UNIVERSITY OF OSLO
Department of Informatics

A Javascript API for accessing Semantic Web

Master Thesis

Arne Hassel

Spring 2012

UNIVERSITAS OSLOENSIS • MDCCCXI •

Acknowledgments

Abstract

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The SW is a many-faced entity, a colossal structure of standards and resources. It's also an idea shared by a multitude of communities, a concept of structured information, and an abstraction of knowledge. It's 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 it's 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's perhaps more famous for his invention, the World Wide Web (WWW), and he's also the one who coined the phrase Semantic Web. It's 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" [16].

The web of data has been in the making since the late 90s, but in terms of traction there's still much to be done. Some complain it's 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 to tools to access the SW. It's been written in and for 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, a JS-evangelist, has called JS the world's most misunderstood language [2]. And if the name and it's syntax wasn't confusing enought, the browsers with their differing implementations weren't 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 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 is a AMD-based framework written in JS that sports a modularized Application Programming Interface (API) to fetch resources in the SW, process it and output in useful way for JS-developers. And perhaps it will help spark the fume that the standards of SW have set, so that we may answer the question: Is SW ready for broad adaptation?

This master thesis will describe the work and choices made during the implementation of GraphiteJS. It's divided into two 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), which tools we made use of (chapter 4), and finally an extensive presentation of the framework itself (chapter 5). The second part offers a discussion of the work (chapter 6), and a conclusion of the matter (chapter 7).

Chapter 2

Background

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

2.1 Semantic Web (SW)

SW is many things to many people, and seeing the whole picture may not always be so easy. A perhaps fitting metaphor is the story of the elephant and the blind men. It's 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 ment 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 [24]. As such, what are some of the descriptions we have of SW?

- A web of data [16]
- An extension of WWW [23]
- A `killer` app [19]
- W3C's vision of the Web of linked data [13]
- A collaborative movement led by the W3C [14]

The list above are some of the descriptions in literature, and they're all true. Other aspects of SW is the set of standards it sports (e.g. RDF, RDFS, OWL, and SPARQL), technological foundations (e.g. LD), applicabilities (e.g. use of 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's 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 also often called a triple. The three elements have meanings that are analogous to their meaning in normal English grammer [22, p. 68-69], i.e. the subject in a statement is the thing that statement describes. An example of statements and how they are represented as a graph is showed in figure 2.1.

- Arne knows Kjetil
- Arne has lastname Hassel

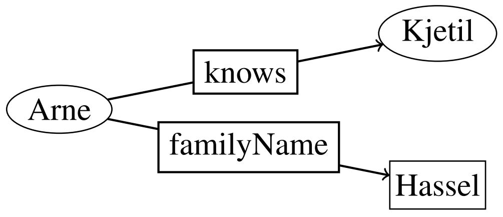

Figure 2.1: A directed graph showing that the subject "Arne" is related to the object "Kjetil" by the predicate "knows", and to the object "Hassel" by the predicate "familyName".

You might've 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 [22, 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 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-readible (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 datatype to a literal [7]. It's power lies in its extensibility by URI-based vocabularies [8]. 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 FOAF, prefixed `foaf`, for the properties).

Not all resources are given IRIs though. The exception to the rule are BNs, which represent resources that haven't any separate form of identification [8] (either because they cannot be named, or it isn't possible and/or necessary at the time of modelling). These resources aren't

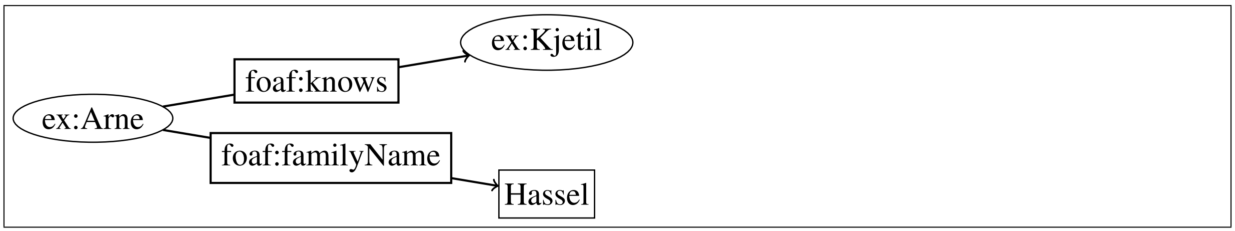

Figure 2.2: Statements from figure 2.1 correctly represented with IRIs.

designed to link data, but to model relations of resources that are given IRIs.

An example of modelling BN is given in figure 2.3, where we've modelled that ex:Arne has a friend, who we don't know anything about except his nicks, Bjarne and Buddy.

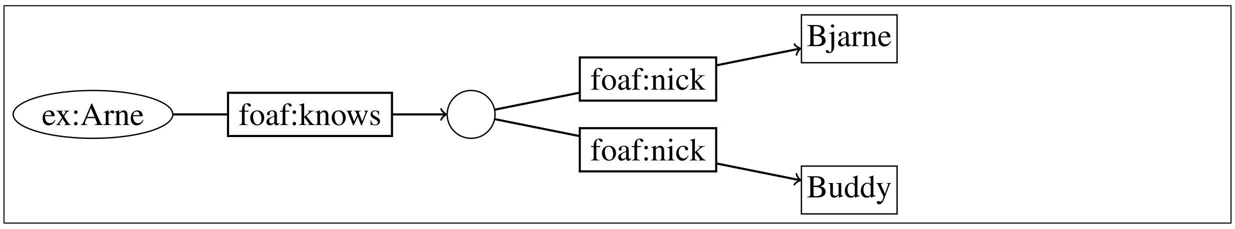

Figure 2.3: A graph containing a BN.

The figures 2.2 and 2.3 are examples of the form of visualization we'll have of RDF-graphs.

2.1.2 Resource Description Framework Scheme (RDFS)

As any other extension of RDF, a RDFS-document is well-formed RDF-document. But RDFS is not a vocabulary in the traditional sense that it covers any topic-specific domain [23, p. 46]. It's 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 instanciating rdf:Statement as a rdfs:Class. It continues by clearifying the semantics of rdf:subject, rdf:predicate, and rdf:object by instanciating them as rdf:Property, and in terms of entailment (explained in section 2.1.6) ties together with rdfs:range and rdfs:domain.

Another extension is the clearification of containers by introducing the class rdfs:Container and the property rdfs:containerMembershipProperty, which is an rdfs:subPropertyOf of the rdfs:member [6].

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 clearifies the use of rdf:value, to encourage its use in common idioms [6].

The classes and properties of RDF and RDFS are presented in table 2.1 and 2.2 respectively.

Class name	comment
rdfs:Resource	The class resource, everything.
rdfs:Literal	The class of literal values, e.g. textual strings and integers.
rdf:XMLLiteral	The class of XML literals values.
rdfs:Class	The class of classes.
rdf:Property	The class of RDF properties.
rdfs:Datatype	The class of RDF datatypes.
rdf:Statement	The class of RDF statements.
rdf:Bag	The class of unordered containers.
rdf:Seq	The class of ordered containers.
rdf:Alt	The class of containers of alternatives.
rdfs:Container	The class of RDF containers.
rdfs:ContainerMembershipProperty	The class of container membership properties, rdf:_1, rdf:_2, ..., all of which are sub-properties of 'member'.
rdf:List	The class of RDF Lists.

Table 2.1: RDF classes [6]

Property name	comment	domain	range
rdf:type	The subject is an instance of a class.	rdfs:Resource	rdfs:Class
rdfs:subClassOf	The subject is a subclass of a class.	rdfs:Class	rdfs:Class
rdfs:subPropertyOf	The subject is a subproperty of a property.	rdf:Property	rdf:Property
rdfs:domain	A domain of the subject property.	rdf:Property	rdfs:Class
rdfs:range	A range of the subject property.	rdf:Property	rdfs:Class
rdfs:label	A human-readable name for the subject.	rdfs:Resource	rdfs:Literal
rdfs:comment	A description of the subject resource.	rdfs:Resource	rdfs:Literal
rdfs:member	A member of the subject resource.	rdfs:Resource	rdfs:Resource
rdf:first	The first item in the subject RDF list.	rdf:List	rdfs:Resource
rdf:rest	The rest of the subject RDF list after the first item.	rdf:List	rdf:List
rdfs:seeAlso	Further information about the subject resource.	rdfs:Resource	rdfs:Resource
rdfs:isDefinedBy	The definition of the subject resource.	rdfs:Resource	rdfs:Resource
rdf:value	Idiomatic property used for structured values.	rdfs:Resource	rdfs:Resource
rdf:subject	The subject of the subject RDF statement.	rdf:Statement	rdfs:Resource
rdf:predicate	The predicate of the subject RDF statement.	rdf:Statement	rdfs:Resource
rdf:object	The object of the subject RDF statement.	rdf:Statement	rdfs:Resource

Table 2.2: RDF properties [6]

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 [4, 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 [9].

Both versions of OWL features three sublanguages/profiles[1]:

- OWL 1, with complexity in increasing order:
    1. OWL Lite (OWLLite): 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 (OWLFull): Maximum expressiveness and the full syntactic freedom of RDF, but with no computational guarantees.
- OWL 2:
    1. OWL Existential Language (OWL EL): Designed to be used with ontologies that contain very large numbers of properties and/or classes [10].
    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 [10].
    3. OWL Rule Language (OWL RL): Aimed at applications that require scalable reasoning without sacrificing too much expressive power [10].

To go through all differences between OWL 1 and OWL 2 would be beyond the scope of this thesis, but sufficient 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, we work with three sublanguages.

2.1.4 Linked Data (LD)

A cornerstone of RDF is that all resources (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 [20].

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

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 [20].

LD are relient on two web-technologies, namely IRIs and Hypertext Transfer Protocol (HTTP). Using the two of them you can fetch any resource adressed by an IRI that uses the http-scheme. When combining this with RDF, LD builds on the general architecture of the Web [3].

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 [20]:

- 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's a movement to publish data on WWW as LOD. Especially toward governmental institutions there's now an increasing trend of opening data[3].

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

[3]Examples of this are platforms such as UK's initiative to open governmental data (`http://data.gov.uk/`), US's approach of the same (`http://www.data.gov/`), and Norway's parliament opening of its

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 becomming open. It's incremental, 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 [17].

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].

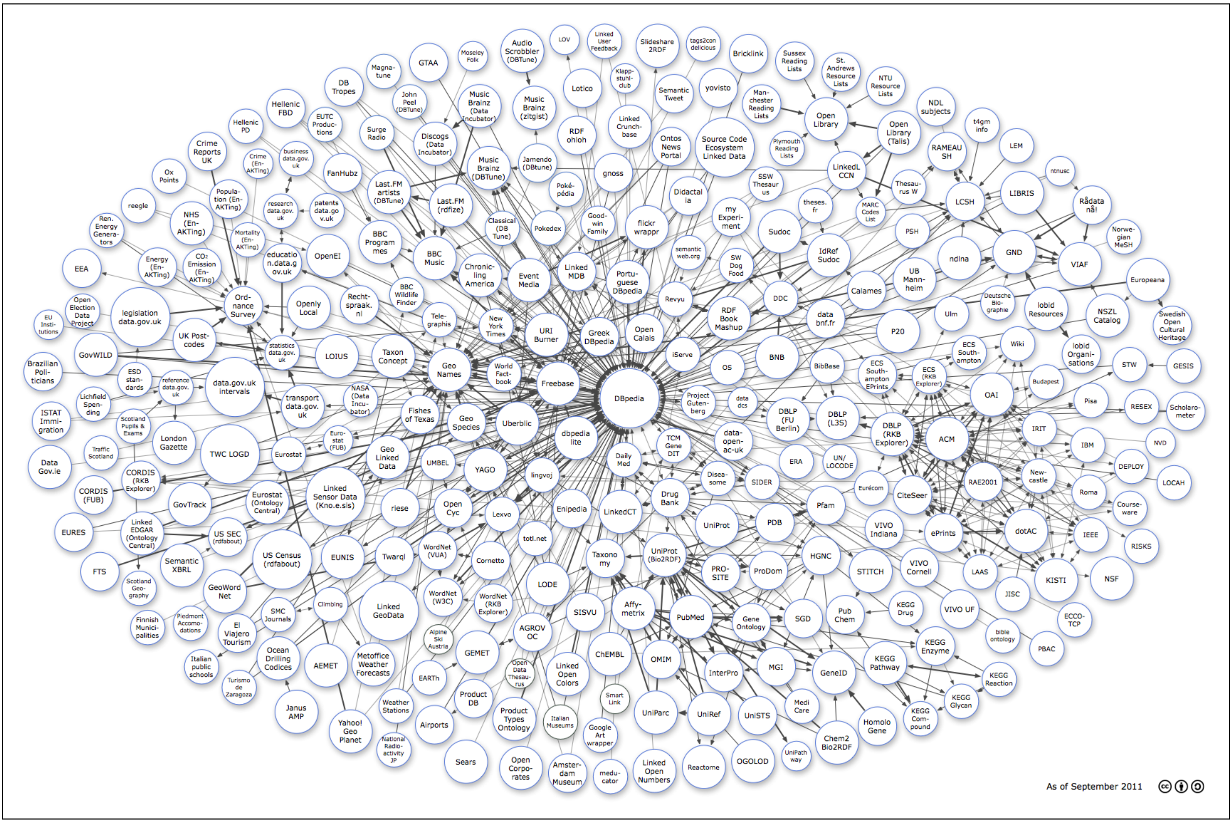

Figure 2.4: Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch. `http://lod-cloud.net/`

2.1.4.2 URL VS URI VS IRI

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

URLs and URIs are the most used terms, as the former denotes resources on WWW, while the

databases through Stortingets datatjeneste (`http://data.stortinget.no/`). You also have other non-governmental organizations, such as Parliamentary Monitoring Organizations (PMOs).

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

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.

Figure 2.5 explaines the parts of URI. The explanation is equally valid for URL and IRI.

URI have the form $scheme : [//authority]path[?query][\#fragment]$, where the parts in brackets are optional.

- **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 adresses. Paths can be organized hierarchically using / as separator.
- **query:** The query can be recognized with the preceding ?, and are typically used for providing parameters.
- **fragement:** Fragments provide an additional level of identifying resources, and are recognized by the preceding #.

Figure 2.5: Shortened explanation on URI given by Hitzler et.al. [23, p. 23].

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 exists many tools that can deserialize it. 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 [8, 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 consideration 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.

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

```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>
```

Another reason for XML being chosen as the default serialization was that it was readily available at the time RDF was being standardized. Figure 2.6 shows a timeline of the development of XML and SW.

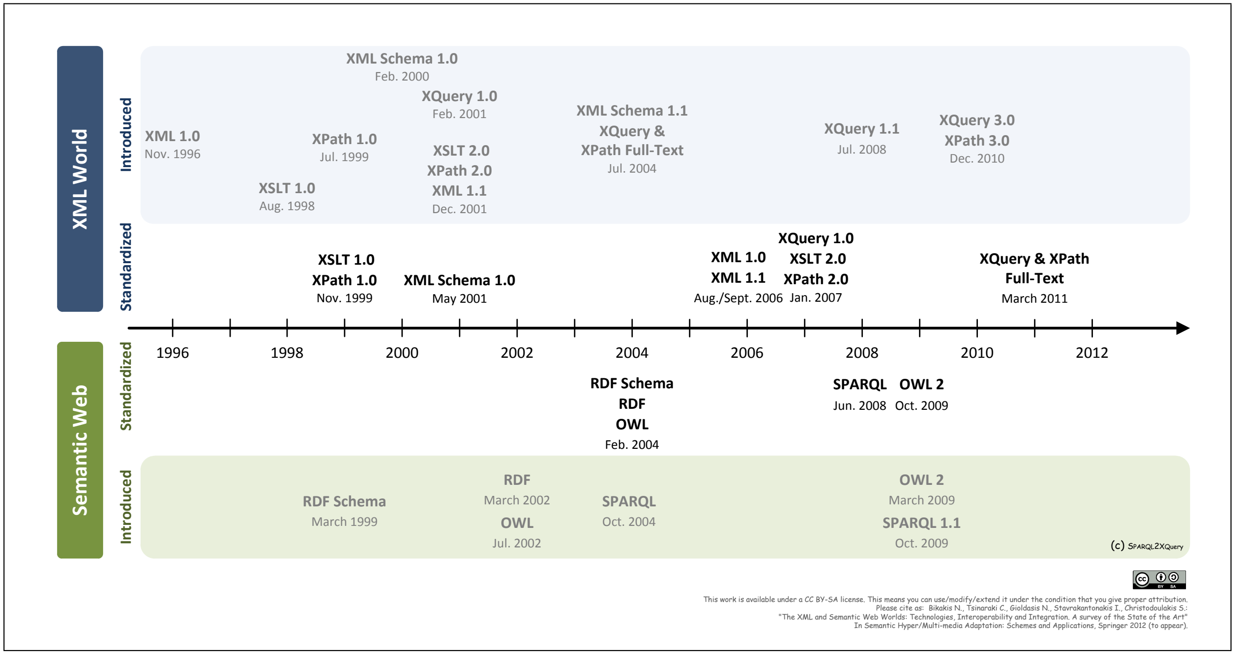

Figure 2.6: XML and Semantic Web W3C Standards Timeline.

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.

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 [15]. The latest version was submitted as a W3C Team Submission 28 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 this syntaxes in use by serialize 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's also syntactic sugar for writing collections. This is done by enveloping the resources as a comma-separated list in parantheses. Lastly, Turtle abbreviates common datatypes, e.g. the number minus five 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's a valuable educational tool because of it's simplicity and readibility.

2.1.5.3 Notation3 (N3)

N3 is meant as a compact and readable alternative to RDF/XML [18]. It dates back to 1998 [23, p. 25], and currently holds status as a Team Submission at W3C, last updated 28 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) [15, sec. 9].

2.1.5.4 N-Triples

N-Triples was designed to be a fixed subset of N3 [5, sec. 3]. It's also a subset of Turtle, in that Turtle adds syntax to N-Triples [15, 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's 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.

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

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

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"
        }
    }
}
```

- lang, optional: the language of the literal
- datatype, optional: the datatype of the literal

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

JSON-LD is the newest serialization to be included by W3C. It became a working draft 12 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 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 [12, 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.

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 @id-property. The absence of @id-property though creates a blank node, as shown in listing 2.8.

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 this is to allow organizations that al-

[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/

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

```
{
    "@context": {
        "ex": "http://example.org/",
        "foaf": "http://xmlns.com/foaf/0.1/"
    },
    "@id": "ex:Arne",
    "foaf:knows": "ex:Kjetil",
    "foaf:familyName": "Hassel"
}
```

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

```
{
    "@context": {
        "ex": "http://example.org/",
        "foaf": "http://xmlns.com/foaf/0.1/"
    },
    "@id": "ex:Arne",
    "foaf:knows": {
        "foaf:nick": [ "Bjarne", "Buddy" ]
    }
}
```

ready have deployed large JSON-based infrastructure to add meaning to their JSON documents that is not disruptive to their day-to-day operations [12]. In practice this will work by having two JSON documents, one being the original JSON document, which isn't 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.

2.1.5.7 Resource Description Framework in Attributes (RDFa)

RDFa is another serialization that recently got promoted in the W3C-system. As of June 7th 2012 it's 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 in

[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/

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"
    }
}
```

a way that doesn't leave any doubt.

The reasoning is that by making use of independently created vocabularies, the quality of meta-data 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, and may be overridden by setting vocab with another IRI.
- 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 Entailment

[TBW]

2.1.6.1 Entailment Regimes

[TBW]

2.1.7 Querying

[TBW]

2.1.7.1 Simple Protocol and RDF Query Language (SPARQL)

[TBW]

2.1.7.2 SPARQL Inferencing Notation (SPIN)

[TBW]

2.2 Javascript (JS)

JS begins its life in 1995, then named Mocha, birthed by Brendan Eich at Netscape [1, 11]. 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 webpages, but was long rediculed and misunderstood by developers [2].

Douglas Crockford states in his article “JavaScript: The World’s Most Misunderstood Programming Language” ten reasons for the confusion centering JS, given in figure 2.7. Luckily there has been some changes to the list since its conception in 2001.

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

Figure 2.7: Douglas Crockford's ten reasons for why JS is misunderstood

Point 1-5 is quite valid yet[14], but can be remedied by good and educational resources for learning JS[15]

Point 6 is (mostly[16]) 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 codify the de facto standard amongst the browsers has proved increasingly successful, groups such as W3Cs HTML Working Group 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 2.2.5).

Point 7 depends on your view of good books, and although there's much left to desire, there are some good books out there[17]. But more importantly, there are several efforts to deliver qualitative resources to educate developers in JS. These resources are increasingly - perhaps fittingly - web-based. There's also an increase of interest on conferences that target developers[18].

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

[15]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`).

[16]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 becomming a thing of the past.

[17]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)

[18]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.

Point 8 is left to be discussed (the author haven't 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 becomming part of the professional world, adoptation into conferences being one of the arguments suggesting this trend. You also have examples of major companies supporting and/or developing JS-libraries[19]. This would suggest that point 9 is not the case anymore[20].

Point 10 is still valid, as it can be hard 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 still, 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.

In this thesis we will differentiate between the implementation of JS and the specification ECMA-262 5.1 Edition (ECMA5). ECMA5 will be specified when there's functionality that haven't been implemented in major browser yet.

2.2.1 Object-Oriented

JS is fundamentally Object-Oriented (OO) as objects are its fundamental datatype [21, p. 115]. It treats objects different than many other programming languages though, as it doesn't have classes and class-oriented inheritance. There are fundamentally two ways of building up object systems, namely by inheritance (explained in section 2.2.1.1 and by aggregation (explained in section 2.2.1.2) [2].

Another design feature is its support of the functional programming style, by treating objects as first-class functions. 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 doesn't 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 2.11.

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.

[19]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.

[20]That being said, percentage-wise it's 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.

Listing 2.11: Use of literals in JS

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

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 2.12 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 2.12: Usage of prototype in JS

```
1 var objectA = {};
2 console.log(objectA); // logs Object
3 console.log(objectA.__proto__); // logs Object
4 Object.prop = 42;
5 console.log(objectA.prop); // logs 42
6
7 var objectB = new Object(),
8     objectC = Object.create(objectB);
9 console.log(objectC.prop); // logs 42
```

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 2.12 as a tree, and have done so in figure 2.8.

So when we call objectB.prop, JS will see if objectB has that property. As it hasn't it will continue to its prototype, which is objectA. As there is no objectA.prop, it will look for the prototype, which contains a link to Object. Now, as Object has the property prop, JS will return its value, which is 42 in our case.

A last note is that Object also have the property __proto__. This can also be manipulated, but JS will take care so that we don't run into an infinite loop when looking for properties that doesn't exist.

2.2.1.2 Dynamic Properties

All mutable objects in JS can be manipulated at run-time. This we also see in listing 2.12, as we add the property "prop" in line 4. Objects are basicly as containers for key-value entities, where the key is a string. In this regard, objects in JS can be regarded as maps, or dictionaries.

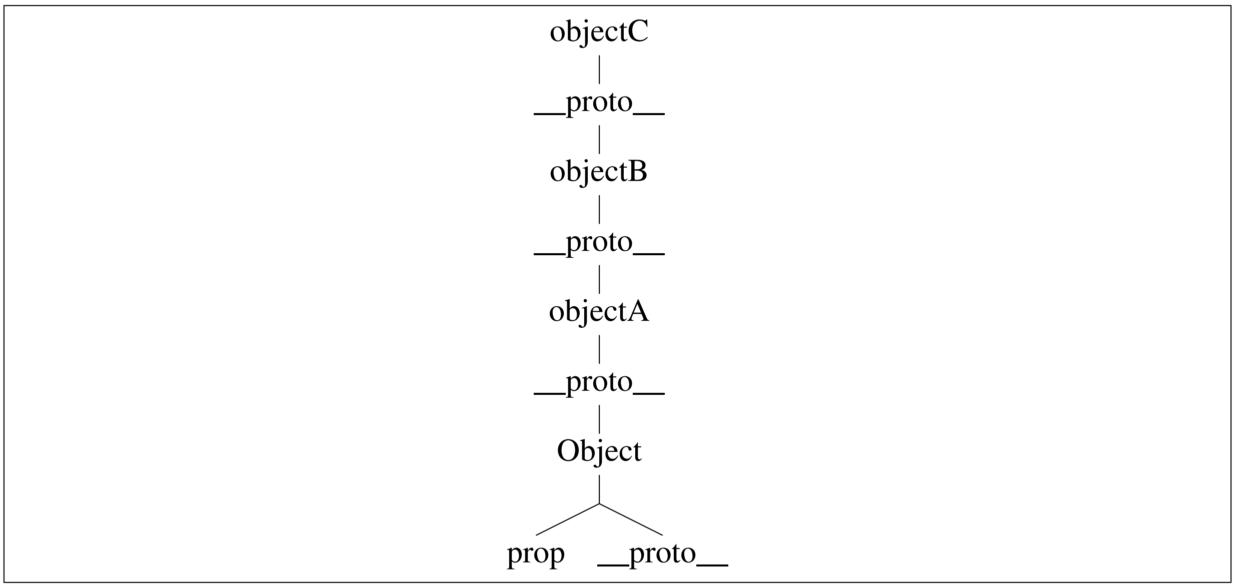

Figure 2.8: Object inheritance created in listing 2.12 visualized as a tree.

In addition, we can at any time manipulate existing properties by replace its values or delete the key alltogether. 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. ECMA5 offers three new functions that change this feature, namely Object.seal, Object.freeze, and Object.preventExtensions (with the responding Object.isSealed, Object.isFrozen, and Object.isExtensible to test whether or not these are set) [11, p. 114-115]. Explaining how these functions are outside the scope of this thesis, but sufficient 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've loaded a set of resource. This asynchronous feature is explained in depth in section 2.2.3.

Functions can be instantiated in many ways, as shown in listing 2.13. A function consists of three elements [21, 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 2.13: Instanciating 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 2.13 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 referrable by their names, if not they are set to a variable, in which case it will be referrable by the variable (line 10 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 2.14. It introduces the use of "this", which will be explained in section 2.2.2.

Listing 2.14: A simple object in JS

```
function ObjectA (x) {
    this.x = x;
    this.methodA = function (y) {
        return this.x + y;
    };
}

var A = new ObjectA(1300);
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 doesn't 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 [21].

If a variable isn't 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's run in, but in most browsers it's 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's declared, which is the global object if nothing else is specified. This is examplified in listing 2.15.

Listing 2.15: 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 2.16. 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 2.16: 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's also a headache for many aspiring JS-developers, as it may be a bit difficult to wrap your head around (and use correctly). Let's look another example of what may go wrong, given in listing 2.17. 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 doesn't have a property named val, it will return "undefined".

Listing 2.17: An example of code gone wrong because of faulty handling of closure

```
var functionA = function (val, func) {
        this.val = val;
        function functionAA () {
            return this.val;
        }
        return func(functionAA);
    },
    function B = function (func) {
        return func();
    };

console.log(functionA(42, functionB)); //logs undefined
```

2.2.3 Asynchronous Loading of Resources

[TBW]

2.2.3.1 Same origin policy

[TBW]

2.2.3.2 XMLHttpRequest Level 2 (XHR2)

[TBW]

2.2.3.3 XDomainRequest (XDR)

[TBW]

2.2.4 CommonJS

[TBW]

2.2.4.1 Asynchronous Module Definition (AMD)

[TBW]

2.2.4.2 Promise Pattern

[TBW]

2.2.5 Serverside implementations

[TBW]

2.2.6 Feature Detection

[TBW]

2.3 Design Patterns

[TBW]

2.3.1 Adapter

[TBW]

2.3.2 Bridge

[TBW]

2.3.3 Builder

[TBW]

2.3.4 Composite

[TBW]

2.3.5 Decorator

[TBW]

2.3.6 Facade

[TBW]

2.3.7 Factory Method

[TBW]

2.3.8 Flyweight

[TBW]

2.3.9 Proxy

[TBW]

2.3.10 Strategy

[TBW]

2.4 Test-driven development (TDD)

[TBW]

Chapter 3

Problem Description and Requirements

[TBW]

3.1 What are the components required for the framework?

[TBW]

3.2 Which Design Patterns are applicable for the components?

[TBW]

3.2.1 How does Javascript support these?

[TBW]

3.3 Does Javascript offer any interesting features for the framework?

[TBW]

Chapter 4

Tools

[TBW]

4.1 JS RDF store (RDFStore)

[TBW]

4.2 rdfQuery

[TBW]

4.3 Buster.JS

[TBW]

4.3.1 Node.JS

[TBW]

4.4 RequireJS

[TBW]

4.5 when.js

[TBW]

4.6 Github

[TBW]

4.7 WebStorm

[TBW]

4.8 Vocabularies

[TBW]

4.8.1 Friend of a Friend (FOAF)

[TBW]

Chapter 5

The Framework

This chapter will list all the modules that have been implemented. They are listed alphabetically, based on their names. The names shoould 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 below)
2. Location: The adress 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 3rd-party derived modules)
7. Test result: If tests are written for the module, its results are listed here

All modules are divided into one of three branches; Graphite, RDFQuery, or RDFStore. The two latter are originally taken from 3rd-party libraries (explained in Tools, Chapter 4), and modules residing in these will note in which degree they have been modified.

After the initial block detailing the attributes, a lengthier description explains the module in whole. Considerations taken will be noted, and variations/possibilities are explained. For a greater look into the signature of the modules, refer to documentation in appendix B.

The source can be forked at `https://github.com/megoth/graphitejs`, or read in appendix A (page 53).

5.1 API

Branch	Graphite
Location	`graphite/api.js`
Dependencies	Dictionary (page 33) , Graph (page 36) , Query (page 41) , Utils (page 46) , When (page 46)
Size:	4.4 kB
SLOC	121
Design Pattern	Facade (page 26)
Test result	10 tests, 11 assertions; Total average: 1968, Average/assertion: 179

The API-module tries to combine the most powerful modules of GraphiteJS 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 it's 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 is 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.

5.2 CURIE

Branch	RDFQuery
Location	`rdfquery/curie.js`
Dependencies	URI (page 46) , Utils (page 46)
Size:	10.5 kB
SLOC	84
Design Pattern	N/A
Test result	13 tests, 13 assertions; Total average: 61, Average/assertion: 5

[TBW]

5.3 Datatype

Branch	RDFQuery
Location	`rdfquery/datatype.js`
Dependencies	URI (page 46)
Size:	14.5 kB
SLOC	260
Design Pattern	Flyweight
Test result	12 tests, 22 assertions; Total average: 56, Average/assertion: 3

[TBW]

5.4 Dictionary

Branch	Graphite
Location	`graphite/dictionary.js`
Dependencies	URI (page 46) . Utils (page 46)
Size:	16.8 kB
SLOC	372
Design Pattern	Composite (page 25) , Flyweight (page 26) , Factory Method (page 26)
Test result	7 tests, 17 assertions; Total average: 34, Average/assertion: 2

The Dictionary-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 Dictionary, 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 Dictionary are:

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

- Symbol (e.g. a IRI)

Part of Dictionary is based on the Factory Method-pattern. Depending on the input, they take different courses of action to return the correct representation. These methods are:

- createObject
- createPredicate
- createStatement
- createSubject
- createTerm

The createTerm is the most arbitrary in that it can take any form of value, and will produce a representation (as long as it's meaningful in terms of possible terms in RDF, e.g. a Function wouldn't be a valid value). But in most cases it's more efficient to hint at what possible terms the value can be (e.g. a predicate, which can only be a Symbol), so it's recommended to use the other functions.

5.5 Engine

Branch	RDFStore
Location	`rdfstore/query-engine/query_engine.js`
Dependencies	Abstract Query Tree (page 35) , Tree Utils (page 45) , Store Utils (page 39) , Query Plan (page 36) , Query Filters (page 35) , RDF JS Interface (page 36) , RDF Loader (page 43) , Callbacks (page 35) , Utils (page 46)
Size:	71.7 kB
SLOC	1543
Design Pattern	N/A
Test result	53 tests, 312 assertions; Total average: 718, Average/assertion: 2

[TBW]

5.5.1 Abstract Query Tree

Branch	RDFStore
Location	`rdfstore/query-engine/abstract_query_tree.js`
Dependencies	Query Parser (page 42) , Tree Utils (page 45) , Utils (page 46)
Size:	27.1 kB
SLOC	540
Design Pattern	N/A
Test result	18 tests, 90 assertions; Total average: 130, Average/assertion: 1

[TBW]

5.5.2 Callbacks

Branch	RDFStore
Location	`rdfstore/query-engine/callbacks.js`
Dependencies	Abstract Query Tree (page 35) , Store Utils (page 39) , RDF JS Interface (page 36) , Tree Utils (page 45)
Size:	18.8 kB
SLOC	437
Design Pattern	N/A
Test result	7 tests, 25 assertions; Total average: 102, Average/assertion: 4

[TBW]

5.5.3 Query Filters

Branch	RDFStore
Location	`rdfstore/query-engine/query_filters.js`
Dependencies	Tree Utils (page 45) , Utils (page 46)
Size:	79.3 kB
SLOC	1435
Design Pattern	N/A
Test result	15 tests, 29 assertions; Total average: 186, Average/assertion: 6

[TBW]

5.5.4 Query Plan

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

[TBW]

5.5.5 RDF JS Interface

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

[TBW]

5.6 Graph

Branch	Graphite
Location	`graphite/graph.js`
Dependencies	**??** (page **??**) , Dictionary (page 33) , Engine (page 34) , Lexicon (page 38) , Utils (page 46) , When (page 46)
Size:	11.7 kB
SLOC	261
Design Pattern	N/A
Test result	4 tests, 8 assertions; Total average: 1019, Average/assertion: 127

The Graph module is the cornerstone of GraphiteJS. It is the abstraction of quadstores, and serves as an accesspoint for all the data processed by the framework. It's signature is somewhat small, but what it powers is the execution of SPARQL-queries. By calling it's 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, we decided to only support a subset of the SPARQL Query Language and SPARQL Update. The subset are:

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

This means we can only add 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 3rd-party code.

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 2.2.4.2).

The algorithm makes use of a property called deferred, and it's used in two places, shown in listing 5.1 and 5.2.

This setup enables the developer to call execute multiple times, while ensuring that each callback-function will not be triggered until the previous call is complete, and the query has been processed. It also ensures that the graph being processed is the latest version.

Listing 5.1: Use of deferred in the initialization of Graph

```
function () {
    this.deferred = When.defer();
    Engine.start(function (graph) {
        this.deferred.resolve(graph);
    })
}
```

Listing 5.2: Use of deferred in Graph's method execute

```
function execute (query, callback) {
    var deferred = When.defer();
    this.deferred.then(function (graph) {
        graph.engine.execute(query, function (graph) {
            deferred.resolve(graph);
            callback(graph);
        });
    });
    this.deferred = deferred;
}
```

5.6.1 Backend

Branch	RDFStore
Location	`rdfstore/persistence/quad_backend.js`
Dependencies	Store (page 39) , Tree Utils (page 45)
Size:	4.0 kB
SLOC	89
Design Pattern	N/A
Test result	2 tests, 57 assertions; Total average: 13, Average/assertion: 0

[TBW]

5.6.2 Lexicon

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

[TBW]

5.6.3 Store

Branch	RDFStore
Location	`rdfstore/persistence/quad_index.js`
Dependencies	B-Tree (page 45) , Tree Utils (page 45)
Size:	4.2 kB
SLOC	101
Design Pattern	N/A
Test result	1 tests, 7 assertions; Total average: 10, Average/assertion: 1

[TBW]

5.6.4 Store Utils

Branch	RDFStore
Location	`rdfstore/persistence/quad_index_common.js`
Dependencies	None
Size:	2.4 kB
SLOC	49
Design Pattern	N/A
Test result	3 tests, 31 assertions; Total average: 16, Average/assertion: 1

[TBW]

5.7 Graphite

Branch	Graphite
Location	`graphite.js`
Dependencies	API (page 32)
Size:	0.5 kB
SLOC	7
Design Pattern	Bridge (page 25)
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. In other words, you're essentially making the module use the Decorator (page 26) pattern. 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's scissored to their use.

5.8 Loader

Branch	Graphite
Location	`graphite/loader.js`
Dependencies	Proxy (page 40) , XHR (page 41) , Utils (page 46)
Size:	1.3 kB
SLOC	26
Design Pattern	Strategy (page 26)
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 ConcreteStrategy.

The module decides which strategy to load by Feature Detection (section 2.2.6).

5.8.1 Proxy

Branch	Graphite
Location	`graphite/loader/proxy.js`
Dependencies	XHR (page 41) , Utils (page 46)
Size:	1.2 kB
SLOC	36
Design Pattern	Proxy (page 26) , Strategy (page 26)
Test result	2 tests, 4 assertions; Total average: 26, Average/assertion: 7

The LoaderProxy-module is a participant in the Strategy Pattern as ConcreteStrategy. It was created to bypass the Same Origin Policy (section 2.2.3.1) by using a proxy-server on the same domain. It uses the LoaderXHR 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 LoaderProxy sends. Basicly it split the IRI to be loaded into separate parts (as explained in figure 2.5). As part of the framework, this service has been created as an application driven by Node.js.

5.8.2 XHR

Branch	Graphite
Location	`graphite/loader/xhr.js`
Dependencies	Utils (page 46)
Size:	1.5 kB
SLOC	40
Design Pattern	Strategy (page 26)
Test result	4 tests, 10 assertions; Total average: 76, Average/assertion: 8

The LoaderXHR-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.

5.9 Query

Branch	Graphite
Location	`graphite/query.js`
Dependencies	Loader (page 40) , Query Parser (page 42) , Utils (page 46) , When (page 46)
Size:	12.1 kB
SLOC	292
Design Pattern	Builder (page 25)
Test result	51 tests, 52 assertions; Total average: 512, Average/assertion: 10

[TBW]

5.10 Query Parser

Branch	Graphite
Location	`graphite/queryparser.js`
Dependencies	Sparql (page 42)
Size:	0.4 kB
SLOC	14
Design Pattern	Strategy (page 26)
Test result	2 tests, 4 assertions; Total average: 10, Average/assertion: 2

[TBW]

5.10.1 Sparql

Branch	Graphite
Location	`graphite/queryparser/sparql.js`
Dependencies	Sparql Full (page 42)
Size:	35.7 kB
SLOC	864
Design Pattern	Flyweight (page 26) , Strategy (page 26)
Test result	36 tests, 56 assertions; Total average: 124, Average/assertion: 2

[TBW]

5.10.1.1 Sparql Full

Branch	RDFStore
Location	`rdfstore/sparql-parser/sparql_parser.js`
Dependencies	None
Size:	1000 kB
SLOC	19114
Design Pattern	Builder (page 25) , Strategy (page 26)
Test result	N/A

[TBW]

5.11 RDF

Branch	RDFQuery
Location	`rdfquery/rdf.js`
Dependencies	CURIE (page 32) , Datatype (page 33) , URI (page 46) , Utils (page 46)
Size:	116.4 kB
SLOC	1473
Design Pattern	N/A
Test result	108 tests, 375 assertions; Total average: 931, Average/assertion: 2

[TBW]

5.12 RDF Loader

Branch	RDFStore
Location	`rdfstore/communication/rdf_loader.js`
Dependencies	Loader (page 40) , RDF Parser (page 43) , Tree Utils (page 45) , Utils (page 46)
Size:	4.0 kB
SLOC	50
Design Pattern	N/A
Test result	N/A

[TBW]

5.13 RDF Parser

Branch	Graphite
Location	`graphite/parser.js`
Dependencies	JSON-LD (page 44) , RDF JSON (page 44) , RDF/XML (page 44) , Turtle (page 45) , Utils (page 46)
Size:	1.4 kB
SLOC	34
Design Pattern	Strategy (page 26)
Test result	5 tests, 12 assertions; Total average: 168, Average/assertion: 14

[TBW]

5.13.1 JSON-LD

Branch	Graphite
Location	`graphite/parser/jsonld.js`
Dependencies	Dictionary (page 33) , Loader (page 40) , Utils (page 46) , When (page 46)
Size:	17.5 kB
SLOC	361
Design Pattern	Strategy (page 26)
Test result	22 tests, 63 assertions; Total average: 249, Average/assertion: 4

[TBW]

5.13.2 RDF JSON

Branch	Graphite
Location	`graphite/parser/rdfjson.js`
Dependencies	Dictionary (page 33) , Utils (page 46)
Size:	1.5 kB
SLOC	39
Design Pattern	Strategy (page 26)
Test result	8 tests, 9 assertions; Total average: 923, Average/assertion: 103

[TBW]

5.13.3 RDF/XML

Branch	RDFQuery
Location	`rdfquery/parser/rdfxml.js`
Dependencies	Dictionary (page 33) , RDF (page 43) , URI (page 46) , Utils (page 46)
Size:	14.7 kB
SLOC	264
Design Pattern	Strategy (page 26)
Test result	29 tests, 136 assertions; Total average: 1785, Average/assertion: 13

15 deferred tests; 36/169 of good tests are faulty; 56/56 of bad tests are faulty.

[TBW]

5.13.4 Turtle

Branch	RDFQuery
Location	`rdfquery/parser/turtle.js`
Dependencies	Dictionary (page 33) , RDF (page 43) , URI (page 46)
Size:	16.9 kB
SLOC	474
Design Pattern	Strategy (page 26)
Test result	4 tests, 31 assertions; Total average: 315, Average/assertion: 10

[TBW]

5.14 Tree Utils

Branch	RDFStore
Location	`rdfstore/trees/utils.js`
Dependencies	None
Size:	13.9 kB
SLOC	380
Design Pattern	Flyweight (page 26)
Test result	2 tests, 4 assertions; Total average: 13, Average/assertion: 3

[TBW]

5.15 B-Tree

Branch	RDFStore
Location	`rdfstore/trees/in_memory_b_tree.js`
Dependencies	None
Size:	27.8 kB
SLOC	547
Design Pattern	Flyweight (page 26)
Test result	4 tests, 152 assertions; Total average: 55, Average/assertion: 0

[TBW]

5.16 URI

Branch	RDFQuery
Location	`rdfquery/uri.js`
Dependencies	Utils (page 46)
Size:	9.5 kB
SLOC	179
Design Pattern	N/A
Test result	74 tests, 99 assertions; Total average: 230, Average/assertion: 2

[TBW]

5.17 Utils

Branch	Graphite
Location	`graphite/utils.js`
Dependencies	Lexicon (page 38)
Size:	26.8 kB
SLOC	498
Design Pattern	Flyweight (page 26)
Test result	37 tests, 141 assertions; Total average: 130, Average/assertion: 1

[TBW]

5.18 When

Branch	Graphite
Location	`graphite/when.js`
Dependencies	None
Size:	25.1 kB
SLOC	279
Design Pattern	Adapter (page 25)
Test result	N/A

[TBW]

Chapter 6

Discussion

[TBW]

Chapter 7

Conclusion

[TBW]

7.1 Further Work

[TBW]

Bibliography

[1]

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

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

[4] *OWL Web Ontology Language Overview*, 2004. [Online; accessed 13-July-2012].

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

[6] Rdf vocabulary description language 1.0: Rdf schema. `http://www.w3.org/TR/rdf-schema/`, February 2004. [Online; accessed 11-July-2012].

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

[8] Resource description framework (rdf): Concepts and abstract syntax. `http://www.w3.org/TR/rdf-concepts/`, February 2004. [Online; accessed 11-July-2012].

[9] *OWL 2 Web Ontology Language Document Overview*, 2009. [Online; accessed 14-July-2012].

[10] *OWL 2 Web Ontology Language Profiles*, 2009. [Online, accessed 14-July-2012].

[11] *ECMAScript Language Specification*, 2011.

[12] *JSON-LD Syntax 1.0*, 2012. [Online, accessed 15-July-2012].

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

[14] Semantic web - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Semantic_Web`, July 2012. [Online; accessed 10-July-2012].

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

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

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

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

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

[20] 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].

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

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

[23] P Hitzler, M Krotsch, and S Rudolph. *Foundations of Semantic Web*. 2010.

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

Appendix A

The codebase for GraphiteJS

Appendix B

The documentation of GraphiteJS