and open source Distributed Version Control System (DVCS), originally created by Linus Torvalds as a response to existing Version Control Systems (VCSs) and Source Control Managements (SCMs). It is fast, reliant, and relatively easy to use. It offers a variety of features, such as:

- Setup, configuring, getting and creating projects.
- Branching and merging, sharing and updating.
- Inspect and compare code.
- Much, much more[5].

Graphite has used Git to share its code base, making it available for all to use and contribute to.

4.3.1 GitHub (GH)

GH is a service that lets you share and collaborate with developers[6], be they friends or strangers. It is a social network that uses Git as its technological foundation, making it easier to connect with other developers. It supports features such as:

- User and organization-accounts.
- Free git-repositories (as long as they are open source).
- Secure transmissions.

[4] `http://www.mozilla.org/rhino/`
[5] For complete list, see the documentation at `http://git-scm.com/docs`
[6] `https://github.com/about/`

- Utilities for presentation of text and code, such as pages, wikis, gists (usually to present snippets of code), issues management, and graphs (visual presentations of data, such as contributors, commit activity, etc).

Graphite has used GH as a central repository for its code base, which is available at `https://github.com/megoth/graphitejs`.

4.4 WebStorm (WS)

WS is a JS Integrated Development Environment (IDE) that is available for Windows, Mac OS, and Linux. It offers a wide array of tools for developing with JS, such as:

- Refactoring.
- Structuring code consistently.
- Integration with GH, Node, and JsTestDriver (test framework for JS, developed by Google).
- Smart duplicated code detector.
- Much, much more[7].

The development of Graphite has been done primarily in WS.

WS is being developed by JetBrains, is primarily licensed, but offers free licenses for educational and open source purposes. It is available at `http://www.jetbrains.com/webstorm/`.

[7] http://www.jetbrains.com/webstorm/features/index.html

Chapter 5

Used Libraries

Don't Repeat Yourself (DRY) is a well-known Three-Letter Acronym (TLA) amongst developers. Another similiar but not so known TLA is Don't Repeat Others (DRO). In designing Graphite I have tried sticking to these principles (amongst others), which have resulted in making use of some third party libraries.

AMD introduced some restrictions to how I could modularize the components I wanted to reuse, and as such there has been some rewriting across the board. The level of rewriting have varied, but unless otherwise noted in the descriptions of the modules in chapter 6, it limits itself to the necessary steps to make it compatible with the AMD pattern.

5.1 Branches

In order to make it clearer where the code in Graphite origins, I have divided all modules into branches. The modules within the Graphite branch represent original code or components which are modified in a degree that makes them differ significantly from their original counterpart. The four other branches are rdfQuery (RDFQuery), rdfstore-js (RDFStore), Underscore.JS (Underscore) and When, and table 5.1 shows the distribution. The branches are explained in the remaining sections of this chapter.

5.2 rdfstore-js (RDFStore)

RDFStore describes itself as "a pure JavaScript implementation of a RDF graph store with support for the SPARQL query and data manipulation language"[1]. It supports a range of features:

- Works in browsers as well as server-side.
- Full support of SPARQL 1.0 and SPARQL Update Language, and partially SPARQL 1.1 (including partial support for property paths).
- Parsers for JSON-LD, Turtle, and N3.
- Implemented W3C RDF Interfaces API[2].

[1] https://github.com/antoniogarrote/rdfstore-js/#readme
[2] http://www.w3.org/TR/rdf-interfaces/

Graphite	RDFQuery	RDFStore
API (section 6.1) Graph (section 6.5) Graphite (section 6.6) JSON-LD (section 6.12.1) Loader (section 6.7) Proxy (section 6.7.1) Query (section 6.8) Query Parser (section 6.9) RDF (section 6.10) RDF JSON (section 6.12.2) RDF Loader (section 6.11) RDF Parser (section 6.12) SPARQL (section 6.9.1) XHR (section 6.7.2)	CURIE (section 6.2) Data-type (section 6.3) RDF/XML (section 6.12.3) Turtle (section 6.12.4) URI (section 6.15) Underscore Utils (section 6.16) When Promise (section 6.13)	Abstract Query Tree (section 6.4.1) B-Tree (section 6.14.1) Backend (section 6.5.1) Callbacks (section 6.4.2) Engine (section 6.4) Lexicon (section 6.5.2) Query Filters (section 6.4.3) Query Plan (section 6.4.4) RDF JS Interface (section 6.4.5) SPARQL Full (section 6.9.1.1) Tree Utils (section 6.14)

Table 5.1: Overview of branches and their modules.

- Have an experimental implementation of RDF graph evens API.
- Custom filter functions.
- Threaded API if WebWorkers are supported.
- Persistent storage with HTML5 LocalStorage (for browsers) or MongoDB (for Node).
- Implementation of the SPARQL Protocol for RDF[3] on the server-side.

Graphite has used some of the components of this library, and rewritten them to fit into the overall system. The components that have been used can be seen in table **??**.

RDFStore is available as a repository at GH[4], and has a nice pace of development. Its author, Antonio Garrote, also lists Christian Langanke as contributor.

5.3 rdfQuery (RDFQuery)

RDFQuery describes itself as "an easy-to-use JavaScript library for RDF-related processing". It depends on jQuery, and is distributed in three versions:

1. **Core RDFQuery:** Creates and queries triplestores.
2. **RDFQuery with RDFa:** Parses RDFa.
3. **RDFQuery with rules:** Enables reasoning with rules.

[3] `http://www.w3.org/TR/rdf-sparql-protocol/`
[4] `https://github.com/antoniogarrote/rdfstore-js/`

Graphite has integrated the its parsers as well as several utility-components. A complete list can be seen in table **??**.

RDFQuery has several contributors to its official project-page[5], which is led by Jeni Tennison, Rene Kapusta, and Haymo Meran. After a long time of development seemingly going dead, it now has a repository on GH[6], which seems to be led by Sebastian Germesin. The repository is a mirror of the original project, and all contribution to the GH-project are contributed back to the official project-page.

5.4 when.js (When)

When is a ”lightweight CommonJS Promises/A and when() implementation“[7]. It allows usage of the Promises pattern (section 2.2.6.1), and also provides several other useful Promise-related concepts. It is being developed by Brian Cavalier, and is available at GH[8].

The module named Promise is an integration of When into Graphite, and further explanation of the module is in section 6.13.

5.5 Underscore.JS (Underscore)

Underscore is a ”utility-belt library for JavaScript [which] provides about 80 functions“[9]. Although JS brings a lot to the table in terms of flexibility and a growing set of APIs (both client-side and server-side), it still lacks somewhat when it comes to utility functions. Underscore is a response to this, and provides handy functions that either functions as shivs (e.g. for older browsers that do not support `forEach` yet) or new altogether. Further explanation of the usage of Underscore is in section 6.16.

[5] `http://code.google.com/p/rdfquery/`
[6] `https://github.com/alohaeditor/rdfQuery`
[7] `https://github.com/cujojs/when#readme`
[8] `https://github.com/cujojs/when`
[9] `http://underscorejs.org/`

Chapter 6

The Graphite Framework

This chapter will list all the modules that have been implemented. They are listed alphabetically, but some are nested within others. The first level of modules are I have named main modules, while those nested within them are submodules. I have nested a module if its dependent only by a certain group of modules, with the main module being dependent by other modules.

The names should reflect their purpose, and as such should give an intuitive hint of what they can do. Apart from their names, all modules have a list of features that are presented in the beginning of their section. The list contains the following attributes:

1. Branch: The branch in which they reside (explained in section 5.1).
2. Location: The address to which they are located in the src-folder.
3. Dependencies: Lists which modules, if any, that the module are dependent on.
4. Size: The size in kilobytes (kBs).
5. Source lines of code (SLOC): The number of source code lines.
6. Design Pattern: The design pattern(s) that have been used as a starting point for this module, if any (not applicable for third party derived modules).
7. Test result: If tests are written for the module, its results are listed here. A complete overview of the test results can be found in appendix B.

After the initial block detailing the attributes, a description explains the module in detail. Considerations taken along the development will be noted, and variations/possibilities explained.

Dependencies between the main modules have been visualized in figure 6.1. The dependencies within each subdomain of modules are listed in their designated main module.

The source can be forked at `https://github.com/megoth/graphitejs`.

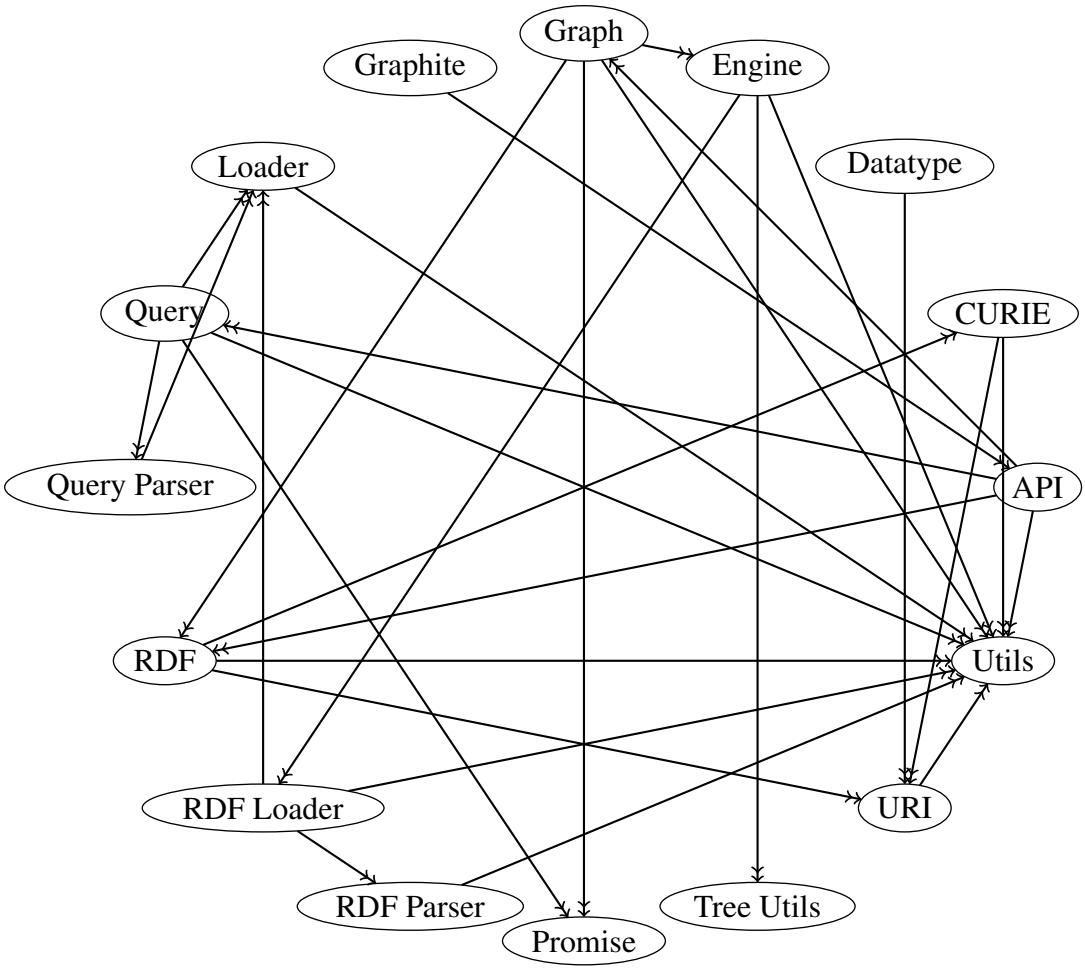

Figure 6.1: Dependencies between the main modules of Graphite.

6.1 API

Branch	Graphite
Location	`graphite/api.js`
Dependencies	Graph (section 6.5) , Query (section 6.8) , RDF (section 6.10) , Utils (section 6.16) , Promise (section 6.13)
Size:	4.4 kB
SLOC	121
Design Pattern	Bridge (section 2.3.2) , Facade (section 2.3.6)
Test result	10 tests, 11 assertions; Total average: 1968, Average/assertion: 179

The API module tries to combine the most powerful modules of Graphite into one module, for easier access to developers new to the framework. The idea is to lower the barrier by combining several modules into one, and built upon their functionality to create new ways of handling the data.

This module differs from the Graphite module in that it acts as a Facade-object for the underlying modules, instead of a simple connection to them. Its signature mirrors in many ways it underlying modules, but adds some methods of its own. In most of the cases though the mapping are one-to-one, which should make a transition from the API module to the Graph- or

Query module easy.

At the heart of the module are the properties `g` and `q`, which respectively are instantiations of the Graph- and Query module. This core properties enables the user to cache data from the SW, and query it. The query can be built piece by piece, until it are executed with the execute-method.

This module partakes in the Bridge pattern as the Implementor, where Graphite works as the Abstraction. It is designed to be easily switched if a system architect wishes to customize the API he wants to serve his team of developers.

6.2 CURIE

Branch	RDFQuery
Location	`rdfquery/curie.js`
Dependencies	URI (section 6.15) , Utils (section 6.16)
Size:	10.5 kB
SLOC	84
Design Pattern	None
Test result	13 tests, 13 assertions; Total average: 61, Average/assertion: 5

The CURIE module handles functions regarding Compact URIs (CURIEs)[1], which are quite common when working with SW, as it eases the task of remembering IRIs, in turn helping to reduce typing errors. The functions are split into creating IRIs from CURIEs or vice versa.

CURIE is not written with a SDP in mind, as it is taken from a third party library. It could be argued it is a utilization the Builder pattern, but the strings it returns can hardly be called complex objects.

6.3 Data-type

Branch	RDFQuery
Location	`rdfquery/datatype.js`
Dependencies	URI (section 6.15)
Size:	14.5 kB
SLOC	260
Design Pattern	Strategy (section 2.3.11)
Test result	12 tests, 22 assertions; Total average: 56, Average/assertion: 3

The Data-type module returns a simple function that returns an object representing a data-type. It is used by the RDF module when handling literals. In addition to the callable function there is also a static function `valid` available, that enables testing whether or not a given value is valid according to a given data-type.

The module uses the Strategy pattern within itself (i.e. no collaboration with other modules). This is done by giving the different data types each a representation with an object containing

[1] `http://www.w3.org/TR/curie/`

the properties `regex`, `strip`, and `value`, and in some cases `validate`. The Context is in this case either the constructor-function, or the static function `valid`.

6.4 Engine

Branch	RDFStore
Location	`rdfstore/query-engine/query_engine.js`
Dependencies	Abstract Query Tree (section 6.4.1) , Tree Utils (section 6.14) , Query Plan (section 6.4.4) , Query Filters (section 6.4.3) , RDF JS Interface (section 6.4.5) , RDF Loader (section 6.11) , Callbacks (section 6.4.2) , Utils (section 6.16)
Size:	71.7 kB
SLOC	1543
Design Pattern	Builder (section 2.3.3) , Facade (section 2.3.6)
Test result	53 tests, 312 assertions; Total average: 718, Average/assertion: 2

The Engine module is a complex module that brings together several submodules, in effect being an implementation of the Facade pattern. Its purpose is to execute queries, and does so by iterations of compiling data, that results in either a formula (as specified in the RDF module), a list of objects with projected variables, or a boolean (depending on the query form, as explained in section 2.1.6.1).

The Builder pattern can be used to understand this module, although it is somewhat hazy. The engine participates as the Director, and RDF JS Interface, Query Filter, and Query Plan all collaborate as Builders. The Product in this case is the result of a query, and it is here it becomes clear that the implementation is not complete, as it is the engine itself that serves the means of getting the Product.

Figure 6.2 shows the dependencies amongst the submodules of the Engine module. Some of the main modules are also represented (i.e. Query Parser, RDF loader, Tree Utils, and Utils), as they have been used by the submodules.

6.4.1 Abstract Query Tree

Branch	RDFStore
Location	`rdfstore/query-engine/abstract_query_tree.js`
Dependencies	Query Parser (section 6.9) , Tree Utils (section 6.14) , Utils (section 6.16)
Size:	27.1 kB
SLOC	540
Design Pattern	None
Test result	18 tests, 90 assertions; Total average: 130, Average/assertion: 1

The Abstract Query Tree module is based on the draft of The SPARQL Algebra[2], and does not apply any SDPs as I can see. Again, the code align closely to the Builder pattern, and it should not be to hard to refactor the module. Another pattern that could easily be applied is the Strategy pattern. I will return to this point in the discussion (section 8.1.1.2).

[2] `http://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html`

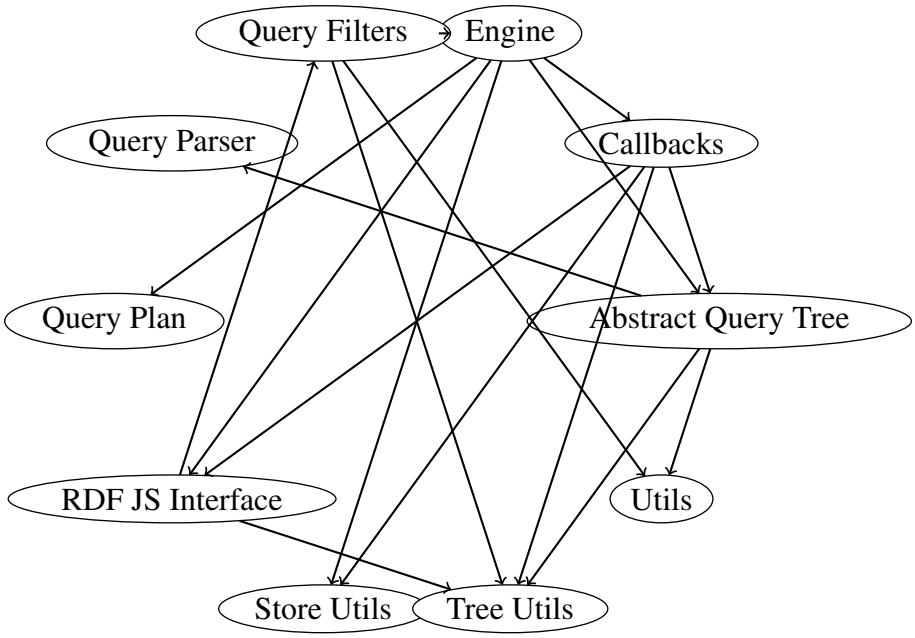

Figure 6.2: Dependencies between the Engine-related modules of Graphite.

6.4.2 Callbacks

Branch	RDFStore
Location	`rdfstore/query-engine/callbacks.js`
Dependencies	Abstract Query Tree (section 6.4.1) , RDF JS Interface (section 6.4.5) , Tree Utils (section 6.14)
Size:	18.8 kB
SLOC	437
Design Pattern	Builder (section 2.3.3)
Test result	7 tests, 25 assertions; Total average: 102, Average/assertion: 4

The Callbacks module is a submodule of the Engine module, and handles the order in which queries should be fired. The module participates in the Builder pattern in collaboration with the RDF JS Interface module, where Callbacks works as the Director, and Interface works as the Builder. It does it with a twist though, as explained in section 6.4.2.

It could also make use of the Observer pattern, which is discussed in section 2.3.8.

6.4.3 Query Filters

Branch	RDFStore
Location	`rdfstore/query-engine/query_filters.js`
Dependencies	Tree Utils (section 6.14) , Utils (section 6.16)
Size:	79.3 kB
SLOC	1435
Design Pattern	Builder (section 2.3.3)
Test result	15 tests, 29 assertions; Total average: 186, Average/assertion: 6

The Query Filters are utility functions for handling queries. It handles aggregation, function

calls, and other filter expressions as part of a SPARQL abstract tree. As such, the module acts as a Builder for the engine, which acts as a Director, meaning that the module partakes in the Builder pattern.

The module could also benefit from the use of the Strategy pattern, as many of the filter expressions could be handled as interchangeable objects.

6.4.4 Query Plan

Branch	RDFStore
Location	`rdfstore/query-engine/query_plan_sync_dpsize.js`
Dependencies	None
Size:	21.6 kB
SLOC	468
Design Pattern	Builder (section 2.3.3)
Test result	1 tests, 12 assertions; Total average: 9, Average/assertion: 1

The Query Plan module handles the different ways you can consolidate the different parts of a SPARQL abstract tree. It takes part of the Builder pattern in collaboration with the Engine, Query Filter, and RDF JS Interface, as previously explained.

An adoption of the Strategy pattern would probably clean up the structure, as well as the Composite pattern.

6.4.5 RDF JS Interface

Branch	RDFStore
Location	`rdfstore/query-engine/rdf_js_interface.js`
Dependencies	None
Size:	20.2 kB
SLOC	536
Design Pattern	Builder (section 2.3.3)
Test result	5 tests, 20 assertions; Total average: 79, Average/assertion: 4

The RDF JS Interface module implements the API defined in the document RDF Interfaces[3]. It outlines many common RDF terms and sports some functions to help creating them. In this we see what resembles an implementation of the Builder pattern (as mentioned in the Engine and Callbacks module).

[3] `http://www.w3.org/TR/rdf-interfaces/`

6.5 Graph

Branch	Graphite
Location	`graphite/graph.js`
Dependencies	Backend (section 6.5.1) , Engine (section 6.4) , Lexicon (section 6.5.2) , Promise (section 6.13) , RDF (section 6.10) , Utils (section 6.16)
Size:	11.7 kB
SLOC	261
Design Pattern	Strategy (section 2.3.11)
Test result	4 tests, 8 assertions; Total average: 1019, Average/assertion: 127

The Graph module is the cornerstone of Graphite. It is the abstraction of quadstores, and serves as an access point for all the data processed by the framework. Its signature is somewhat small, but what it powers is the execution of SPARQL-queries. By calling its method `execute` with a query (be it an instantiation of the module Query, or a plain String) you can add and retrieve data.

To limit the scope of the thesis, I decided to only support a subset of the SPARQL Query Language and SPARQL Update. The subset are:

- **Query Forms:** `ASK`, `CONSTRUCT`, `INSERT`, `LOAD`, and `SELECT`.
- **Solution Sequences and Modifiers:** `ORDER BY`.
- **Aggregates:** `GROUP BY`.
- **Aggregate Algebra:** `Count`, `Sum`, `Avg`, `Min`, and `Max`.

This means Graphite will only support adding data to the graph, not delete, clear, or update it. Also, subqueries are not supported. This limitation was made to avoid some common problems when dealing with logics in quadstores, as well as limitation imposed by underlying third party code.

The Strategy pattern has been implemented in order to handle the supported forms of queries. It is handled internally, with the function execute fetching a concrete strategy from a map of functions (e.g. `executes["select"]` contains the function that handles results from `SELECT` queries).

An important feature of the Graph module is lazy loading, which secures that the order in which we call resources are handled correctly. This works by making use of the Functional Feature (section 2.2.1.3) combined with the Promise Pattern (section 2.2.6.1). Lazy loading as a design pattern is discussed in section 8.2.2.

The Graphs' and submodules' dependencies are listed in figure 6.3 (Tree Utils are present to show external dependencies).

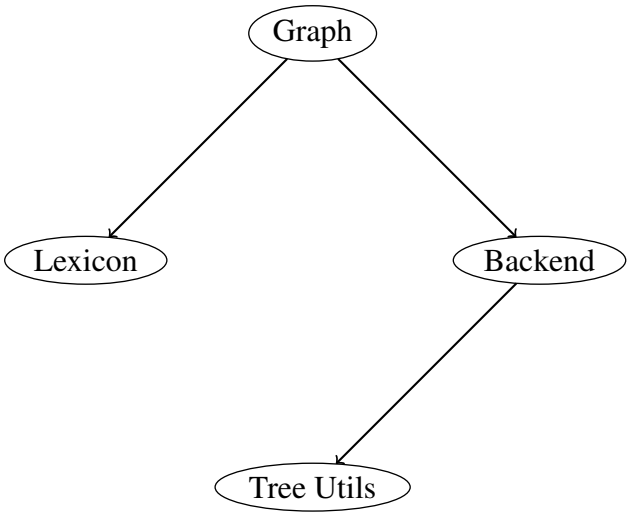

Figure 6.3: Dependencies between the Graph-related modules of Graphite.

6.5.1 Backend

Branch	RDFStore
Location	`rdfstore/persistence/quad_backend.js`
Dependencies	Tree Utils (section 6.14) , Tree Utils (section 6.14)
Size:	4.0 kB
SLOC	89
Design Pattern	None
Test result	2 tests, 57 assertions; Total average: 13, Average/assertion: 0

The Backend module handles the storage of RDF-related data in one graph. It makes use of no SDPs.

6.5.2 Lexicon

Branch	RDFStore
Location	`rdfstore/persistence/lexicon.js`
Dependencies	None
Size:	9.8 kB
SLOC	242
Design Pattern	None
Test result	2 tests, 9 assertions; Total average: 11, Average/assertion: 1

The Lexicon module handles all the graphs that are in play, and resolves terms across graphs. It makes no use of SDPs.

6.6 Graphite

Branch	Graphite
Location	`graphite.js`
Dependencies	API (section 6.1)
Size:	0.5 kB
SLOC	7
Design Pattern	Bridge (section 2.3.2)
Test result	1 tests, 2 assertions; Total average: 7, Average/assertion: 4

The Graphite module is designed to be the main entry point for beginners. It sits at the forefront of the framework (all other modules resides in the folders that are its siblings), and is designed to be easily included into a larger context with an AMD-library. Developers can include this module without knowing anything about it, and start going through tutorials, the documentation, or just play around.

For now it merely returns the API module, but it can be easily extended. One way of doing this is to include the Utils module, which gives the extend-method for objects. By instantiating the different modules whose interface you wish to make highlight, you can combine them into one single object. This could be useful to a system architect who wishes to modify the framework for his project, minimizing the amount of time his developers need to spend to learn the framework. With his alteration he could simply present them with a modified API, that is scissored to their use.

6.7 Loader

Branch	Graphite
Location	`graphite/loader.js`
Dependencies	Proxy (section 6.7.1) , Utils (section 6.16) , XHR (section 6.7.2)
Size:	1.3 kB
SLOC	26
Design Pattern	Strategy (section 2.3.11)
Test result	1 tests, 1 assertions; Total average: 16, Average/assertion: 16

The Loader module fetches resources, and does so depending on what functionality the system supports. All dependent modules prefixed Loader participates in the Strategy Pattern as a ConcreteStrategy.

The dependencies within the submodules of Loader is shown in figure 6.4.

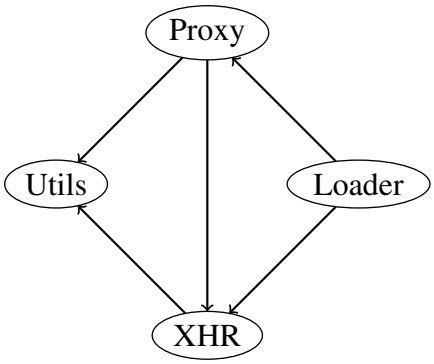

Figure 6.4: Dependencies between the Loader-related modules of Graphite.

6.7.1 Proxy

Branch	Graphite
Location	`graphite/loader/proxy.js`
Dependencies	XHR (section 6.7.2) , Utils (section 6.16)
Size:	1.2 kB
SLOC	36
Design Pattern	Bridge (section 2.3.2) , Proxy (section 2.3.10) , Strategy (section 2.3.11)
Test result	2 tests, 4 assertions; Total average: 26, Average/assertion: 7

The Proxy module is a participant in the Strategy Pattern as ConcreteStrategy. It was created to bypass the Same Origin Policy (section 2.2.5.1) by using a proxy-server on the same domain. It uses the XHR module to make this connection, and if successful, the service would return the data the framework would otherwise be denied.

This does require a service to be set up on the server, which accepts the formatted query which the Proxy sends. Basically it split the IRI to be loaded into separate parts (as explained in section 2.1.4.2). As part of the framework, this service has been created as an application driven by Node.

6.7.2 XHR

Branch	Graphite
Location	`graphite/loader/xhr.js`
Dependencies	Utils (section 6.16)
Size:	1.5 kB
SLOC	40
Design Pattern	Bridge (section 2.3.2) , Strategy (section 2.3.11)
Test result	4 tests, 10 assertions; Total average: 76, Average/assertion: 8

The XHR module makes use of the XHR2-object available in most modern browsers. It makes use of the Strategy Pattern by participating as a ConcreteStrategy to the Loader module.

6.8 Query

Branch	Graphite
Location	`graphite/query.js`
Dependencies	Loader (section 6.7) , Query Parser (section 6.9) , Utils (section 6.16) , Promise (section 6.13)
Size:	12.1 kB
SLOC	292
Design Pattern	Builder (section 2.3.3) , Bridge (section 2.3.2)
Test result	51 tests, 52 assertions; Total average: 512, Average/assertion: 10

The Query module builds a complex structure that aligns the SPARQL abstract tree, which is used in the Engine module. It partakes in the Builder pattern by being the Director-participant, whereas the Query Parser module (and its submodules) is the Builder-participant.

It also shares another collaboration with the Query Parser module, namely through the Bridge pattern. It serves as an API that can be changed independently of the Query Parser.

6.9 Query Parser

Branch	Graphite
Location	`graphite/queryparser.js`
Dependencies	SPARQL (section 6.9.1)
Size:	0.4 kB
SLOC	14
Design Pattern	Builder (section 2.3.3) , Bridge (section 2.3.2) , Strategy (section 2.3.11)
Test result	2 tests, 4 assertions; Total average: 10, Average/assertion: 2

The Query Parser module is designed to be extensible, i.e. if there are other ways of serializing the abstraction of a SPARQL query, than it can be extended with this module (e.g. to differ between SPARQL 1.0 and SPARQL 1.1).

The module is designed with the Builder pattern in mind, by participating as the Builder. The Query module is Director, and decides in which order parts of the SPARQL abstract tree is to be added. It further delegates this responsibility to the chosen strategy, e.g. the module that participates as ConcreteStrategy (while the module itself participates as Context).

Figure 6.5 display the dependencies between the modules partaking in the works of the Query Parser.

6.9.1 SPARQL

Branch	Graphite
Location	`graphite/queryparser/sparql.js`
Dependencies	SPARQL Full (section 6.9.1.1)
Size:	35.7 kB
SLOC	864
Design Pattern	Builder (section 2.3.3) , Strategy (section 2.3.11)
Test result	36 tests, 56 assertions; Total average: 124, Average/assertion: 2

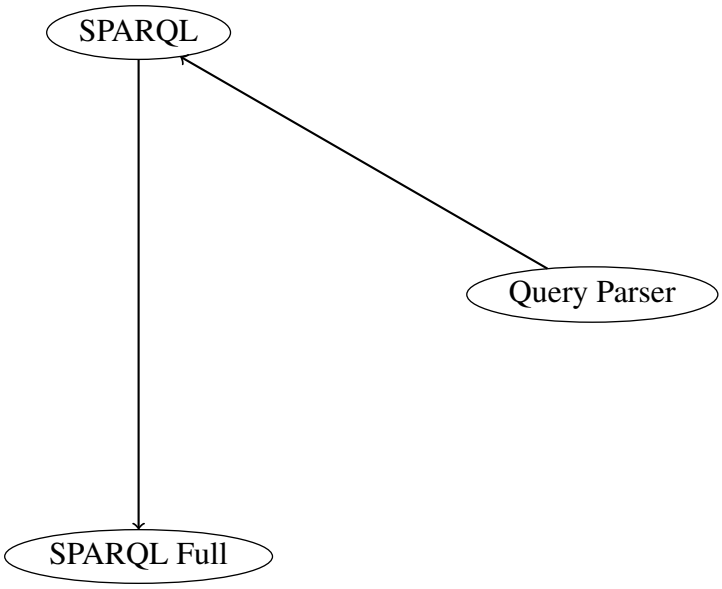

Figure 6.5: Dependencies between the modules related to the Query Parser in Graphite.

The SPARQL module sports an array of methods that allows building parts of the SPARQL abstract tree. The tree has some constraints concerning how it can be structured, and the module takes care of this.

The module partakes in the Strategy pattern as a concrete strategy. As of now it is the only strategy available, which may make the use of the pattern unnecessary.

The module also participates as a Builder in the Builder pattern. This responsibility is delegated from the Query Parser, which is also the same that acts as Context in the Strategy pattern.

As the module does not support parsing of all the elements in the SPARQL abstract tree, it also imports the use of SPARQL Full module.

6.9.1.1 SPARQL Full

Branch	RDFStore
Location	`rdfstore/sparql-parser/sparql_parser.js`
Dependencies	None
Size:	1000 kB
SLOC	19114
Design Pattern	Interpreter (section 2.3.7)
Test result	Not available

The SPARQL Full module is by far the biggest component in Graphite. It is generated by PEG.js, which is a parser generator for JS[4]. That is also why it is much bigger and more complex than it needs to be. But it does parse a complete SPARQL query, and as I have not been able to create a complete one myself, I have implemented it as part of my framework.

The pattern resembles the Interpreter pattern, as there is a representation of the SPARQL grammar, and I use the module to evaluate its representation into a structured tree of terms from

[4] `http://pegjs.majda.cz/`

that grammar.

It works as a starting point for manipulating queries, by feeding it with a complete query, and then modify its parts as necessary through the Query module.

6.10 RDF

Branch	Graphite
Location	`graphite/rdf.js`
Dependencies	CURIE (section 6.2) , RDF (section 6.10) , URI (section 6.15) , Utils (section 6.16)
Size:	16.8 kB
SLOC	372
Design Pattern	Composite (section 2.3.4) , Strategy (section 2.3.11)
Test result	7 tests, 17 assertions; Total average: 34, Average/assertion: 2

The RDF module offers a wide arsenal of methods, and creates a common ground for producing objects pertaining to terms in RDF. Many of the methods origins from the N3-parser in RDFStore, but has been restructured to promote a consistent API. As such, the method `toNT` is represented in all objects retrieved from RDF, and it presents the different terms in N3-compliant syntax (e.g. IRI = <(IRI)>).

The module is used by all the parsers, and the Engine is dependent on the `toQuads` method it appends on all its objects. The objects available through RDF are:

- BlankNode,
- Collection (e.g. a list),
- Empty (i.e. rdfs:nil),
- Formula (i.e. a set of statements),
- Literal,
- Statement, and
- Symbol (e.g. a IRI).

RDF makes use of the Strategy pattern, as all objects listed above have methods `toNT` and `toQuads`, meaning there is no need to test for type or feature to know whether or not they can be called.

RDF also makes use of the Composite pattern, as Collection and Formula will call on their leafs when `toNT` and `toQuads` are called.

6.11 RDF Loader

Branch	Graphite
Location	`graphite/rdfloader.js`
Dependencies	Loader (section 6.7) , RDF Parser (section 6.12) , Utils (section 6.16)
Size:	4.0 kB
SLOC	50
Design Pattern	Facade (section 2.3.6)
Test result	N/A

The RDF Loader module is a very simple module, and does in fact only consist of a single function. The function takes an IRI that can be dereferenced as a graph, the name of that graph, and a function to call when it is loaded. What it passes along is the graph that is been fetched, ready for further processing.

The module acts as facade for the underlying modules that sports a much greater API, and delivers a single, easy-to-use function.

6.12 RDF Parser

Branch	Graphite
Location	`graphite/parser.js`
Dependencies	JSON-LD (section 6.12.1) , RDF JSON (section 6.12.2) , RDF/XML (section 6.12.3) , Turtle (section 6.12.4) , Utils (section 6.16)
Size:	1.4 kB
SLOC	34
Design Pattern	Strategy (section 2.3.11)
Test result	5 tests, 12 assertions; Total average: 168, Average/assertion: 14

The RDF Parser module enables parsing RDF independent of its serialization. For now it needs to be configured by the user to let it know which parser to use, but the goal is to make it detect the serialization on its own.

The module has been designed with the Strategy pattern in mind. By treating all parsers as different strategies to parse RDF, it enables adding and removal additional parsers quite easily (e.g. if we want to be able to parse RDFa). All submodules participates as a ConcreteStrategy.

Another pattern that all submodules use is the Interpreter pattern. The RDF module defines a unified grammar, which they make use of as they evaluate the different serializations.

Figure 6.6 show the dependencies within the RDF Parser modules (RDF, Loader, Promise, URI, and Utils being included to show external dependencies).

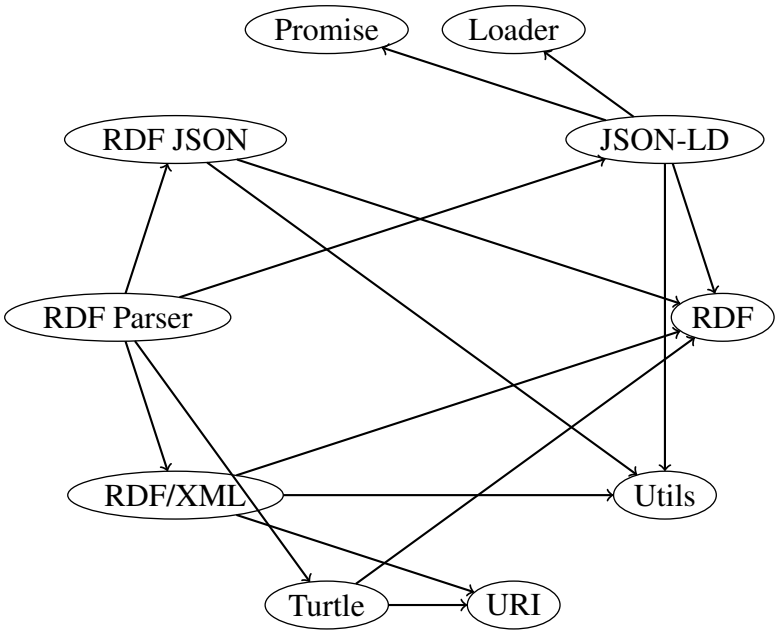

Figure 6.6: Dependencies between the modules related to the RDF Parser in Graphite.

6.12.1 JSON-LD

Branch	Graphite
Location	`graphite/parser/jsonld.js`
Dependencies	Loader (section 6.7) , Promise (section 6.13) , RDF (section 6.10) , Utils (section 6.16)
Size:	17.5 kB
SLOC	361
Design Pattern	Interpreter (section 2.3.7) , Strategy (section 2.3.11)
Test result	22 tests, 63 assertions; Total average: 249, Average/assertion: 4

The JSON-LD module parses JSON-LD into RDF. It also makes use of the Loader and Promise modules as it supports dereferencing URLs that are used in `@context`.

I decided to implement my own JSON-LD parser instead of reusing the one made available by JSON-LD CG[5]. I wanted to get a good understanding of JSON-LD, and though creating my own processor could be a good exercise for doing this. When it was complete, it worked well as part of Graphite, it did the work needed, and as such I deemed it unnecessary to integrate even more third party software into the framework.

But as the specification of JSON-LD continue to evolve, my parser will undoubtedly fall behind, and not be able to parse every possible variation. At this point it would probably be preferable to integrate a specialized library, perhaps using the Adapter pattern.

[5] `https://github.com/digitalbazaar/jsonld.js`

6.12.2 RDF JSON

Branch	Graphite
Location	`graphite/parser/rdfjson.js`
Dependencies	RDF (section 6.10) , Utils (section 6.16)
Size:	1.5 kB
SLOC	39
Design Pattern	Interpreter (section 2.3.7) , Strategy (section 2.3.11)
Test result	8 tests, 9 assertions; Total average: 923, Average/assertion: 103

The RDF JSON module parses RDF JSON.

6.12.3 RDF/XML

Branch	RDFQuery
Location	`rdfquery/parser/rdfxml.js`
Dependencies	RDF (section 6.10) , URI (section 6.15) , Utils (section 6.16)
Size:	14.7 kB
SLOC	264
Design Pattern	Interpreter (section 2.3.7) , Strategy (section 2.3.11)
Test result	29 tests, 136 assertions; Total average: 1785, Average/assertion: 13

The RDF/XML module is taken from the library RDFQuery, and supports about 60% of the tests given in the official RDF/XML test suite[6].

6.12.4 Turtle

Branch	RDFQuery
Location	`rdfquery/parser/turtle.js`
Dependencies	RDF (section 6.10) , URI (section 6.15)
Size:	16.9 kB
SLOC	474
Design Pattern	Interpreter (section 2.3.7) , Strategy (section 2.3.11)
Test result	4 tests, 31 assertions; Total average: 315, Average/assertion: 10

The Turtle module originates from RDFQuery, and supports all of the tests given by the Turtle Test Suite[7].

[6] `http://www.w3.org/TR/rdf-testcases/`
[7] `http://www.w3.org/TeamSubmission/turtle/tests/`

6.13 Promise

Branch	When
Location	`graphite/promise.js`
Dependencies	None
Size:	25.1 kB
SLOC	279
Design Pattern	None
Test result	Not available

The Promise module is an integration of the When library. It implements the Promise pattern (section 2.2.6.1), which gives us additional tools to handle asynchronous calls.

6.14 Tree Utils

Branch	RDFStore
Location	`rdfstore/utils.js`
Dependencies	B-Tree (section 6.14.1)
Size:	13.9 kB
SLOC	380
Design Pattern	None
Test result	2 tests, 4 assertions; Total average: 13, Average/assertion: 3

The Tree Utils sports several handy functions used by many of the components originating from RDFStore. It also contains an implementation of a B+ tree, which is used by the Engine.

6.14.1 B-Tree

Branch	RDFStore
Location	`rdfstore/rdf-persistence/in_memory_b_tree.js`
Dependencies	None
Size:	27.8 kB
SLOC	547
Design Pattern	None
Test result	4 tests, 152 assertions; Total average: 55, Average/assertion: 0

The B-Tree module is an implementation of a generic B-tree, more specifically an adaptation of one made for C[8]. It does not make use of any SDPs.

The module could have been integrated into the Tree Utils module, as it is the only making use of B-tree. But as the purpose of this module is so simple and clear, and it may be that I wish to reuse its function, I have decided to let it stay as an independent module.

[8] `http://www.gossamer-threads.com/lists/linux/kernel/667935`

6.15 URI

Branch	RDFQuery
Location	`rdfquery/uri.js`
Dependencies	Utils (section 6.16)
Size:	9.5 kB
SLOC	179
Design Pattern	None
Test result	74 tests, 99 assertions; Total average: 230, Average/assertion: 2

The URI module originates from the RDFQuery project, and handles many utility-functions used when working with IRI. It applies no SDPs as I can see.

6.16 Utils

Branch	Underscore
Location	`graphite/utils.js`
Dependencies	None
Size:	26.8 kB
SLOC	498
Design Pattern	None
Test result	37 tests, 141 assertions; Total average: 130, Average/assertion: 1

The Utils module is a collection of utility functions used throughout the framework. Almost all of the functions originate from the Underscore project (section 5.5), and as such it does not apply any SDPs.

When I started integrating code from Underscore to the Utils module (section 6.16), the plan was to keep it at the minimum, only importing what I needed. But throughout the development, more and more code got included (and tests accompanying them), until about 40 of them had become a part of the module. They differ from the original code that they do not implement themselves as shivs, meaning that the function `Array.each` is implemented as `Utils.each`, which take a collection as the first parameter.

Chapter 7

The Demo

As part of the development of Graphite my supervisors wanted me to implement an application that showed some of the capabilities of Graphite. I created a music application, that loaded data in different serializations, and that enabled the user to browse this data by search and filtering. It is available as part of the code base[1], and on `http://folk.uio.no/arnehass/music/`.

The demo exists in two versions. Version 1 is built by using the API-module, which works like a facade-object, tying the two most important modules together, namely the Graph- and Query-module. In version 2 the facade is discarded, and the application uses the modules Graph, Query, and Loader directly.

7.1 Structure

The application's data is structured using Turtle and JSON-LD. It uses several vocabularies, as listed below:

- Dublin Core Metadata Initiative (`dc`: `http://purl.org/dc/elements/1.1/`),
- Friend of a Friend (`foaf`: `http://xmlns.com/foaf/0.1/`), and
- Music Ontology Specification (`mo`: `http://purl.org/ontology/mo/`).

In addition, I created my own vocabulary, specific to the demo, prefixed `ma` and localized as `http://example.org/music/v1#`. The terms introduced were:

- `ma:listensTo`: a property stating the relation between a user and a track.
- `ma:spotify`: a property stating the URI a track has to its instance in Spotify, if any.
- `ma:User`: a class, used to state a given resource as a user of the application.

The application uses jQuery to manipulate the DOM, and the jQuery plug-in jQuery.template() to handle the templates.

[1] `https://github.com/megoth/graphitejs/tree/master/demo-music`

Part III

Discussion and Conclusion

Chapter 8

Discussion

Chapter 2 describes the pillars which I have built Graphite upon. What I have learned while designing the framework can be categorized within the intersections of these pillars (as visualized in figure 8.1), and I have structured the discussion based on those intersections. Finally, at the end of the discussion, I discuss related work.

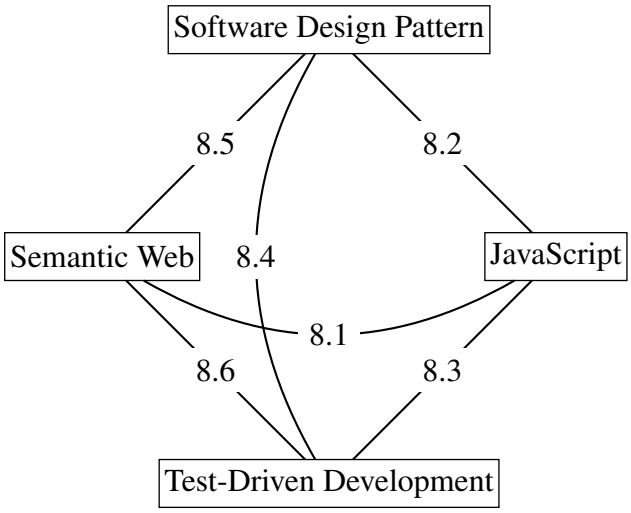

Figure 8.1: Intersections of the pillars of this thesis, as described in chapter 2. The number given is the corresponding section.

8.1 Semantic Web and JavaScript

Graphite has been a great challenge to implement, many obstacles have been put down, only to face even more. This section describes the challenges regarding the intersection of SW and JS.

8.1.1 Representation of Data

The greatest challenge when working with Graphite have been how to structure the data internally. It was not that I had difficulties representing graphs with tree-based structures, but the fact that different components had different requirements from the structures, making reusability more difficult to achieve.

8.1.1.1 RDF

One problem that seemed to pop up again and again were the different representations of RDF. As I included code from third party libraries, I got at least one representation per library. This is not an unusual problem when using code from other projects, but care should be taken to create a component that can be reused easily by other components.

The document RDF Interfaces addresses this problem by defining ”a set of standardized interfaces for working with RDF data in a programming environment“ [27]. RDFStore actually implements this standard (in the RDF JS Interface module), and the engine uses this as its representation of RDF. As of now, Graphite uses one additional representation, situated in the RDF module. The plan is to integrate this module with the aforementioned implementation of RDF Interfaces.

So how should I implement the functionality concerning representations of terms in RDF? I believe the Decorator pattern (section 2.3.5) is a fitting design for this problem. The reasoning goes that there is an implementation of RDF Interfaces acting as the ConcreteComponent, defining a set of terms suitable when working with RDF. This terms can then be dynamically altered by the ConcreteDecorators, that would be modules altering functionality to meet the ones expected by processors.

8.1.1.2 SPARQL

Representing SPARQL has been easier to work with than RDF, as it has been handled by the components that are all part of the RDFStore project. RDFStore makes use of the standards defined by W3C, such as the SPARQL Algebra[1]. The grammar in the algebra maps mostly to the grammar in the SPARQL 1.1 Query Language, and each grammar token is easily represented in JS. This led to the following components:

1. The Query module: A bridge to the Query parser, so that we could change/alter the behavior of the Query module independently of the Query Parser.

2. The Query Parser module: A simple implementation of the Strategy pattern, allowing me to insert other parsers if needed. As of this writing, it only makes use of the SPARQL parser.

3. The SPARQL module: A SPARQL parser with a set of public functions that may parse parts of a query. As I did not have time to create a complete parser, I made use of the parser from RDFStore, which is used when there is need to parse a complete query.

This meant that implementing the Query module would simply mean reuse the various representations of tokens, and inserting them wherever appropriate. However, this endeavor proved harder than anticipated, when I discovered that the SPARQL parser produced two distinct structures of patterns.

This forced me to implement logic that tested which of the trees were used as base for the query, and implement different behavior accordingly. This differences probably also has an effect on the engine (as it support both kinds), and I believe the completion of a SPARQL parser independent of the one from RDFStore will allow easier creation of reusable components.

[1] `http://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html`

8.1.2 Modularity

A feature I surprisingly spent a lot of time on was modularity. One issue was that I needed to support flexibility, since I wanted to return the modules as either:

- A map of functions (i.e. an object without any intrinsic data),
- An instantiated object (i.e. an object with intrinsic data), or
- A function (be it a constructor or not).

In the process I identified five patterns of modular JavaScript patterns, which are explained in section 2.2.8. I decided to go for the AMD pattern, as this was the first pattern supported by the test framework, Buster. This was in spite the fact that neither RDFStore nor RDFQuery used this pattern (they use CommonJS Modules and namespaces respectively). That being said, it was not very hard to convert either.

This has led to some consequences though, that I want to highlight here. The first is that the sheer number of modules are not necessarily a good match with asynchronous loading. When I have tested the demo (chapter 7), loading modules takes about one second. This can be remedied, as Require (amongst others) features a code optimizer. In addition to minifying the code (removing whitespace, shortening names, etc) it also throws all modules into one file (by using it on Graphite it turn about 1.8 MB into approximately 350 kB). The optimizer is not to happy about the uses of regular expressions though, and as of yet I cannot use the optimized code. So I do not know how the framework would have fared if being optimized.

A complication of choosing AMD as the module pattern is that most server-side runtime environments do not support it out of the box. This was considered when I chose to go with AMD, as I focused on creating a framework for the web first and foremost. Luckily, there are tools that enable us to run AMD based modules on server-side (e.g. Require has a module for Node[2]). But that does add complexity and increases the overhead of the framework.

8.1.3 The Engine

The Engine module is probably the most powerful module of all the modules implemented in Graphite, and it is one of the cornerstones of the framework. It is also the most complex and difficult to understand module, and requires further work.

As mentioned in section 6.5, I only support parts of the SPARQL grammar. If SPARQL is to be *the* way to query data in the framework, Graphite need support the complete grammar. The grammar should be represented in an individual module, stripped down to its core representations and functionalities, and should be used by both the engine and the query parsers. If more functionality is needed, it can be implemented by the Mixin or Decorator patterns.

In addition, how it works should be more transparent. I believe one way of doing this to use proven design patterns. I have already mentioned its shaky implementation of the Builder pattern, and how it acts a facade (i.e. using the Facade pattern). But this is probably not a conscious choice, but rather a result of me trying to understand the inner working. I also identified parts of an Observer-pattern, which could be leveraged further.

[2] `http://requirejs.org/docs/node.html`

8.1.3.1 Entailment

Graphite does not support inferring data in any way. When querying, the engine merely looks for patterns, and does no attempt of backward of forward reasoning. I believe that this could be made possible through a plug-in system for the engine. The developer could configure the engine with a specific entailment regime (falling back to simple or no entailment by default), in essence using the Strategy pattern (or maybe the Decorator pattern). This requires a more rigid structure of the engine, i.e. a standard set of functions that plug-in developers could use as hooks.

I believe this task might be too tedious for a web-based application, but could be useful if one were to port the framework to the server-side (further discussed in section 8.1.5), making applications such as SPARQL end-points more powerful.

8.1.3.2 External Service

As of now Graphite only supports working with an internal engine. In case developers want to make use of external processing power, such as to make use of federated queries in SPARQL, a module should be interchangeable with the engine, and as such they should sport the same interface.

8.1.4 Asynchronous Functionality

One important feature I have made use of is the possibility to load resources asynchronously. I have also made it so that the sequence in which queries are inserted are ordered, so querying `INSERT DATA` before `SELECT` will ensure that the data is loaded into the graph before the results for the `SELECT` query are prepared.

In the progress of ensuring this I have been somewhat over-zealous, as I ended up not differing between multiple use of `LOAD` queries. This means that loading data using the `LOAD` query will result in a halt in asynchronous loading, i.e. it will in effect be synchronous (waiting for one resource to have been loaded and processed before starting the other). This should be differentiated, so that the framework can load and deserialize representations so that the data is ready for processing when the engine is available.

Another take on this is to make use of the Observer pattern, and implement the result from SPARQL queries as objects that can be notified when there are changes to the dataset. To give an idea of how this might worked, we can see how the framework AngularJS handles data-binding[3]. They alleviate the need for DOM manipulation on web pages by maintaining a connection between the data presented and the model they are based on. Whenever there is change in the model, that will be reflected in the view. This effect could be useful when working with RDF as well, and would probably be appreciated by developers.

8.1.4.1 XDomainRequest (XDR)

As a warning, I have included this bit on using the XDR object that is available in some of Internet Explorer (IE) browsers. It was designed to handle data-transfer across domains, to

[3]This pattern are also available at other frameworks, such as Backbone (`http://backbonejs.org/`)

overcome SOP (section 2.2.5.1). This is a design it shares with XHR2, but they differ in some important ways.

Most importantly, the developer is not allowed to customize the header sent, i.e. he cannot specify in which format he wishes the data to be replied. Also, GET and POST are the only allowed verbs, and no authentication or cookies are allowed to be sent, meaning modifying through a SPARQL end-point is very difficult. There are other caveats of using XDR[4], but these should be enough to get the message: Stay clear of using XDR to request cross-domain resources.

8.1.5 Server-side implementation

I have restrained myself to limit Graphite to support browser environment first and foremost. But I have not let the prospect of supporting server-side environment go completely. By using feature detection we can test which implementation to use (i.e. we have implemented the modules in question using the Strategy pattern), reusing a lot of the code.

The reason I think a server-side implementation should not be thrown off the table is that I think some functionality are best handled server-side, given challenges such as security, resource allocation, data consistency, and many more. I then believe that supporting both environments would ease the development, as developers would not need to use two different frameworks.

8.1.6 Marketing of SW in JS communities

I have mentioned briefly (in section 2.1.5.6) that JSON-LD CG is skeptical to promote JSON-LD as a serialization of RDF, and by extension promoting as part of the SW standards. This is a legitimate skepticism, as SW has its fair share of skeptics.

One skeptic is Luciano Floridi, who in his publication ”Web 2.0 vs. the Semantic Web: A Philosophical Assessment“ writes ”Regarding the Semantic Web, I argue that it is a clear and well-defined project, which, despite some authoritative views to the contrary, is *not* a promising reality and will probably fail in the same way AI has failed in the past“ [16].

Mike Bergman has another view, making the observation that ”the *structured Web* [...] is a transition phase from the initial document-centric Web to the eventual semantic Web“[5]. He uses the term Structured Web, which I find interesting. I view it as a subpart of what SW is, but much more neutral and applicable in terms of attracting interest from both SW and JS communities.

I tend to agree with JSON-LD CG that SW might be a bit to much to heave upon newcomers. And especially if newcomers come from the JS community, they might not be interested in all the baggage SW offers. They want something that is easy to use, and promotes good practices. SW has a lot going on it for the latter part (being part of W3C, having a good process of standardizing), but still have much to desire on the former.

Now, this thesis is not a philosophical study, nor have I any data in social research of JS communities to make any claim of what is the ”best way™“ to go. I simply offer my thought

[4] A good list is available at `http://blogs.msdn.com/b/ieinternals/archive/2010/05/13/xdomainrequest-restrictions-limitations-and-workarounds.aspx`.

[5] `http://www.mkbergman.com/391/more-structure-more-terminology-and-hopefully-more-clar`

that SW as a whole might be a bit too much for newcomers that are used to working with JS. Put emphasis on what your work do, what problems it solves, and maybe tone down the promises SW offers to solve. Keep it simple and pragmatic. That, at least, is my two cents.

8.2 JavaScript and Software Design pattern

Applying SDPs in JS can be problematic because of the trivial fact that most design patterns are not designed with JS in mind. And this is the way it should be, as truly good, reusable SDPs can be used independently of any programming language. But one feature that many SDPs assume is contracts, i.e. interfaces. JS does not support this, and the closest emulations are objects that tests the presence of properties.

The absence of interfaces leaves us with a choice: do we want to use emulations or drop them altogether. In this thesis I have chosen to go with the latter, as I already test for a modules' properties with unit tests. The fact that I have implemented this framework on my own is also a factor, as there have been no need for a contracted API in order to collaborate easier with other developers. That is a factor that could change though, and preparations should be made.

I believe that documentation is a viable option for communicating a contracted API. Documentation could be spread across multiple documents, depending on the flexibility and social functions needed (e.g. a comment field, to invite others to pitch their ideas). But the continuing life of Graphite is outside the scope of this thesis, and I will not dwell on it further in this text.

8.2.1 Third party libraries

Third party libraries are mentioned this context as it has been a recurring theme that none of the code I have implemented from third party libraries have been structured with SDPs in mind. Although I am not qualified to offer any in-depth analyzes of why this is the case, I do have some hypotheses that may be of interest.

One reason is the simple fact that design patterns do not seem to be very popular in JS. Reasons for this are entirely speculations on my part, but I believe design patterns are rooted in communities that have not opened their eyes for JS yet. I think this will change though, as JS becomes more popular in the professional communities, especially those that have a degree in Computer Science (CS) (i.e. I believe that people having a degree in CS in general are more attuned to the abstract level of solution described by SDPs).

As with many of the topics discussed in this thesis, I also believe SDPs have a case of the egg and the chicken in JS. It is not that popular because there are no great examples of it, and no great examples of it are being developed because it is not popular.

Another reason is the fact many libraries are very small, and very specific to certain tasks. Especially in the Node community there seems to be a widespread philosophy to keep it simple. In those libraries, there are probably no need for SDP. But in order to alleviate the functions of those libraries, SDP can be a helpful guide in how to structure their collaboration (I hope that Graphite can be a good example of this).

8.2.1.1 Absence of the Adapter pattern

I decided to go head on with the third party code implemented in Graphite, i.e. port the whole code instead of creating modules applying the Adapter pattern. The reasons for this was manifolded:

1. Full control of I/O: When implementing the RDF/XML and Turtle parser from RDFQuery, they did not output the data in a way I could easily insert it into the engine.
2. Fewer modules: Applying the Adapter pattern would mean implementing an intermediary module, and increase the number of modules. As discussed in section 8.1.2, this may increase the time it takes to load the framework into the application.
3. Little to no documentation: Both RDFQuery and RDFStore have bad documentation, i.e. it is difficult to read and understand. As such, it felt better to dive into the code, and learn the functionality by porting tests and iteratively adjust it.

8.2.2 Additional SDPs

During the development of Graphite I have stumbled across SDPs beside the ones given in the book Design Patterns. I have restricted myself to referring to this book only, but will mention some of the patterns here, as they have interesting qualities.

Lazy Loading is described by Martin Fowler et.al. as a pattern that ”interrupts [the] loading process for a moment, leaving a marker in the object structure so that if the data is needed it can be loaded only when it is used“ [17]. It may be implemented as lazy initialization, virtual proxy (used by the Proxy pattern), value holder, and ghost.

Lazy initialization has in fact been used in Graphite, in the Graphite module to be exact. As it handles the manipulation offered by the Query module, it can be described as lazy initialization, first triggered when the method `execute` is called.

Addy Osmani structures two additional patterns for JS (besides the ones he has adopted from Design Patterns), namely the Constructor pattern and the Module pattern [24]. I have made use of the AMD implementation, which is described as one of multiple implementations available of the latter. But when it comes to the former, I have trouble calling it a design pattern, as they simple describe the different ways to initialize objects in JS. A useful educational pattern for newcomers to JS, but not really helpful when describing the collaboration between multiple components.

8.2.3 Architectural Styles

Roy Fielding has in his dissertation[6] a ”survey of common architectural styles for network-based application software within a classification framework that evaluates each style according to the architectural properties it would induce if applied to an architecture for a prototypical network-based hypermedia system“ [14]. He has evaluated these styles with into 13 properties (Network Performance, User-perceived Performance, Network Efficiency, Scalability, Simplicity, Evolvability, Extensibility, Customizability, Configurability, Reusability, Visibility, Portability, and

[6]Freely available at `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

Reliability) and describes five categories of style (Data-flow, Replication, Hierarchical, Mobile Code, and Peer-to-Peer).

Graphite is network-based, and it could have been interesting to see if any of the styles described by Fielding could have helped communicating the purpose and work of the components. But analyzing them in the context of Fielding's work is outside the scope of this thesis, and I have included this discussion to highlight alternatives to describe the functionalities of the API.

8.2.4 Representational State Transfer (REST)

The scope of this thesis being mentioned, a style described by Fielding that is interesting to take an extra look upon is REST. The reason for this is that REST is becoming increasingly popular, also within the JS community[7]. There is also work done on REST in RDF[8].

It could be interesting to either extend or use Graphite as part of an application that implements REST. This could become a platform for automating interaction with data structured in RDF, and leverage is usability to something more than a framework.

8.3 JavaScript and Test-driven development

Buster has been a delight to work with, and been a valuable asset to the development of Graphite. I will recommend all to use TDD as a tool to produce good and solid code, and Buster is a good alternative to use. If the syntax is not your cup of tea, there are other viable options, such as JsTestDriver (backed by Google[9]) and Jasmine (behavior-driven[10]).

The definitive gain in using TDD when writing your code is the ease you profit when refactoring code. developing Graphite has involved several rewrites, and failing tests have shown the way to patch things up when something goes wrong. This will also help other developers, if any should join in to collaborate, as they can test that their additions/revisions will not break existing functionality.

There is also the possibility to use test for code coverage (a value that describe to which degree the source code has been tested), with tools such JSCoverage[11]. There is a project to run code coverage in Buster (buster-coverage[12]), but I have not been able to apply it.

---

[7] There are several successful frameworks implementing a RESTful architecture, such as AngularJS (`http://angularjs.org/`), Ext JS (`http://docs.sencha.com/ext-js/4-1/#!/api/Ext.data.proxy.Rest`), and qooxdoo (`http://manual.qooxdoo.org/current/pages/communication/rest.html`).

[8] My own supervisor, Kjetil Kjernsmo, has contributed to this with his paper "The necessity of hypermedia RDF and an approach to achieve it" (available at `http://folk.uio.no/kjekje/2012/hypermedia-rdf.pdf`).

[9] `http://code.google.com/p/js-test-driver/`

[10] `http://pivotal.github.com/jasmine/`

[11] `http://siliconforks.com/jscoverage/`

[12] `http://gitorious.org/buster-coverage`

8.4 Software Design pattern and Test-driven development

As already mentioned in section 8.2, applying SDPs in JS projects may prove difficult since JS do not offer contracts to objects. Use of TDD may remedy this fact, as we can test for properties. One way of implementing this is shown in listing 8.1.

Listing 8.1: Testing for properties in JS

```
function testProperties(obj, properties) {
  var prop;
  for (prop in properties) {
    if (!obj.hasOwnProperty(prop)) throw new Error("Haven't implemented
        property " + prop);
  }
}
// using the Buster framework
buster.testCase("Testing contract", {
  "A test that passes": function () {
    var myObj = { propA: 42 };
    refute.exception(function () {
      testProperties(myObj, [ "propA" ]);
    });
  },
  "A test that fails": function () {
    var myObj = {};
    refute.exception(function () {
      testProperties(myObj, [ "propA" ]);
    });
  }
});
```

The example could be further elaborated, e.g. by automatically check modules that participates as ConcreteStrategy in the Strategy pattern.

8.5 Semantic Web and Software Design pattern

Some parts of SW may not gain much by involving SDPs, but I believe they can be effectively used by implementors that wish to create reusable components. The specifications do not say to much of how to implement the collaboration across the standard they propose, and it is this gap that SDP may be used as social contracts.

8.6 Semantic Web and Test-driven development

Many of the specifications that take part of SW contains test suites, and TDD is perfectly suited to test the development of implementations trying to support these. In many cases it can help identifying problems within implementations, becoming a common, objective ground for developers to discuss solutions.

8.7 Related Work

During my work on this thesis, I have yet to find academic work that tread the same path as I have outlined. But I have found several libraries that tries to solve the same problem Graphite has undertaken, and that may offer interesting features. Two of them, RDFQuery and RDFStore, as I have mentioned several times throughout this thesis, have offered me several reusable components. In this section I will elaborate on their success, independent of their implementation in Graphite.

I have also listed additional libraries of interest, and all in all there are 14 of them. For a complete list of projects I researched in search of related work, see appendix C.

8.7.1 backplanejs

[TBW: `https://code.google.com/p/backplanejs/`]

8.7.2 Javascript RDF/Turtle Parser

[TBW: `http://www.kanzaki.com/works/2006/misc/0308turtle.html`]

8.7.3 jOWL

[TBW: `http://jowl.ontologyonline.org/`]

8.7.4 JS3

JS3 was first committed 19th of November, 2010 and last updated three days later. It describes itself as ”An insane integration of RDF in ECMAScript-262 V5“ [26]. It sports an API for manipulating RDF values and to some extent graphs. But it has no parsing, reasoning or querying capacities.

8.7.5 Jstle

Jstle was first committed 21st of April 2010, and last updated three days later. It describes itself as ”Jstle is a terse JavaScript RDF serialization language“ [23]. It is a proof of concept, and seems to provide a Turtle-like representation of RDF in JS. But no support for parsing, reasoning or querying.

8.7.6 Flint SPARQL Editor

[TBW: `http://openuplabs.tso.co.uk/demos/sparqleditor`]

8.7.7 pushback

[TBW: `http://code.google.com/p/pushback/`]

8.7.8 rdflib.js

rdflib.js seems to be more of a collection of RDF related functionality at the moment than a complete framework for working with RDF with JS. In some cases it also seems like a continuation of RDFQuery, as much of its code resembles a lot (e.g. the use of $rdf as namespace, its dependency on jQuery, and that its structured as a namespace at all). At the moment it features:

- Parses RDF/XML, Turtle, N3, and RDFa.
- Serializes RDF/XML, Turtle, and N3.
- Partial SPARQL support.
- Read/Write Linked Data client, using Web Distributed Authoring and Versioning (WebDav) or SPARQL Update.
- Local API for querying store.
- Can be run server-side with Node.

As mentioned, rdflib.js uses namespaces to structure its code, and this makes it somewhat hard to decouple, not to say reuse. But it does seem to have an active development, and having Tim Berners-Lee contributing is not hurting.

Also, although I have not found any source that confirms it, I suspect rdflib.js to be a continuation of the Tabulator project. Read more in section 8.7.14.

8.7.9 RDFStore

RDFStore seems to me to be the most complete project in terms of API and expressive power. It features partial SPARQL support, some parsers, and it uses some standards (full list of features can be read at section 5.2).

The project has some following[13], but is mostly a one-man project (Antonio Garrote). This may contribute to the fact that the project has a complete API, and has a overall good architecture. It has a steady development since the first commit was made in 17th of February 2011, and nothing implies this to change anytime soon.

The module pattern implemented is CJS modules (section 8.1.2), which makes it a perfect fit for Node, but may also be run in browsers. The project do contain quite messy code at times, which is quite clear in the tests. Also, I suspect the coverage is quite low, as some modules do not have tests at all.

RDFStore is, as it declare in its description, ”still at the beginning of its development“[14], but I believe this project to have a lot of potential.

[13] At the time of this writing, it had been forked 12 times, and had 126 watchers.

[14] `https://github.com/antoniogarrote/rdfstore-js#contributing`

8.7.10 RDFQuery

RDFQuery do seem to suffer from inactive development. It has two code bases, one at Google Code[15] and one at GitHub[16] (the latter mirrors the former, and promises to commit its changes to the original code base). The last change to Google Code was 3rd of September 2011, while the last to GitHub was 21st of June 2011. So it seems that the development has gone somewhat stale.

RDFQuery do seem to be mentioned more often than RDFStore, which may be because the fact that its older (the first commit was 17th of October 2008), and may have been the first JS based project that actually got implemented a big code base. That is at least what seems to be the case during my research into this.

The module pattern implemented is namespaces, which makes it somewhat hard to decouple. It also is dependent on jQuery, i.e. increasing the code overhead, which may be of distaste for some developers.

Although RDFQuery do not have an active development any longer, it still has a lot of treats, and some of it is quite reusable (as proven in Graphite). And it will probably be useful as a reference for other projects, but it does not seem to have any traction of its own anymore.

8.7.11 sgvizler

Sgvizler[17] is a library in JS "which renders the result of SPARQL SELECT queries into charts or HTML elements" [29]. It is a cool display of how data fetched with SPARQL can be presented on web pages. During the development of Graphite, I wanted to include a demo that made use of Sgvizler, but in the end I did not have time for it.

It is important to note that Sgvizler is not a framework for handling data structured with RDF, but rather a presentation tool. Its scope may be narrow, but it does what it does good.

8.7.12 Simple JavaScript RDF parser and query thingy

The development of the Simple JavaScript RDF parser and query thingy seems to be around 5th of November 2005. Its latest version came out 25th of May 2006, and it does not seem to have any big usage. It supports loading and parsing of RDF/XML-documents, and a crude API for querying.

8.7.13 SPARQL JavaScript Library

[TBW: `http://www.thefigtrees.net/lee/blog/2006/04/sparql_calendar_demo_a_sparql.html`]

[15] `http://code.google.com/p/rdfquery/`
[16] `https://github.com/alohaeditor/rdfQuery`
[17] Available at `http://code.google.com/p/sgvizler/`.

8.7.14 Tabulator

The Tabulator project[18] is generic data browser and editor. It is offered in two ways, as a Firefox extension and as a web application. It does not seem to be developed any further, but its code base (whole 120 files of JS) offers a lot of functionality, and it seems that some of it is continued in rdflib.js (such as the files jquery.uri.js and jquery.xmlns.js). Tim Berners-Lee was also involved in the Tabulator project, so this may not come as a big surprise.

[18] Available at `http://www.w3.org/2005/ajar/tab`.

Chapter 9

Conclusion

As part of this thesis I implemented Graphite, a framework that offers an API in JS for accessing SW. I used parts of other projects and sewed it all together to one, functional prototype. It is not complete, further work is required, but it stands as an example of how a framework could look and behave.

Creating a framework offers many challenges. How should the code be structured? If you wish to modularize your functionality, how should you divide it? Do modularity increase the overhead? What are the appropriate ways to make your components collaborate? All these questions, and more, have found their answers in Graphite.

Graphite is an example of how a framework *could* look like. This time I emphasize could, because this is merely one of many possible implementations that can be made. Implementations that have other answers to the challenges they faced. And this fact reveals one of the conclusions this thesis offers: Components should be created with reusability in mind. This is definitely the case with framework configured to handle resources in SW.

SW is like a big cake of standards. Each slice contains a mixture of its ingredients, and even if you were to split them into separate parts, like glaze, it is still best consumed together. The metaphor can also be used to explain why we need multiple parts to make it all work (i.e. to make the cake taste good); some parts (e.g. sodium and flour), is just not good by itself.

SW consists of many standards, and to mix them all together takes some planning. In this thesis I have used SDPs as guides to map participants and collaborations. This proved to be somewhat complex, as there were two factors working against us: One is that JS does not support interfaces, which is used thoroughly in the classic descriptions. Another is that none of the third party code I implemented seemed to have SDPs in mind when implemented, which meant that I had to restructure some code to fit my purpose.

Restructuring is a cumbersome process, which introduces many opportunities to break existing functionality. I used TDD throughout the development to prevent this, which proved effective. It also allowed me to move code across modules, to restructure the very purpose of modules. Patterns have guided me in this process, and I believe it to be an effective route.

The construction of Graphite has been an important part of the realizations revealed in this thesis. If anything, it has shown that the magnitude of SW requires many components to collaborate. It also shows that one framework probably will not ”get“ it all. So to ease the work of developers, both those constructing the frameworks and those using them, care should be made to make components reusable. And to guide how those components should function and

be structured, we use standards, to tap into the work of many, bright people before us.

9.1 Further Work

The engine needs to be decoupled and structured more clearly with SDPs in mind. This would open the code for other developers, and enable them to introduce other algorithms that are more effective. This is particularly important if we want to introduce inferring capabilities.

The SPARQL parser needs to be complete, and the engine needs to be able to process all possible variances. Until that is complete, the framework can not make full use of the querying possibilities of SW.

A module parsing RDFa would be useful, especially if browsers should loosen up on SOP. Also, the RDF/XML parser needs to be completed, and the JSON-LD parser probably needs to be updated (or replaced with another altogether) as the specification becomes a recommendation by W3C.

Storage in browsers could be an interesting functionality to look into, as off-line capacities may become a trend. This could be used as part of caches, and thereby increase the speed of applications that are used multiple times and which do not need to fetch data from an external resource every time.

The engine module should be one alternative of many; The developer should be able to configure the system to use an external SPARQL endpoint as query processor. Graphite supports this to some degree, but as of now it would be more like a hack.

Graphite should be able to be run server-side. As part of this, serializers should be implemented, so that the service can serve RDF in any format requested.

Bibliography

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobsen, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1977.

[2] Dave Beckett. Rdf/xml syntax specification (revised). `http://www.w3.org/TR/rdf-syntax-grammar/`, February 2004. [Online; accessed 11-July-2012].

[3] David Beckett and Tim Berners-Lee. Turtle - terse rdf triple language. `http://www.w3.org/TeamSubmission/turtle/`, March 2011. [Online; accessed 15-July-2012].

[4] Tim Berners-Lee. Semantic web road map. `http://www.w3.org/DesignIssues/Semantic.html`, November 1998. [Online; accessed 10-July-2012].

[5] Tim Berners-Lee. Notation 3 logic. `http://www.w3.org/DesignIssues/Notation3.html`, August 2005. [Online: Accessed 20-July-2012].

[6] Tim Berners-Lee. Linked data. `http://www.w3.org/DesignIssues/LinkedData.html`, June 2009. [Online, accessed 14-July-2012].

[7] Tim Berners-Lee and Dan Connolly. Notation3 (n3): A readable rdf syntax. `http://www.w3.org/TeamSubmission/n3/`, 2011. [Online; accessed 15-July-2012].

[8] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[9] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. `http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf`. [Online; accessed 14-July-2012].

[10] Dan Brickley and R.V. Guha. Rdf vocabulary description language 1.0: Rdf schema. `http://www.w3.org/TR/rdf-schema/`, February 2004. [Online; accessed 11-July-2012].

[11] Douglas Crockford. Javascript: The world's most misunderstood programming language. `http://www.crockford.com/javascript/javascript.html`, January 2001. [Online; accessed 10-July-2012].

[12] Douglas Crockford. The application/json media type for javascript object notation (json). http://www.ietf.org/rfc/rfc4627.txt, July 2006. [Online, accessed 19-July-2012].

[13] Ecma International. *ECMAScript Language Specification*, 2011. [Online; accessed 13-July-2012].

[14] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000. [Available online http://www.ics.uci.edu/~fielding/pubs/dissertation/net_arch_styles.htm, accessed 27-July-2012].

[15] David Flanagan. *JavaScript: The Definitive Guide, Sixth Edition*. O'Reilly Media, Inc., 2011.

[16] Luciano Floridi. Web 2.0 vs. the semantic web: A philosophical assessment. *Episteme*, 6(1):25–37, 2009.

[17] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[19] John Hebeler, Matthew Fisher, Ryan Blace, and Andrew Perez-Lopez. *Semantic Web Programming*. Wiley Publishing, Inc., 2009.

[20] P Hitzler, M Krotsch, and S Rudolph. *Foundations of Semantic Web*. Chapman & Hall/CRC, 2010.

[21] Graham Klyne and Jeremy J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/, February 2004. [Online; accessed 11-July-2012].

[22] Paul Krill. Javascript creator ponders past, future. http://www.infoworld.com/print/39704, June 2008. [Online; accessed 10-July-2012].

[23] D Newcome. dnewcome/jstle - github. https://github.com/dnewcome/jstle, April 2010.

[24] Addy Osmani. *Learning JavaScript Design Patterns*. O'Reilly Media, Inc., 2012. [Online, accessed 19-July-2012; Available as free e-book].

[25] Dirk Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, Eidgenössische Technische Hochschule Zürich, 2000. [Available online http://dirkriehle.com/computer-science/research/dissertation/, accessed 30-July-2012].

[26] Nathan Rixham. webr3/js3 - github. https://github.com/webr3/js3, November 2010.

[27] Nathan Rixham, Manu Sporny, Mark Birbeck, Ivan Herman, and Benjamin Adrian. Rdf interfaces 1.0. `http://www.w3.org/TR/2011/WD-rdf-interfaces-20110510/`, May 2011.

[28] John Godfrey Saxe. *The Poems of John Godfrey Saxe*. Houghton, Mifflin and Company, 1881.

[29] Martin G. Skjæveland. Sgvizler: A javascript wrapper for easy visualization of sparql result sets. *ESWC 2012*, 2012. Is yet to be published, but is to be a demo paper in the workshop/poster/demo proceedings of ESCW 2012.

[30] Architecture of the world wide web, volume one. `http://www.w3.org/TR/webarch/`, 2004. [Online, accessed 14-July-2012].

[31] World Wide Web Consortium (W3C). *OWL Web Ontology Language Overview*, 2004. [Online; accessed 13-July-2012].

[32] Rdf test cases. `http://www.w3.org/TR/rdf-testcases/`, February 2004. [Online; accessed 15-July-2012].

[33] World Wide Web Consortium (W3C). *OWL 2 Web Ontology Language Document Overview*, 2009. [Online; accessed 14-July-2012].

[34] World Wide Web Consortium (W3C). *OWL 2 Web Ontology Language Profiles*, 2009. [Online, accessed 14-July-2012].

[35] World Wide Web Consortium (W3C). *SPARQL 1.1 Query Language*, May 2011. [Online; accessed 16-July-2012].

[36] World Wide Web Consortium (W3C). *JSON-LD Syntax 1.0*, 2012. [Online, accessed 15-July-2012].

[37] Semantic web - w3c. `http://www.w3.org/standards/semanticweb/`, July 2012. [Online; accessed 10-July-2012].

Part IV

Appendices

Appendix A

Code Base

As the code base for Graphite is rather large (approximately 36 000 SLOC, or between 500 and 1000 pages, depending on the format you print it in), I have decided to just refer to the repository at GH.

The complete framework, with all tests and demos, is available at `https://github.com/megoth/graphitejs`.

Appendix B

Test Results

[TBW]

Appendix C

Findings of Related Work

[TBW]