FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Web Response Ontology for Data Import in Smart Buildings

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Reinhold Gschweicher, Bsc
Registration Number 0828055

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Dipl.-Ing. Dr.techn. Filip Petrushevski

Vienna, 26th November, 2017

Reinhold Gschweicher	Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Reinhold Gschweicher, Bsc
2084 Obermixnitz 23

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

Wien, 26. November 2017

Reinhold Gschweicher

Acknowledgements

Foremost, I would like to express my gratitude to my advisor, Wolfgang Kastner, who gave me the opportunity to write my master's thesis as a member of the Automation Systems Group. Likewise, I would like to thank Filip Petrushevski for his steady support. Both of them were always available for questions and discussion of relevant issues.

Moreover, my thanks go to my friends who always supported and helped me.

Especially, I would like to thank my parents, Sylvia and Franz for their mental, but also financial support during my time as a student and throughout my life.

Abstract

Smart Buildings collect and store data from lots of sensors to save energy and improve the comfort level of the occupants among other things. The usefulness of the collected samples can be enhanced by fusing them with external data sources. These improved data points could be analysed to help save energy and therefore money. However before working with the data, it must first be retrieved from web servers. These provide their information in various formats like CSV, JSON and XML. Each format must be treated and parsed differently.

Ontologies are a great tool for combining data points and representing knowledge. In this thesis, a response description ontology to describe web service responses in CSV, JSON or XML format is created. To verify the applicability of the ontology it is used by a parser to extract the specified data points in web service responses. The extracted data is then used to calculate one day heat energy consumption forecasts in a smart office building situated in Vienna.

Contents

Abstract **vii**

Contents **ix**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem statement . . . 2
1.3 Methodology . . . 3
1.4 Structure of the work . . . 3

2 State of the art **5**
2.1 Information exchange in the World Wide Web . . . 5
2.1.1 Information encoding . . . 5
2.1.1.1 CSV . . . 5
2.1.1.2 XML . . . 6
2.1.1.3 JSON . . . 7
2.1.2 Message exchange protocols . . . 7
2.1.2.1 Uniform Resource Locator . . . 8
2.1.2.2 HTTP . . . 8
2.1.2.3 REST . . . 8
2.2 Ontology . . . 9
2.2.1 Resource Description Framework (RDF) . . . 10
2.2.1.1 RDF Serialization Formats . . . 11
2.2.2 RDF Schema (RDFS) . . . 14
2.2.3 Web Ontology Language (OWL) . . . 15
2.2.4 SPARQL Protocol And RDF Query Language (SPARQL) . . . 16
2.2.5 Related Work . . . 17
2.3 Conclusion . . . 19

3 System Specification and Design **21**
3.1 Use cases . . . 22
3.2 Functional and non-functional requirements . . . 22
3.3 Class Design . . . 23

3.3.1 Important Terms . . . 23
3.3.2 Classes . . . 24
3.3.3 Properties . . . 27
3.3.3.1 Response . . . 27
3.3.3.2 Structure Object . . . 27
3.4 System Design . . . 29

4 Implementation **31**
4.1 Infrastructure . . . 31
4.1.1 SPARQL Endpoint . . . 32
4.1.2 Building Management System . . . 32
4.1.3 Building Data Interface . . . 33
4.1.3.1 JEDataCollector . . . 33
4.1.3.2 Improvement: SQL Driver . . . 34
4.2 Ontology Instances . . . 35
4.2.1 Geolocation Service . . . 35
4.2.2 Weather Service . . . 36
4.2.2.1 Weather Underground Geolookup . . . 37
4.2.2.2 Weather Underground Historic Data . . . 39
4.2.3 Building Information Service . . . 39
4.2.3.1 JEVis Structure . . . 40
4.2.3.2 JEVis Data . . . 42
4.3 SPARQL Queries . . . 43
4.3.1 Root of path . . . 43
4.3.2 Getting the full path . . . 44
4.4 PyCaster . . . 46
4.4.1 Data Import . . . 47
4.4.1.1 Sparql-py . . . 47
4.4.1.2 jeapi-py . . . 47
4.4.2 Data Processing . . . 47

5 Evaluation **49**
5.1 Energy Data . . . 49
5.2 Anomalies . . . 50

6 Conclusion **53**

A Ontology **55**
A.1 response description . . . 55

B SQL driver **65**
B.1 Configuration . . . 66
B.2 Message Sequence chart . . . 67

List of Figures **69**

List of Tables **71**

Bibliography **73**

CHAPTER 1

Introduction

1.1 Motivation

Keeping the temperature of a workplace or a residential building at a comfortable level is a complicated and resource intensive task. According to Statistics Austria [Aus], in 2013, one third of the useful energy consumption in Austria can be attributed to space and water heating. Figure 1.1 shows the useful energy consumption in Austria. The main energy consumption falls into one of three major categories: production purposes, traction and finally heating, lighting and computing combined. In absolute values, space and water heating accounted for 332 PJ, traction for 380 PJ, production purposes for 374 PJ, and lighting and computing for only 33 PJ.

Seeing these numbers it is easy to understand why much effort has been put into minimizing the heating costs in residential as well as commercial buildings.

Just reducing the cost of heating would be easy (not heating at all). The hard part is to minimize the costs while keeping the comfort level of the people in the building high. To reach this goal, building automation is an essential part [Kas06]. The roots of building automation lie in the heating domain. Among others, core domains of building automation are Heating, Ventilation and Air Conditioning (HVAC), lighting and shading. Today, building automation in commercial buildings is well established.

As a side effect of building automation extensive databases with sensor data are often available. Furthermore, services providing data like geolocation or weather forecasts are readily available over the Internet. However, the import, fusion and analysis of these different data inputs is a difficult and building or service specific task. As a result, a high amount of duplicated effort for parsing and importing data samples is invested.

In recent years small, versatile and connected devices generally called Internet of Things (IoT) devices have been growing in popularity. The rapid development of new IoT

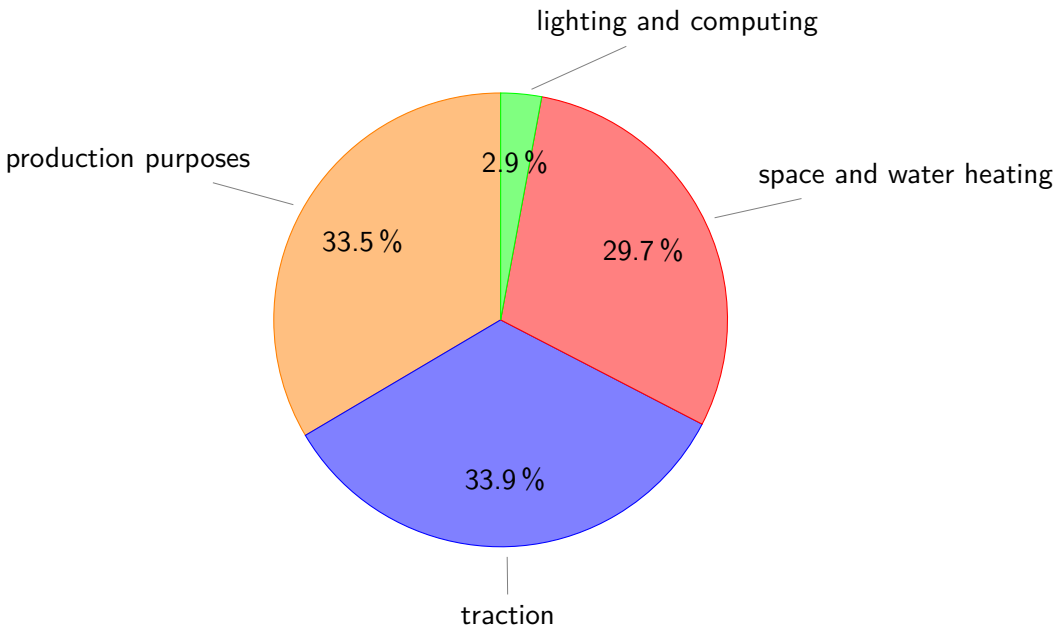


Figure 1.1: Useful energy analysis in Austria [Aus]

devices has also increased the number of possible data sources, because many of those devices are equipped with sensors. With this rise the complexity of manually selecting data points and writing specialized importers increases.

Sometimes, the effort of analysing the available data samples, selecting the best samples and writing code to import those samples is too high. Therefore, much useful data is sitting in databases unused with waiting potential.

1.2 Problem statement

The most efficient control mechanism would be one, that knows the future and can plan accordingly. Control loops can only react to changes in their sensor values after the events are measurable. Prediction and forecasts try to close this gap by using statistics and computer algorithms predict the most likely outcomes. Facility managers can use these forecasts to improve the performance of an automated system. They can be used to monitor the performance of the heating system, find anomalies and improve the control parameters of the heating system.

However, the improvement can only be significant if the forecasts are accurate. Bad forecasts can lead to bad decisions, which may result in increased costs.

An energy load forecasting solution needs to be tailored to every building and service in use by hand. Each service can have a different structure or encoding (like CSV, JSON or XML). To change a data source (for example, a weather service providing weather

forecasts), the program computing the energy load forecast needs to be rewritten or updated with a new parser adapted to a different structure and possibly a different encoding.

This thesis proposes the use of an ontology to describe the response of these various data sources. To use data points from a different provider, only minimal changes to the program and an ontological description of the data source should be necessary.

1.3 Methodology

First, research of the state of the art and related work of short term energy load forecasts in smart buildings is done. Then, the current state of ontologies for weather forecasts and sensors in smart buildings is evaluated. Next, the requirements and non-requirements of a forecasting system and the ontologies are defined. According to the found requirements and the chosen weather service, a prototype is implemented. Existing ontologies are used or extended as needed. Finally, the results are examined and possible future improvements are discussed.

1.4 Structure of the work

This section provides a general overview of the structure of this thesis.

Chapter 2 discusses the state of the art regarding information encoding, information exchange and ontologies.

Chapter 3 contains a requirements analysis to define the scope of the ontology. The requirements are then used to establish the design of a prototype system and the ontology.

In Chapter 4, models based on the defined ontology describing the responses of different services are shown. A prototype application using these models is presented. The prototype has the ability to obtain weather and sensor data from various sources and then use this data to calculate a one day forecast. The application uses information defined in the ontological models to import the correct data points from the various service responses.

Chapter 5 evaluates the imported data and calculated results. The root mean square error of the forecast is used to find anomalies in the data set. Those anomalies are further inspected and possible causes identified.

Finally, Chapter 6 concludes this thesis, summarizes the findings and gives an outlook on possible future work.

CHAPTER 2

State of the art

2.1 Information exchange in the World Wide Web

There are many ways to transfer information in the World Wide Web. In this section, some commonly used Web standards for information exchange and encoding are described.

2.1.1 Information encoding

Before sending information over the Internet, the information should be encoded and structured. This facilitates the implementation of computer programs transferring information. In the following sections, the Web standards CSV, XML and JSON are shown. These Web standards create structured data which can be parsed by computers.

2.1.1.1 CSV

The first described standard is abbreviated as CSV and stands for Comma Separated Values. It describes a simple text only data format for tabular data [Sha05]. Each row represents one record. Each record may be separated into multiple fields/columns by a delimiter. The first row can be used as header to describe the contents of the single columns. The header should have the same number of fields as normal record lines.

As the name suggests, the records are separated by a comma. However as there is no single "master" specification for this format [Sha05] other delimiters are used as well. Common delimiters are comma (,), semicolon (;), white spaces and tabulators. The fields may contain numbers or text. The text can be quoted, but like the separator this is implementation specific.

2.1.1.2 XML

The eXtensible Markup Language (XML) is a text based data format to exchange structured information over the Internet. It was developed by the XML Working Group of the World Wide Web consortium in 1996 and like HTML is a subset of the Standard Generalized Markup Language (SGML) [BPSM+98]. Additional restrictions on XML include case sensitive names and that the characters '<' and '&' are not allowed in data fields and must be entered as '<' and '&'. A complete list of differences and restrictions can be found at [Cla97].

For the intended use as document exchange format [BPSM+98], XML is designed for ease of implementation [BPSM+98]. Programs to read and process, or create XML-structured documents should be easy to create.

A well formed XML document has a prolog and a root element. Other elements are nested within the root element. The prolog is used for the XML declaration, which defines the version of XML being used, and other definitions describing the XML document.

Elements are encapsulated by so called start- and end-tags. An element with no content can be encoded by an empty-element tag instead of a start- and end-tag. Both variants are shown in the following listing:

```
<building> <floor/> </building>
```

In this example, <building> is the start-tag, </building> the end-tag, and <floor/> is the empty-element tag.

Attributes can be used to associate name-value pairs with elements. The attributes only appear inside of start-tags or empty-element tags.

The following example extends the previous example with two attributes: the city the building is situated in and the number of floors in the building.

```
<building city="Vienna">
  <floor number="2"/>
</building>
```

One strength of XML is the ability to add meta-data to tags in the form of attributes [Str]. Another strength of XML is the support for mixed content. The keyword "#PCDATA" can be used to indicate binary data as content. "#PCDATA" historically stands for "parsed character data".

2.1.1.3 JSON

JavaScript Object Notation (JSON) is intended to be an easy data-interchange format [jso]. It is a language independent text format, but uses a convention common to C-like languages.

JSON uses four primitive types to represent information. Those types are *string*, *number*, *boolean* and *null*. Furthermore, two structured types (*objects* and *arrays*) can be used [Bra14].

An object in JSON is described in [Bra14] as "an unordered collection of zero or more name/value pairs". The name is a string and the value can be a string, number, boolean, null, array or another object. After each name, a colon (:) symbol is used as separator. An object is enclosed by curly brackets containing the name/value pairs. Such pairs are separated by a single comma (,).

The second structured type is the array. An array is represented by squared brackets ([and]). It contains an ordered sequence of zero or more values. The values are separated by commas and can be of any primitive data type or structured type. The elements in one array may also be of different types.

JSON, unlike XML, has a less verbose message object encoding overhead [AP12] which results in smaller encoded messages. When binary data is to be encoded, XML can be more efficient because of the ability of attributes to contain binary data. JSON needs to encode this binary data for example with Base64, which introduces more overhead.

As being based on a subset of JavaScript [jso], JSON can be easily parsed in JavaScript. Other languages have already adopted JSON support with many tools facilitating the serialization of data structures as JSON-text.

The building with two floors example can be represented in JSON in the following way:

```
{
  "building" : {
    "city" : "Vienna",
    "floors" : 2
  }
}
```

2.1.2 Message exchange protocols

This section describes some Web standards used to exchange various types of information over the Internet.

2.1.2.1 Uniform Resource Locator

Before describing the protocols, the terms Uniform Resource Locator (URL), Universal Resource Identifier (URI) and International Resource Identifier (IRI) are explained. This explanation is based on the work of DuCharme [DuC13].

Uniform Resource Locator (URL) and Universal Resource Identifier (URI) are two terms often used interchangeably. But URL and URI have a slightly different meaning. A URL may also be called a Web address. It is a compact way for a client to specify which resource on the Web (a specific Web page) it wants. The URL contains the information about the name of the resource, the location on the server, and the server itself. The popularity and simplicity of Web addresses led developers to use URLs for resources, which were not web pages at all. This confused many people. To solve this confusion, engineers defined the specification for Universal Resource Names (URN). A URN starts with `urn:` and uses the character ':' as separator. The term URI was introduced as broader term for URL and URN. As the URN standard was not not widely adopted, the terms URL and URI are often used as synonyms.

IRIs are a generalization of URLs, because they allow non-ASCII characters to be used in the IRI character string [SR14].

2.1.2.2 HTTP

The Hyper Text Transfer Protocol (HTTP) is described in [FR14] as "a stateless protocol for distributed, collaborative, hypertext information systems." To indicate the target resource and relationships between resources the Unified Resource Identifier standard (URI) is used.

The protocol defines method tokens to indicate the operation that is to be performed on the specified resource. The common methods are OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT. Each method requests some other action or processing of the request from the server. It indicates the purpose of the client's request [FR14]. For example, GET is used to indicate a request for a transfer of the specified resource. POST indicates, that the body of the request needs to be processed by the server. PUT is used to replace the resource specified by the URI by the request payload.

HTTP allows MIME-like [FB96] messages, which provide meta information about the content of the message.

Most HTTP connections consist of a GET request to request information for a resource represented by a URI [FR14].

2.1.2.3 REST

The Representational State Transfer (REST) paradigm is a software architectural style of the World Wide Web [FT00]. REST defines a Web-friendly or Web-like way of designing

and implementing Web services [AP12]. Uniform Resource Identifiers (URIs) are used to uniquely identify a resource. There is only a small set of methods to interact with the resources. Those methods are based on HTTP methods like GET, POST and PUT [Sch14]. REST itself is no protocol, but rather a description of an architecture with its constraints and rules. A system conforming to the guidelines and constraints of REST can be called RESTful.

2.2 Ontology

This section explores what an ontology is and how it can be represented.

According to Steyskal [Ste14] one of the most cited definition of the term ontology is the following:

"An ontology is an explicit specification of a conceptualization." [G+93]

Stated differently, an ontology is a way to represent and organize knowledge about a domain of interest [Sch14].

The advantage over relational databases is the possibility to additionally store semantics of data and the rules describing the schema. Using an ontology to describe data, adds the possibility to create a knowledge base. Using this knowledge base, reasoners can be used to infer new information [Ste14],[Sch14].

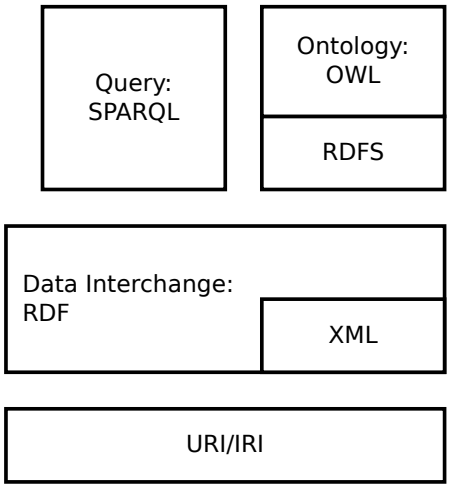


Figure 2.1: Used subset of the Semantic Web Stack [Bra]

Figure 2.1 shows an overview over the relevant parts of the Semantic Web stack [Bra]. The stack is built upon the URI/IRI layer. Using URIs or IRIs, the higher layers can uniquely identify resources. As explained earlier in Section 2.1.2.1, these technologies are also used in the World Wide Web. Just like a Web-page can be uniquely identified and accessed by its URL, a resource can also be identified by its URI.

The next layer defines the data interchange using the RDF (Resource Description Framework), which is based on XML (eXtensible Markup Language). RDF is both human readable and at the same time machine-processable. A more in-depth description of RDF and its serializations can be found in Section 2.2.1.

RDFS (RDF Schema) and OWL (Web Ontology Language) are extensions to RDF. They enable the possibility to define and model ontologies. Both languages extend RDF with the features that facilitate describing complex relations between resources [Ste14].RDFS and OWL are described in Sections 2.2.2 and 2.2.3, respectively.

Naturally, developers want to access the modeled data and ontologies. The standardized query language for querying RDF and OWL data is SPARQL [PAG09] (SPARQL Protocol And RDF Query Language). SPARQL is described in Section 2.2.4.

2.2.1 Resource Description Framework (RDF)

As the name implies, RDF (Resource Description Framework) is a framework to express information about resources. Resources can be anything including documents, people, physical objects and abstract concepts [SR14].

RDF Version 1.0 was released as W3C recommendation in 2004 [CK04]. An updated version RDF 1.1 was released in 2014 [CWL14].

The RDF standard was developed to structure information and represent knowledge on the World Wide Web [Kof14]. Most information on the Web displayed on websites is intended to be viewed by people. RDF is designed to structure information to be human readable as well as easily processed by applications [SR14, Ste14]. This also facilitates information exchange between applications without loss of semantics.

In RDF, the basic unit of information is a triple consisting of a subject, a predicate and an object. The subject can be viewed as resource identifier, the predicate as a property name, and the object as a property value [DuC13]. The predicate puts the two resources, subject and object, into a relationship. The relationship is phrased in a directional way (from subject to object) and called a *property* [SR14].

In graphs, the basic building blocks are nodes and edges [Kof14]. Triples can be visualized as a connected graph. Figure 2.2 shows a graphical representation of the simple triple "Alice knows Bob". The subject (Alice) is connected to the object (Bob) by the predicate (knows). The nodes of the graph represent the subjects and objects. The nodes are connected by directed edges, which represent the predicates [SR14]. Usually, one triple doesn't describe a resource entirely. For example, Alice and Bob can also have an age or other people they know. Each triple describes a resource in more detail. Those triples are combined into a directed graph called *RDF Graph* [Ste14].

To query the resulting graph for information the language SPARQL can be used. SPARQL is explained in more depth in Section 2.2.4.

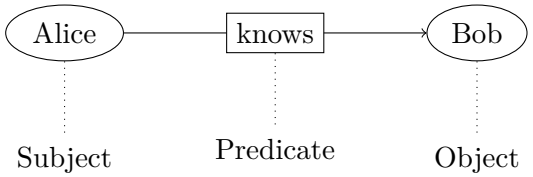


Figure 2.2: RDF triple showing sentence "Alice knows Bob"

In RDF 1.0 resources are identified by URIs. With RDF 1.1, resources are uniquely identified by IRIs [Woo14]. As described in Section 2.1.2.1, IRIs (International Resource Identifiers) are a generalization of URLs (Universal Resource Identifiers). URLs are most notably known to identify Web addresses. IRIs are more general because they allow non-ASCII characters to be used in the IRI character string [SR14].

RDF can be used to publish and interlink data on the Web [SR14]. Because the resources are uniquely identified using IRIs the referred nodes don't necessarily need to be within the same document [Kof14]. They may be located elsewhere on the Web. Therefore, RDF provides a simple way to describe distributed data [AH11].

Literals in RDF are basic values that are not IRIs [SR14]. They are used for values such as strings, numbers and dates [CWL14]. To enable the automatic parsing of literals, they are associated with a datatype. Examples for data types are *xsd:string* for simple strings, *xsd:integer* for arbitrary-sized integer numbers, *xsd:decimal* for arbitrary-precision decimal numbers, or *xsd:date* for dates (yyyy-mm-dd) [CWL14]. In the case of strings, the literal value may even be localized using well formed language tags [PD09] and a datatype of *xsd:langString*.

2.2.1.1 RDF Serialization Formats

The mentioned standard defines the concepts of RDF, but not the details of how the RDF triples and graphs are serialized. This section lists available serialization formats defining a concrete syntax on how to represent, store or exchange RDF triples.

Together with the RDF 1.0 specification, the serialization RDF/XML was introduced in 1999. It was historically the first RDF serialization format recommended by the W3C. Being based on XML, this format is easily processed by XML parsers. A disadvantage of RDF/XML is its verbosity, which often obfuscates the underlying RDF graph and hence makes it hard to understand for a human reader [Kof14].

Apart from RDF/XML there are several other serialization formats in use (see Figure 2.3 for an overview). Over time, these other formats have been adopted and standardized as W3C recommendations [SR14]. According to [SR14], the interesting languages are the following:

- Turtle family (N-Triples, Turtle, TriG and N-Quads)

- JSON-LD
- RDFa
- RDF/XML

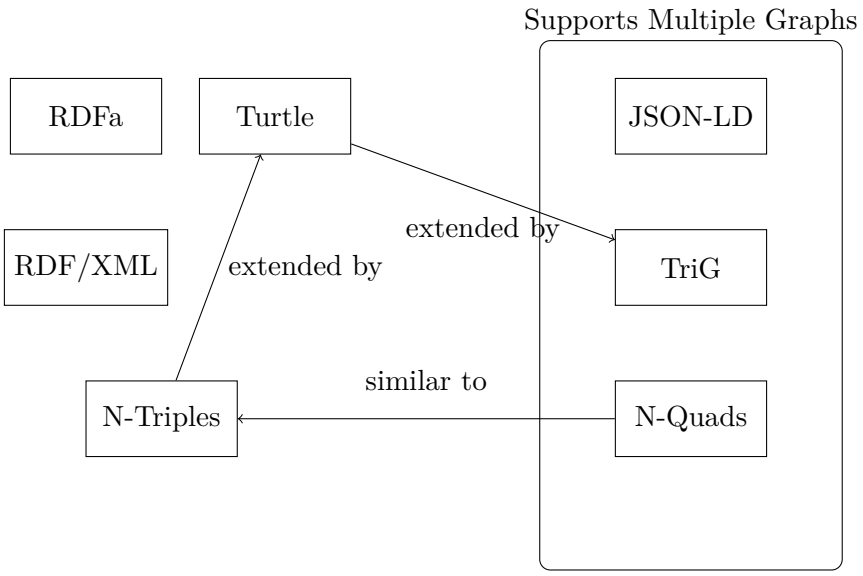


Figure 2.3: RDF 1.1 serialization formats [SR14]

Since the Turtle language is widely supported and easy to read for humans, further RDF examples in this thesis are using the Turtle language. The remaining section describes the Turtle syntax and the provided features.

Published as a W3C recommendation in February 2014, Turtle [BBLPC14] is an extension to N-Triples. It provides syntactic shortcuts for namespace prefixes, lists and shorthands for datatyped literals [SR14]. Namespace prefixes can be used to make the serialization more compact and easier to read. Instead of the full IRI, a short prefixed link can be used to identify a resource.

```
<http://example.com/thesis/alice> <http://xmlns.com/foaf/0.1/knows> <http
    ://example.com/thesis/bob> .
```

The previous example can be rewritten using prefix definitions as follows:

```
# prefix definitions
@prefix thesis: <thesis http://example.com/thesis/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
# triples
thesis:alice foaf:knows thesis:bob .
```

With Turtle 1.1, the base and the prefix can also be described in the SPARQL way. The next example is equivalent to the previous one, but uses the SPARQL notation.

```
# prefix definitions
PREFIX thesis: <thesis http://example.com/thesis/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
# triples
thesis:alice foaf:knows thesis:bob .
```

Apart from prefix definitions, the examples show several different syntactic rules:

- IRIs are enclosed in between '<' and '>' characters.
- A prefix must be defined before it is used.
- A triple is terminated with the dot-character '.'.
- Comments are indicated with the character '#'. The comment spans from the '#' character to the end of the line.

The next example shows a slightly modified version of the relationship between Spiderman and the Green Goblin taken from [BBLPC14].

```
@base <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/relationship/> .

<green-goblin>
    rel:enemyOf <spiderman> ;
    a foaf:Person ;    # in the context of the Marvel universe
    foaf:name "Green Goblin" .

<spiderman>
    rel:enemyOf <green-goblin> ;
    a foaf:Person ;
    foaf:name "Spiderman", "Die Spinne"@de .
```

The token 'a' is another shortcut which represents a 'rdf:type' relation. It mimics the English language. The first occurrence in the example can be read as "green-goblin is a foaf:Person".

Another feature shown in the example above is the use of predicate and object lists. The following triples are equivalent (omitting the prefix definitions):

```
<spiderman>
    rel:enemyOf <green-goblin> ;
    a foaf:Person .
```

```
<spiderman> rel:enemyOf <green-goblin> .
<spiderman> a foaf:Person .
```

Using the *predicate list*, one can omit the repeating subject of the triples. Instead of terminating the first triple with an '.' the character ';' is used to indicate that the next predicate-object pair describes the previous subject.

The following example shows the usage of *object lists* and *language tags*.

```
<spiderman> foaf:name "Spiderman", "Die Spinne"@de .
```

With an *object list* the objects are repeated with the same subject and predicate. The series of objects are separated using the character ",". The second feature is the use of a *language tag*. The language tag is preceded by a '@' and can be used to localize strings.

Literals are used to identify values like strings, numbers or dates. Turtle provides a shorthand syntax for several data types. Table 2.1 shows the abbreviated and lexical forms supported by the Turtle language.

Data Type	Abbreviated	Lexical
xsd:string	"spiderman"	"spiderman"^^xsd:string
xsd:integer	-5	"-5"^^xsd:integer
xsd:decimal	-5.0	"-5.0"^^xsd:decimal
xsd:double	-4.2E9	"-4.2E9"^^xsd:double
xsd:boolean	true	"true"^^xsd:boolean

Table 2.1: Quoted Literal shorthands [BBLPC14]

More expressive languages are needed to describe ontologies. The W3C standardized languages that fulfill this role are RDFS and OWL. They are based on RDF and extend it with the main language features necessary to create ontologies (formal and expressive conceptualizations) [Kof14]. The extensions RDFS and OWL are explained in the following sections.

2.2.2 RDF Schema (RDFS)

The RDF Schema (RDFS) is an extension to the RDF specification. RDF Schema 1.0 has been a W3C Recommendation since February 2004 [BGM04]. The latest version (RDF Schema 1.1) was released in February 2014 [BG14].

RDFS adds capabilities to express the meaning of resources in an RDF graph. To add this semantic information to RDF properties, RDFS introduces features to define classes and hierarchies. Like other RDF resources, additional meta information is represented as RDF triples, which means no additional representation format is needed [Kof14].

RDFS introduces the concept of classes (*rdfs:Class*), which can have properties (*rdf:Propterty*) to further describe the classes [Kof14]. For these classes and properties,

hierarchical relations can be described [Kof14, DuC13]. To define a hierarchy of classes the property *rdfs:subClassOf* may be used. As mentioned, the creation of hierarchies is not limited to classes. Properties can be put into a hierarchical relation as well by using the *rdfs:subPropertyOf* property.

At the root of the class hierarchy is the class *rdfs:Resource*. All resources described by RDF are instances of the class *rdfs:Resource* or its subclasses [BG14]. One subclass of *rdfs:Resource* is the class of literal values *rdfs:Literal* for values such as strings and integers.

All classes are instances of *rdfs:Class*. Even *rdfs:Class* is an instance of *rdfs:Class* [BG14].

These added capabilities enable the definition of simple ontologies called taxonomies [KKR12, Kof14, Ste14].

2.2.3 Web Ontology Language (OWL)

The OWL 2 Web Ontology Language (OWL 2) was released in December 2012 as a W3C recommendation [C+12]. OWL 2 is an extension and revision of the OWL Web Ontology Language (OWL 1), which was published in February 2004 [MVH+04]. OWL was developed to facilitate the creation and distribution of ontologies over the Web. The goal was to make Web content more accessible to machines [C+12]. As a description language OWL 2 represents the meaning of resources on the web in a way machines can understand [Kof14].

The OWL 2 specification [C+12] defines ontology as "formalized vocabularies of terms, often covering a specific domain and shared by a community of users". The terms of an ontology are defined by describing their relationships with other terms within the ontology.

To store and exchange the created ontologies a concrete syntax is needed. OWL 2 defines one required syntax and some optional ones. The one required syntax is RDF/XML. The optional syntaxes are OWL/XML, Functional Syntax, Manchester Syntax and Turtle. Every syntax provides different advantages like easier processing using XML tools (RDF/XML), easier discovery of formal structures (Functional Syntax), easier reading and writing of RDF triples (Turtle).

In OWL, the number of individuals connected through a property can be defined. These constraints are called cardinality constraints. The cardinality can be specified by defining a maximum, minimum or an exact amount [Kof14].

Another way to describe properties in more detail provided by OWL is the distinction between object and datatype properties. Object properties are used to link different individuals to each other, whereas data properties are used to link datatype properties to individuals.

In OWL 2, there are two different ways to define semantics: *Direct Semantics* and *RDF-based Semantics*. The Direct Semantics provide a structural specification

independent of any specific syntax. There exists a mapping from the Direct Semantics to RDF triples. RDF-Based Semantics assign meaning directly to RDF graphs to represent OWL 2 ontologies. The previously mentioned mapping can also be used to transform an RDF graph satisfying certain restrictions into an OWL 2 ontology. OWL 2's RDF-Based Semantics is fully compatible with RDF Semantics and extends them [Kof14].

The above mentioned features are just a small part of OWL 2. For a complete discussion about the specification, an interested reader may refer to [C+12].

2.2.4 SPARQL Protocol And RDF Query Language (SPARQL)

The previous languages encode the data in RDF triples, which can be stored in RDF stores. In order to be useful, this data needs to be accessed. To simplify the interaction with RDF graphs a standardized language is desirable. Such a language prevents a lot of duplicated work on specialized programs to parse and interpret these graphs.

SPARQL is a recursive acronym and stands for *SPARQL Protocol And RDF Query Language*. The "S" in "SPARQL" actually stands for "SPARQL" [AH11]. SPARQL 1.0 was introduced as W3C recommendation in January 2008 [PS+08] and updated with the new version 1.1 on March 2013 [HSP13].

RDF is a directed labeled graph data format, and SPARQL is the standard query language to access RDF data. Therefore, SPARQL is essentially a graph-matching query language [PAG09, Kof14].

To construct a simple SPARQL query, at least the two keywords *SELECT* and *WHERE* are needed. The *SELECT* statement defines which variables should be returned by a query. The *WHERE* clause contains the basic graph pattern to match against the RDF data [HSP13]. To represent graph patterns, SPARQL uses the Turtle [BBLPC14] RDF serialization syntax.

After the values have been matched and the results are listed in a table, the second part of the query contains solution modifiers which may be applied. Classical database query operators such as *DISTINCT*, *ORDER BY*, *LIMIT* and *OFFSET* are available [PAG06]. Basic graph patterns require the entire query pattern to match. But not all structures are complete (not everyone has a middle name). To add a graph pattern that adds to the solution if available, but does not reject a solution if unavailable the keyword *OPTIONAL* is used [HSP13].

Initially, SPARQL was only designed to query an RDF store. With version 1.1 the keyword *CONSTRUCT* was added, which enables the ability to update the RDF store.

The update to version 1.1 brought many new features like value assignment, value aggregation (like *SUM* and *COUNT*), path expressions (like '?','*' and '+') and nested queries [C+13]. The introduction of those new keywords and capabilities raised the expressiveness of SPARQL significantly [Kof14].

A few selected path expressions are listed in Table 2.2. The keyword *elt* indicates a path element, which may be composed of path constructs. These expressions allow to extend basic SPARQL graph patterns into *Property Paths*. A triple pattern is a simple property path of length exactly one [HSP13]. Using path expressions a path of arbitrary length connecting two resources can be described.

Syntax Form	**Property Path Expression Name**	**Matches**
iri	PredicatePath	Simple path of length one indicated by an IRI
^*elt*	Inverse Path	Inverse path (object to subject)
elt1 / *elt*2	SequencePath	A sequence path of *elt*1 followed by *elt*2
*elt*1 | *elt*2	AlternativePath	Matches when elt1 or *elt*2 or both alternative paths matches
*elt**	ZeroOrMorePath	A path of zero or more matches of *elt* between a subject and an object
elt+	OneOrMorePath	A path of one or more matches of *elt* between a subject and an object
elt?	ZeroOrOnePath	A path of zero or one matches of *elt* between a subject and an object

Table 2.2: Property paths introduces with SPARQL 1.1 [HSP13]

The concept of *nested queries* or *sub queries* allows one to embed SPARQL queries within other queries. This enables results which could not be achieved otherwise. It's worth noting that the nested queries are evaluated first and the solution of the inner query is expanded into the outer query, where further processing may occur.

A Web service which accepts SPARQL queries is called a *SPARQL endpoint* [DuC13].

2.2.5 Related Work

This section outlines some of the ontologies used to describe or store weather and sensor data.

As part of the project ThinkHome Steyskal [Ste14] created an ontology to store preferences of various users. ThinkHome is an ontology-based intelligent home using artificial intelligence to improve the control of home automation functions.

Kofler [Kof14] presents an OWL based ontology to describe the smart home environment. The ontology provides a knowledge base for autonomous control of user comfort and energy efficiency in smart homes.

Staroch [Sta13] constructs an OWL ontology describing current weather conditions and weather forecasts. It aims to provide knowledge for a smart home to make decisions based on current and future weather conditions.

Kastner et al. [KKR12] describe an ontology which describes the energy usage and energy creation in the field of home and building automation.

Huang et al. [HLZ15] create several ontologies to implement a dialog system between humans and their smart homes. The dialog system aims to provide the user with a more natural way to communicate with the smart home. To make this possible, the two ontologies *Home Ontology* and *Family Member Ontology* are provided. The former is used to define and store information on space, device location, and device status. The latter models and stores information about the family members in the home environment. A described scenario is the request "Turn on the fan.". The system responds with "Turn on the fan in the living room?", which the user answers in the affirmative. The scenario demonstrates the advantage of a dialog system by reacting to an incomplete command (the location of the fan to operate was not specified).

Gao et al. [GCF16] create the Point of Interest (POI) ontology model to store and query points of interest from social networks, such as Twitter, Foursquare, Google and others. Pictures, Tweets and other social network information is often linked to POIs. The problem they solve is the collecting and merging of replicated data. Instead of clustering or data ranking, the authors introduce the POI ontology as a unified structure. The use of SPARQL is outlined to insert and query the stored (and merged) data. With the introduction of this unified data source, query applications can retrieve data of a single POI without requesting it multiple times from different sources.

Compton et al. [CBB+12] introduce the Semantic Sensor Network (SSN) ontology, which is produced by the W3C Semantic Sensor Network Incubator group (SSN-XG). The ontology is an OWL2 ontology for describing the relationships between sensors, stimulus and observations. Data itself, units of measurement or locations are not part of the ontology. The latest version is available at `http://purl.oclc.org/NET/ssnx/ssn`.

Daniele et al. [DdHR15] present the Smart Appliances REFerence (SAREF) ontology. Their ontology aims to provide a common architecture to enable semantic interoperability for smart appliances. Those smart appliances form heterogeneous systems, which need standardized interfaces to communicate on a sensor and device level. Those standards exist, but for two devices to communicate a translation between standards needs to be defined. For a system with devices using, for example, 47 different standards, a total of $47 \cdot 46 = 2162$ translations is required. With SAREF only 47 translations between the standard and the reference ontology are necessary.
The ontology is centered around describing devices, their locations, functions, energy production and consumption, etc. The ontology can be found at `http://ontology.tno.nl/saref/`.

Lefrançois et al. [LKGZ16] creates the Smart Energy Aware Systems (SEAS) ontology, that describes energy systems and their interactions among other things. The ontology is modularized and versioned. It aims proviede nodes with a way to "expose, exchange, reason and query knowledge in a semantically interoperable manner". Due to its modularity it is able to describe systems and features of interest in many different fields. Examples

include devices, forecasting, properties (time, comfort, statistics), smart grid, smart home, offers and markets. The ontology can be found at `https://w3id.org/seas/`.

Ahvar et al. [AST+17] present the FUSE-IT ontology, which merges already existing ontologies, such as Semantic Sensor Network (SSN), Smart Appliances REFerence (SAREF), Smart Energy Aware Systems (SEAS), and others. The ontology aims to provide a common information base to implement smart building management systems to ensure "global physical and cyber security, trust and safety in critical sites". The ontology covers the four key domains energy supply and efficiency, facility and building automation, information and communication technology (ICT), as well as security and safety. The paper states that other ontologies don't cover all four domains, and consequently cannot provide a unified information base.

2.3 Conclusion

Several ontologies have been created covering many fields like weather, comfort, energy usage and sensor networks. Along with descriptive information many ontologies store the data points as well.

Unfortunately, the majority of data points is not provided within an ontology. Web services commonly offer their information in raw formats without context. To use these data sources, programs often need to be customized or specifically developed for each Web service. This leads to duplicated code and effort.

Therefore, in this thesis a new ontology, which can be used to describe different data structures, is created. Using this ontology, a program can be used for many different Web services, saving the effort of reimplementing nearly the same data parsing programs over and over again. The extracted data can be used by programs or inserted into another ontology.

CHAPTER 3

System Specification and Design

In this section the response description ontology and the prototype system are specified. The full ontology in turtle syntax is available in Appendix A.1.

In Ontology Development 101 [NM01], the following steps are proposed to create a new ontology:

1. Determine the domain and scope of the ontology
2. Consider reusing existing ontologies
3. Enumerate important terms in the ontology
4. Define the classes and the class hierarchy
5. Define the properties of classes - slots
6. Define the facets of the slots
7. Create instances

This thesis roughly follows these steps. The first two steps are covered by the requirements analysis (Section 3.2) and the state of the art research (Section 2.2.5). Section 3.3 covers step three to six by enumerating terms, defining classes (Section 3.3.2) and their properties (Section 3.3.3).

The last step of creating instances of the ontology classes is described in Section 4.2.

3.1 Use cases

UC1: The designer creates a new model with a data point.

- To create the model, the designer gets a response from the server.
- The response is analysed to find the required data point.
- A path describing the objects and conditions a parser must take to get from the root of the response to the data point is created.
- As last step the previously found path is used to create a model containing all needed information from the path.

UC2: The modeller adds a new data point to an existing model.

- To create the model, the modeller gets the response from the server.
- The response is analyzed to find the required data point.
- A path describing the objects and conditions a parser must take to get from the root of the response to the data point is created.
- The found path is integrated into the existing model without disturbing previously found paths.

UC3: The parser is used to extract a data point from a response.

- A response and an identifier of a data point are given.
- The parser searches the model for a path to the specified data point.
- The found path is used to parse the response to extract the specified data points from the response.

3.2 Functional and non-functional requirements

In this section each requirement is defined and then explained in more depth.

- R1: "Support for sparse descriptions of the response."

  If only one specific data point in a response is needed then it should be sufficient to only specify the path from the root to the desired data point. The specification of the whole response should not be mandatory. This requirement is derived from *UC1*.

- R2: "The addition of new data must not invalidate existing structures."

  If a new entry is to be added to an existing model then only the new elements of a path leading there need to be defined. This requirement is derived from *UC2*.

- R3: "The model should be able to describe XML, JSON and CSV responses."

  XML, JSON and CSV are three major established data formats. The ontology must be able to describe the structures of all three formats.

- R4: "Conditions on processing a node must be describable in the ontology."

  Conditionals are essential to describe key-value pairs often seen in data responses. The key attribute can be a specific string or any other attribute. A parser can drop processing a subtree if the condition is not met. The use cases *UC1* and *UC2* both describe a path, with elements only valid if a condition is met.

- R5: "Data attributes may have a different name (identifier) than in the response."

  The name of a temperature value in different responses may be something like 'C' or 'TemperatureC', but the name in the model may be a different one. The use case *UC3* specifies, that an identifier is used to specify the data point to parse for.

- R6: "It should be possible to substitute a name of an attribute."

  The instances of the ontology need to be generic. The structure of two responses may be identical, except for the identifiers of some attributes. It should be possible to use the instance and replace keywords to prevent multiple instances which are exactly the same except for one attribute name.

- R7: "The type and the format of a data point need to be describable in the ontology."

  It should be possible to define the data type (such as string, float, date) and the format (e.g. the date format) of data points. The parser may need this additional information to convert strings to other data formats.

- R8: "The ontology should store no data."

  The ontology is supposed to describe the structure of the response. No data samples are to be stored in the ontology itself.

3.3 Class Design

After defining the scope of the new ontology the classes are specified in this section.

3.3.1 Important Terms

The third step of [NM01] is to enumerate important terms. In this section, important terms are listed along with reasons for their importance.

- response

  The purpose of the ontology is to describe the structure of web responses. Therefore, this keyword is included.

- CSV, XML, JSON

  These three encoding formats are specified by requirement *R3*.

- tag, attribute

  These keywords are well known XML keywords.

- value, value type, value format

  Requirement *R7* states to describe the data points with their type and also format.

- condition

  Requirement *R4* describes, that data points can be excluded if specific conditions are not met.

3.3.2 Classes

The fourth step of Ontology Development 101 [NM01] is to define the classes for the ontology. This section follows this process and elaborates the constructed class hierarchy.

First the *Response* class is defined. To model the encoding of the response, the specialized classes *CSV Response*, *JSON Response* and *XML Response* are used. Figure 3.1 shows the resulting class hierarchy.

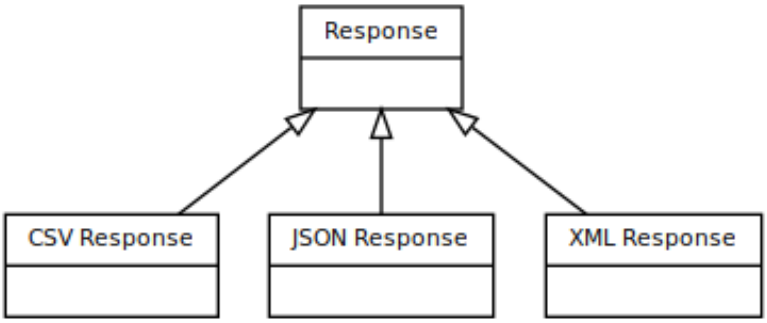


Figure 3.1: Classes defining the encoding of response

The first response structure type designed is CSV. As outlined in Section 2.1.1.1 this format describes a table of values. In this thesis, it is assumed, that the table has a header with names for each column. Figure 3.2 shows the only class needed to define the name of a desired column.

The next modeled encoding is XML. As described in Section 2.1.1.2, this format uses tags and attributes to describe its elements and values. Figure 3.3 shows the classes *XML*

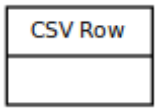


Figure 3.2: CSV structure classes

Attribute and *XML Tag*. A tag can contain other tags as well as attributes. Therefore, the superclass *XML Object* is introduced.

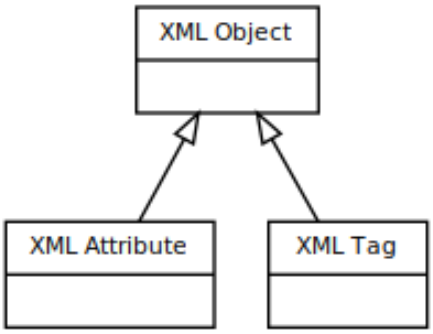


Figure 3.3: XML structure classes

The last encoding modeled in this thesis is JSON. JSON has four primitive types (*string*, *number*, *boolean* and *null*), as well as two structured types (*object* and *array*). Figure 3.4 shows the hierarchy if all those types are modeled.

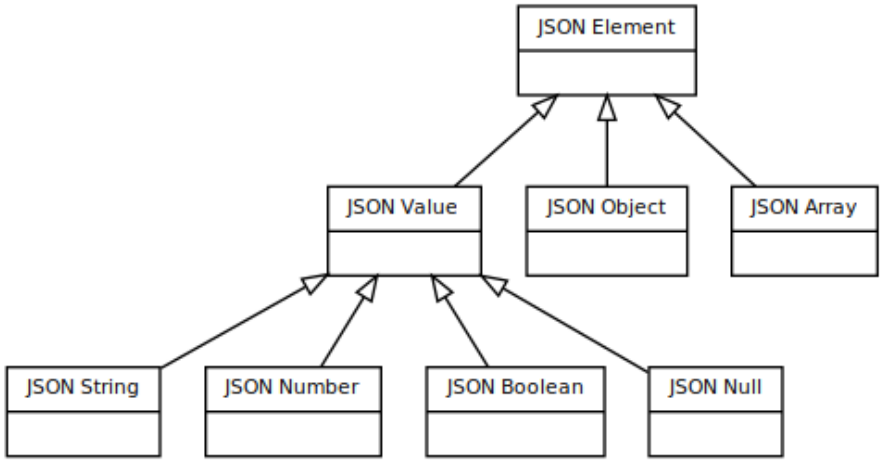


Figure 3.4: JSON structure classes, a first try

Specialized primitive data type classes are removed to simplify the class hierarchy. This information will be encoded as property (slot) of the class *JSON Value*. Furthermore,

the class *JSON Array* is removed. The parser is expected to match every member of a JSON array to the defined structure. With these two simplifications the simplified hierarchy is shown in Figure 3.5.

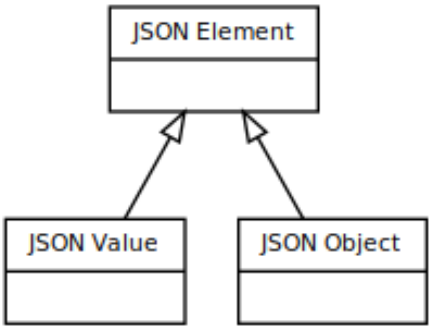


Figure 3.5: JSON structure classes, simplified

Notice, that the hierarchy of XML and JSON classes is very similar. They could be combined to shrink the number of ontology classes. One could argue, that the parser needs to know which encoding is used, and therefore different classes for different encodings are needed. However, the encoding is specified by the *Response* subclasses and won't change during the parsing of a web response.

For the final class hierarchy, the classes *JSON Element* and *XML Object* are combined to form the class *Structure Object*. *XML Tag* and *JSON Value* are combined into *Structure Tag*. And lastly *XML Attribute* and *JSON Value* form *Structure Attribute* to represent primitive types like strings, numbers, dates and times. The CSV class *CSV Row* can be represented by the class *Structure Attribute*, as both describe a named field containing values. The final class hierarchy can be seen in Figure 3.6.

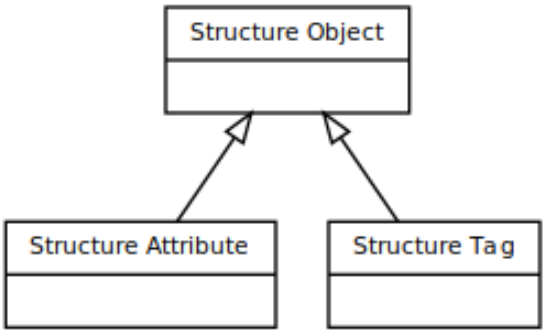


Figure 3.6: Structure classes, combination

3.3.3 Properties

Step five of [NM01] recommends to define the properties of the classes (slots). Section 3.3.3.1 describes the slots of the *Response* classes, whereas Section 3.3.3.2 covers *Structure Object*'s classes.

Additionally step six is covered by defining the facets of the slots. The facets, or cardinality, define how many attributes of the same kind an instance can have. For example, a binary tree node can have at most two children.

3.3.3.1 Response

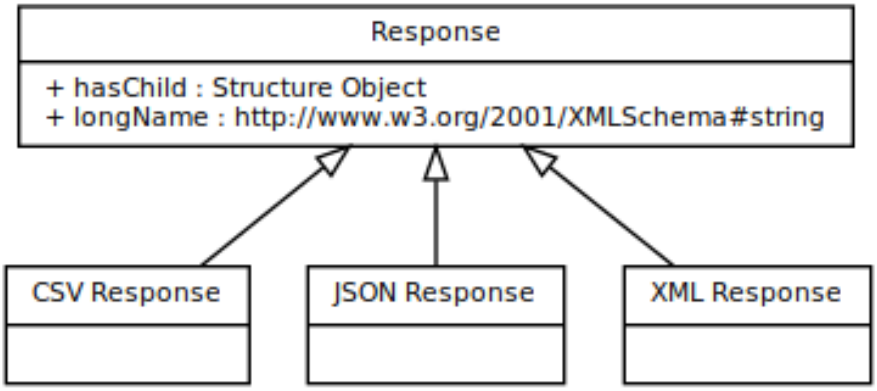


Figure 3.7: Ontology response classes

Currently, three different data encodings are implemented: CSV, JSON and XML, fulfilling requirement *R3*. As shown in Figure 3.7, the response classes are children of the class *Response*. Those classes are used to indicate the type of the encoding of the response. A response has one or more *hasChild* properties with links to *Structure Object*s describing the structure of the response. The property *longName* describes the response in a human readable form. It is for informational purposes and provides a way to describe the response in more detail. This property is optional.

3.3.3.2 Structure Object

To describe the structure of the response, instances of the *Structure Object* class and its children are used. Each object of the class *Structure Object* (or its children) may have any combination (or none) of the properties *tag*, *hasName* or *hasCondition*. In the current implementation, each property has a maximum cardinality of one.

- The property *tag* is a string identifying the current object in the actual response.
- The property *hasName* is a string identifying the object independently of the actual response. This property is a result of requirement *R5*.

- The property *hasCondition* is a link to a *Structure Attribute* with *tag* and *value*. For a *Structure Object* to be valid, either no condition is present, or the condition is satisfied. The condition is satisfied if the specified *Structure Attribute* is found. *Structure Object*s where the condition is not met should be discarded by the parsing program. This satisfies requirement *R4*.

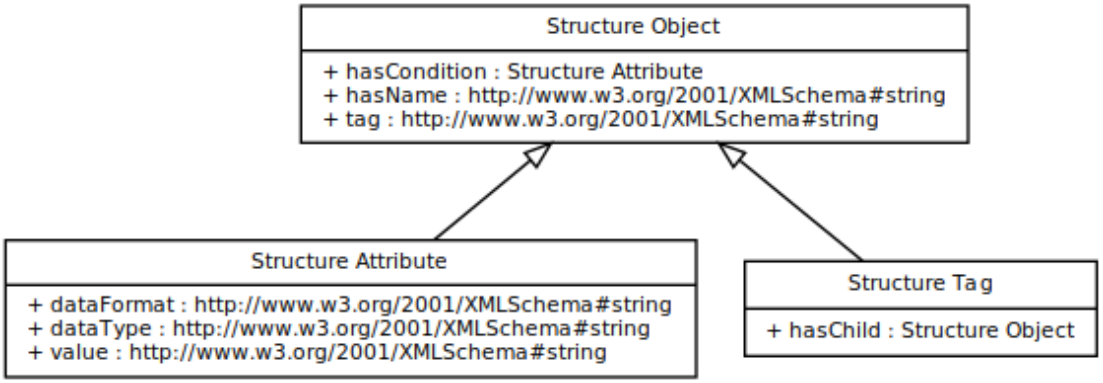


Figure 3.8: Ontology Structure Object classes

As shown in Figure 3.8 two subclasses of *Structure Object* are implemented.

- Structure Tag

  A *Structure Tag* is used to build the structure of the response. In addition to the inherited properties, it has the property *hasChild*. This property is a reference to another instance of *Structure Object*. A *Structure Tag* can have one or more children.

  The property *tag* is used to indicate keywords which contain other structures.

- Structure Attribute

  A *Structure Attribute* describes the object's semantics. To give meaning to the values, as required by *R7*, the properties *dataType* and *dataFormat* are used. Examples for values of *dataType* are "datetime" or "float". If none is given the default data type is "string", which tells the parser to do no data conversion. In the case of "datetime", the property *dataFormat* can be used to store a format string. This gives meaning to a string and supports the computer in interpreting a string as a date or time object.

To make the models more general, the properties *tag* and *value* can store special keywords. In this thesis, a property containing "!sub:building!" should be replaced by a variable specified by the keyword "building". This satisfies requirement *R6*.

Because of the nature of RDF graphs responses can be sparsely described (requirement *R1*), as well as new data points can be added without disturbing previously modelled aspects of the response (requirement *R2*).

The last requirement *R8*, is satisfied by not providing a way to store the measurements of data points in the response description ontology.

3.4 System Design

To validate the created ontology and show that it can be used to model and import data from the Internet, a prototype system was implemented. The system uses the newly defined ontology to model and import weather data, geolocation information and sensor data from different sources. These sources are required to provide their data in a CSV, JSON or XML encoded format.

In addition to importing data, the system needs to perform certain computations. As the title of the thesis suggest the goal of the computation will be short term prediction of the energy consumption of a smart building.

CHAPTER 4

Implementation

In this chapter, the developed prototype (Section 4.4), the infrastructure supporting the prototype (Section 4.1) and the created ontology instances for the data sources (Section 4.2) are described.

Among those sources are a geolocation service, a weather service and a building information service. These data sources are specific for the prototype application, but any service responding in XML, JSON or CSV could be modeled.

4.1 Infrastructure

This section describes the infrastructure around the prototype application.

Figure 4.1 shows an overview of this infrastructure. Building sensor data is gathered by the Building Management System (BMS) *Desigo*.

As the data is not directly available (see Section 4.1.2), it is imported via a Building Data interface called *JEVis*. Section 4.1.3 describes the structure of JEVis and the contributions to JEVis to import data from Desigo. To provide access to the created instances of the response-description ontology, the SPARQL Endpoint *Blazegraph* is used. *OpenStreetMaps* is used as geolocation service to find the location (city or latitude and longitude) based on the building's name. With this location, the weather service *Weather Underground* is contacted to find the nearest airport weather station and get its historic weather data.

The central part of the prototype is the application *PyCaster*. This application uses the provided infrastructure and instances of the response-description ontology (see Section 4.2) to import the necessary data to calculate daily forecasts of the heat energy consumption of a building.

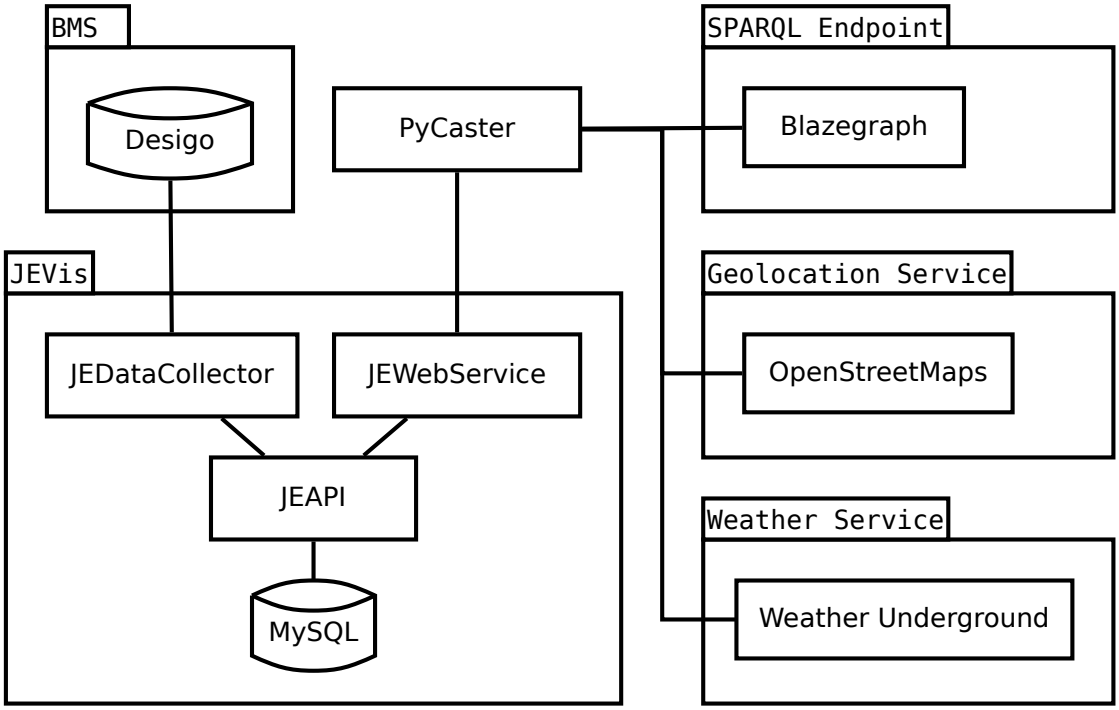


Figure 4.1: Overview of the system and infrastructure around it

4.1.1 SPARQL Endpoint

Rather than writing a program to traverse the RDF graphs of the ontology and find the requested information, a SPARQL endpoint can be used. As the name suggests queries for information need to be written in SPARQL. This implementation uses *Blazegraph* due to its user friendliness and ease of use.

The ontology which is accessed by Blazegraph is described in Section 4.2.

4.1.2 Building Management System

A Building Management System (BMS) interacts with the building's sensors and actuators, collects and archives the sensor data, and provides an interface for the facility manager to interact with the building system (to change room temperatures or heating strategies, for instance).

For the prototype system, a BMS called *Desigo* is accessed. Desigo is a BMS storing the most recent sensor data in a Microsoft SQL (MSSQL) database. Older data is archived after a configurable time-frame. As a security measure, the database is accessible only locally. Direct connections to the database over the network are ignored by the server.

This means that the building data is not available through a Web service and therefore can't be used directly by the prototype implementation.

4.1.3 Building Data Interface

As a central place to aggregate, store and access sensor data, the system JEVis3 is used. JEVis provides a RESTful JSON encoded API to access the stored structures and data points.

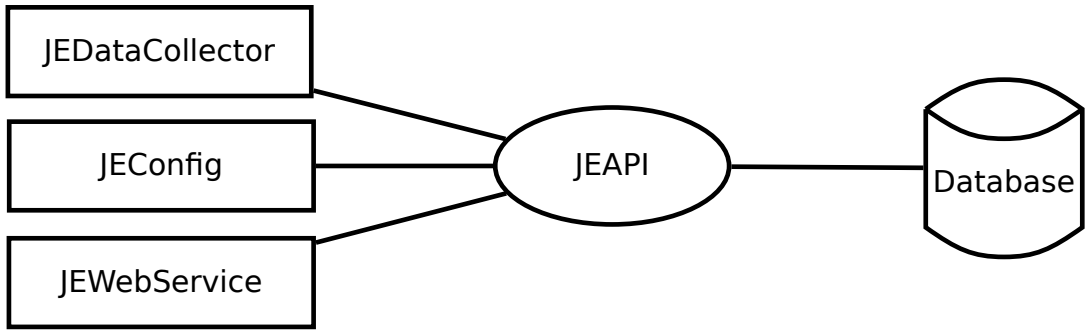


Figure 4.2: JEVis3 overwiev

The system consists of the following components as shown in Figure 4.2:

- Database (MySQL)
- JEConfig
- JEDataCollector - extendable through plugins called *JEDriver*s
- JEWebservice

Distributed instances called *JEDataCollector*s can be deployed to periodically import data into the JEVis database. To configure the JEVis system (configure instances of JEDataCollector, create structures for sensors and much more), a tool name *JEConfig* is provided. *JEWebservice* provides the earlier mentioned RESTful JSON encoded API to access the data structures and data points in the JEVis system. For Java programs, an API called *JEAPI* is defined. All programs part of JEVis are using this API to access and modify the database.

4.1.3.1 JEDataCollector

This program is used to import data from many different sources into the JEVis system. There can be multiple JEDataCollector instances running on different machines. Each instance is configured to connect to a specific JEVis server. From this server, the instances get their configurations. The capabilities of the JEDataCollector can be extended by plugins called JEDrivers. They are made available to the instances by the JEVis server.

At the time of writing, the following drivers are available:

- HTTP-Driver: Acquires raw data from a Web server. The connection can be in plain text (HTTP) or secured (HTTPS).
- FTP-Driver: Acquires raw data from a file transfer server using the FTP protocol. The connection to the server can be in plain text or encrypted.
- sFTP-Driver: Acquires raw data using a secure shell (SSH) connection. These types of connections are encrypted by default.
- CSV-Driver: Parse raw CSV encoded data.
- XML-Driver: Parse raw XML encoded data.

A JEDriver can implement the capabilities *connection*, *parsing* or both at the same time. Examples for drivers with connection capabilities are HTTP-Driver, FTP-Driver and sFTP-Driver. Those drivers acquire data from the specified source. This raw data needs to be processed before it can be imported into the JEVis system. The interpretation of the data is done by drivers with parsing capabilities. Examples for drivers with such capabilities are CSV-Driver and XML-Driver.

For example, assume measurements from a sensor are available through an integrated web-server serving a CSV-file. To acquire the data, the drivers HTTP-Driver and CSV-Driver can be combined. The HTTP-Driver is used to get the raw data in form of the CSV-file. Then the CSV-Driver is configured to parse the time stamp and the value of each measurements saved in the CSV file. Finally, the parsed information can be imported into the JEVis system.

4.1.3.2 Improvement: SQL Driver

As mentioned earlier, external connections to the Desigo database are not allowed. The only direct access to the sensor data is by having local access on the physical machine running Desigo with its MSSQL database.

Desigo stores the monitored building data in a Microsoft SQL database. The most efficient way to reliably import the sensor data from Desigo was to access the Microsoft SQL database used by Desigo locally.

Unfortunately, at the start of this thesis JEVis was not able to import sensor data from an SQL database. Therefore, JEDataCollectors capabilities are extended by a newly developed *SQL JEDriver*. The SQL driver has the capabilities *connection* and *parsing*. This means that the driver is able to connect to a SQL database and parse its content. The parsed data can then be imported into the JEVis system. The new driver supports the SQL databases MySQL and MSSQL. PostgreSQL and other SQL databases could be integrated in the future.

Since the MSSQL database is only accessible locally, an instance of JEDataCollector is deployed on the same server as Desigo.

The available configuration parameters of *SQL JEDriver* and an example are described in Appendix B.

4.2 Ontology Instances

The previously introduced response-description ontology is used to describe responses from various sources. The data providers described in the following sections were chosen because they either offered limited access to their API free of cost or offered unrestricted access through their APIs. The proposed ontology makes it easy to integrate other services if needed.

4.2.1 Geolocation Service

OpenStreetMap is used as geolocation service. It is a community maintained geolocation service and provides free access to its geolocation data. It can be accessed through the Web portal or through an API with XML encoded responses.

The URL used to search for a specific structure in OpenStreetMap is `http://www.overpass-api.de/api/xapi?way[name=<name>]` where '<name>' is replaced by the name of the structure.

Listing 4.1 shows an example response provided by the OpenStreetMap search API. To keep this section readable, the example only shows a section of the actual response. The top level tag (*osm*) contains tags labeled 'note', 'meta', 'node' and 'way'. The tag 'node' contains a tag called 'tag', which has two attributes 'k' and 'v' representing a key-value-pair. One of the key values is 'addr:city'. The interesting data is the value of this pair (in the case of the example 'Wien').

Listing 4.1: Example of a shortened OpenStreetMap XML response

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="Overpass API">
<note>The data included in this document is from
  www.openstreetmap.org. The data is made available
  under ODbL.</note>
<meta osm_base="2016-10-25T11:53:03Z"/>
  <node id="310744920" lat="48.2695697" lon="16.4267149"/>
  <node id="348140578" lat="48.2695654" lon="16.4267635">
    <tag k="entrance" v="main"/>
  </node>
  <way id="28297041">
    <nd ref="310744920"/>
    <nd ref="348140578"/>
    <tag k="addr:city" v="Wien"/>
    <tag k="addr:country" v="AT"/>
    <tag k="addr:housenumber" v="6"/>
    <tag k="addr:postcode" v="1210"/>
    <tag k="addr:street" v="Giefinggasse"/>
    <tag k="building" v="yes"/>
```

```
    <tag k="name" v="ENERGYbase"/>
    <tag k="note" v="FH Technikum Wien Bauteil E
      (&quot;ENERGYbase&quot;)"/>
    <tag k="ref" v="E"/>
  </way>
</osm>
```

Figure 4.3 shows a model of the response description for the OSM API request. It states that the response of the instance 'OSM' is an 'XML Response'. The relevant data is in an XML tag containing another with the label 'tag'. The key-value-pair is modeled as a condition. If an attribute with the tag 'k' contains the value 'addr:city' the value of the attribute with the tag 'v' is returned.

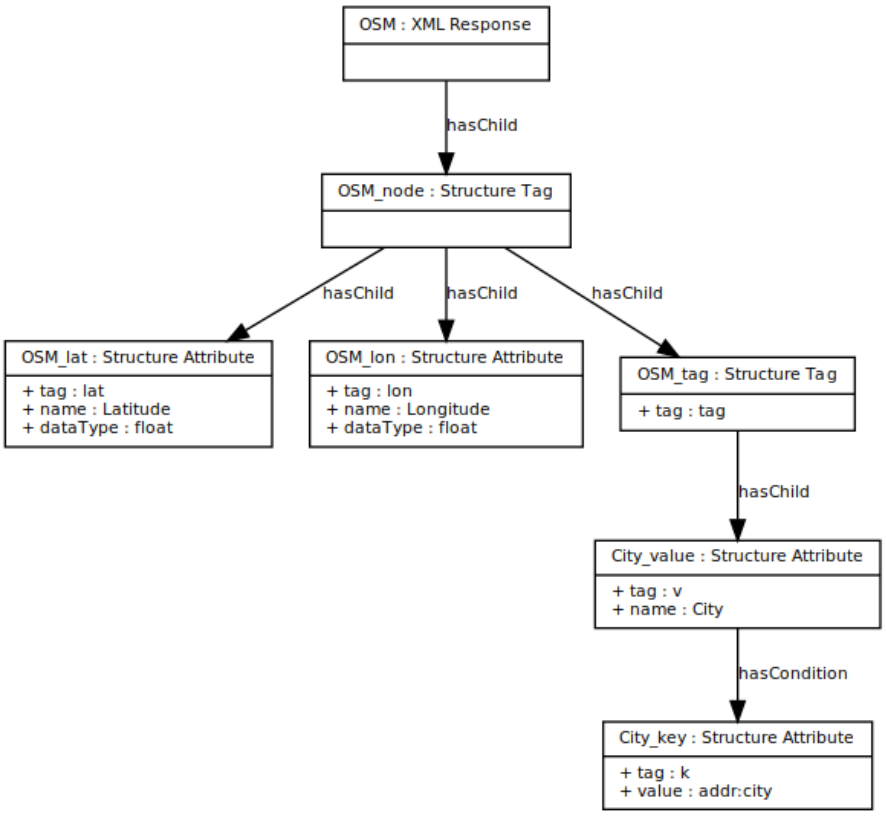


Figure 4.3: OpenStreetMaps response ontology

4.2.2 Weather Service

Weather Underground is used as the weather service.

They provide limited free of cost access to their services through a JSON encoded API. It is also one of the few services providing historical weather data free of charge.

4.2.2.1 Weather Underground Geolookup

The historical weather data is only available for airport weather stations. To find those weather stations, Weather Underground provides a Geolookup service. The URL of this service is `http://api.wunderground.com/api/<api-key>/geolookup/q/<lookup>.json`. The '<api-key>' must be a valid Weather Underground API-key (it can be an account in the free tier). The '<lookup>' string can be the name of the city or the latitude and longitude separated by a comma.

By providing the API-call with a lookup value, the API returns (among other information) a list of airport weather stations and their International Civil Aviation Organization (ICAO) four letter airport codes.

Listing 4.2 shows an example for the JSON encoded result of a Weather Underground Geolookup query. The result is shortened to keep the section readable. The ICAO airport code can be found at `/location/nearby_weather_stations/airport/station/icao`.

Listing 4.2: Example of Weather Underground's Geolookup JSON response

```
{
  "response": {
    "version":"0.1",
    "termsofService":
      "http://www.wunderground.com/weather/api/d/terms.html",
    "features": {
      "geolookup": 1
    }
  },
  "location": {
    "type":"INTLCITY",
    "country":"OS",
    "country_iso3166":"AT",
    "country_name":"Austria",
    "state":"",
    "city":"Wien",
    "tz_short":"CET",
    "tz_long":"Europe/Vienna",
    "lat":"48.20848846",
    "lon":"16.37207603",
    "nearby_weather_stations": {
      "airport": {
        "station": [
          { "city":"Wien / City",
            "state":"",
            "country":"Austria",
            "icao":"",
            "lat":"48.19834137",
            "lon":"16.36643028" },
          { "city":"Vienna Schwechat",
            "state":"", "country":"OS",
            "icao":"LOWW",
```

```
            "lat":"48.11083221",
            "lon":"16.57083321" }
        ]
      }
    }
  }
}
```

In Figure 4.4, the structure of the response of WeatherUndergrounds geolookup query is shown.

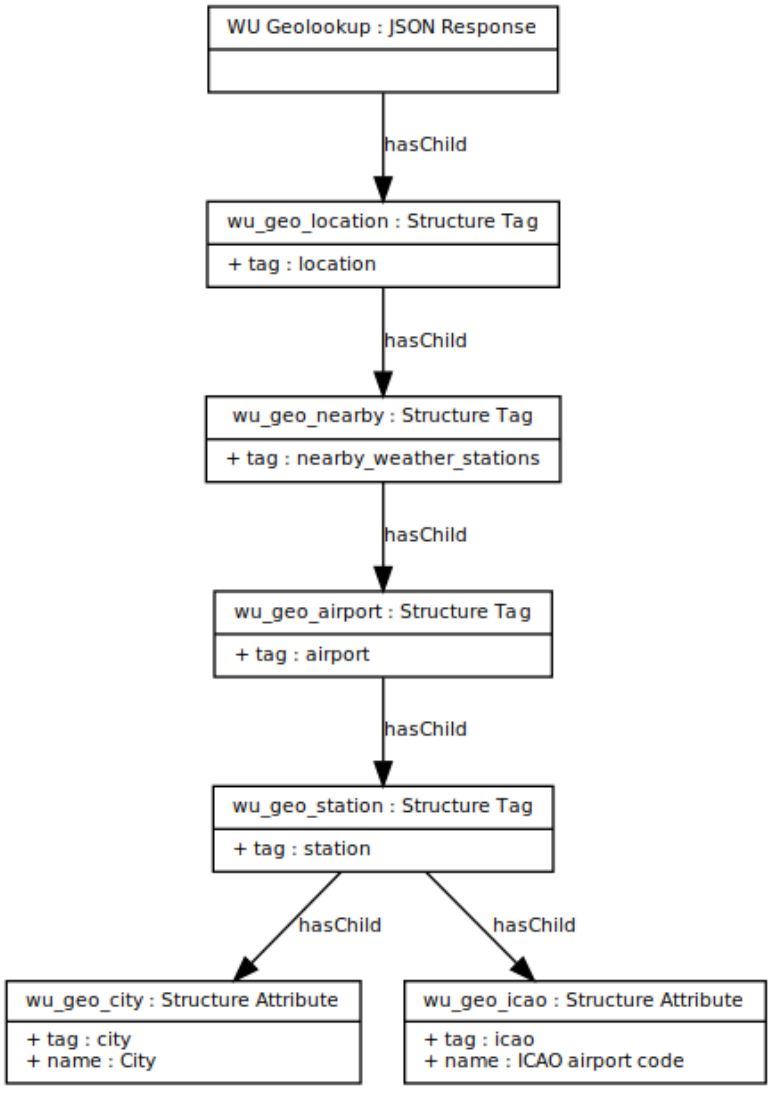


Figure 4.4: Weather Underground Geolookup ontology

4.2.2.2 Weather Underground Historic Data

Using the ICAO identification, historic data from the specified airport weather station can be requested on a day to day basis. The responses are encoded in CSV.

The URL to call the API is of the form `http://www.wunderground.com/history/airport/<icao>/<datestring>/DailyHistory.html?format=1`. '<icao>' needs to be substituted with the ICAO airport code of the earlier found airport. The date-string is in the form '%Y/%m/%d'. Listing 4.3 shows the first two lines of the request `https://www.wunderground.com/history/airport/LOWW/2016/1/21/DailyHistory.html?format=1`.

Listing 4.3: Example of Weather Underground's Historic CSV response

```
TimeCET, TemperatureC ,Dew PointC ,Humidity ,Sea Level PressurehPa ,
  VisibilityKm ,Wind Direction ,Wind SpeedKm/h,Gust SpeedKm/h,
  Precipitationmm ,Events ,Conditions ,WindDirDegrees ,DateUTC
12:00 AM,0,-5,55,1019,35,West,28.8,,,,Overcast,280,
  2016-01-20 23:00:00
```

In Figure 4.5, the structure of the response of Weather Undergrounds historic query is shown. In this thesis, only the time and temperature in degree Celsius are extracted.

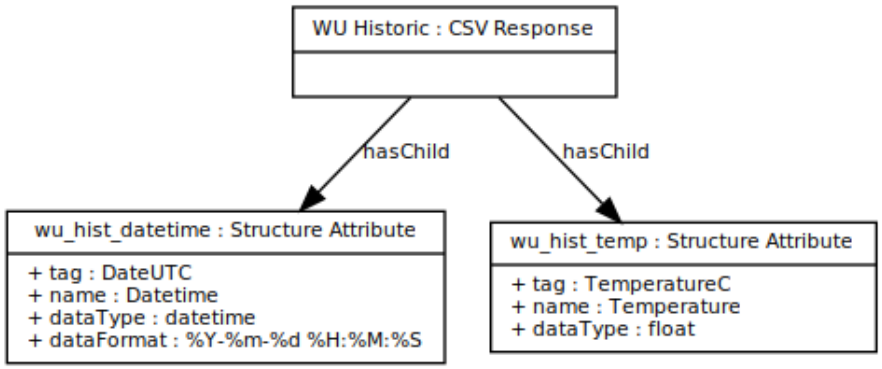


Figure 4.5: Weather Underground historic weather data

4.2.3 Building Information Service

As described in the overview, JEVis is used to import, store and provide the data-points from the BMS. JEVis provides a RESTful interface encoding its responses in JSON. To get sensor data, the unique ID of the data-points in the JEVis system containing the sensor data needs to be found. In another step, the found ID can be used to request corresponding data.

4.2.3.1 JEVis Structure

At the time of writing, JEWebService did not provide a way to get a tree containing all JEVis-nodes with 'id', 'name' and 'jevisclass' by using a single API call. By combining the separate calls to get the root node's ID (`<server>/objects?root=True`), and the information about a specific ID (`<server>/objects/<id>`) including name, class and other node IDs in relationship with the specified node, it is possible to create a tree representing the structure of all nodes in the JEVis system encoded as JSON objects. Listing 4.4 shows the first few entries of a tree defining the structure of the JEVis system.

Listing 4.4: JSON example of a generated JEVis structure

```
{"jevisclass": "System",
  "id": "1", "name": "System", "children": [
    {"jevisclass": "Administration Directory",
      "id": "2", "name": "Administration", "children": [
        {"jevisclass": "Group Directory",
          "id": "10", "name": "Group Directory", "children": [
            ...
          ]
        }
      ]
    }
  ]
}
```

Figure 4.6 shows the ontology used to get the objects containing heat and temperature. The node 'jevis_struct_cond_name_building' has the value '!sub:building!'. The flag '!sub:<name>!' tells the importer using the model to substitute the flag with the value previously specified for 'building'. This reduces the number of duplicate models. If the JEVis structure is the same, but the name of the building is different, the same model can be used.

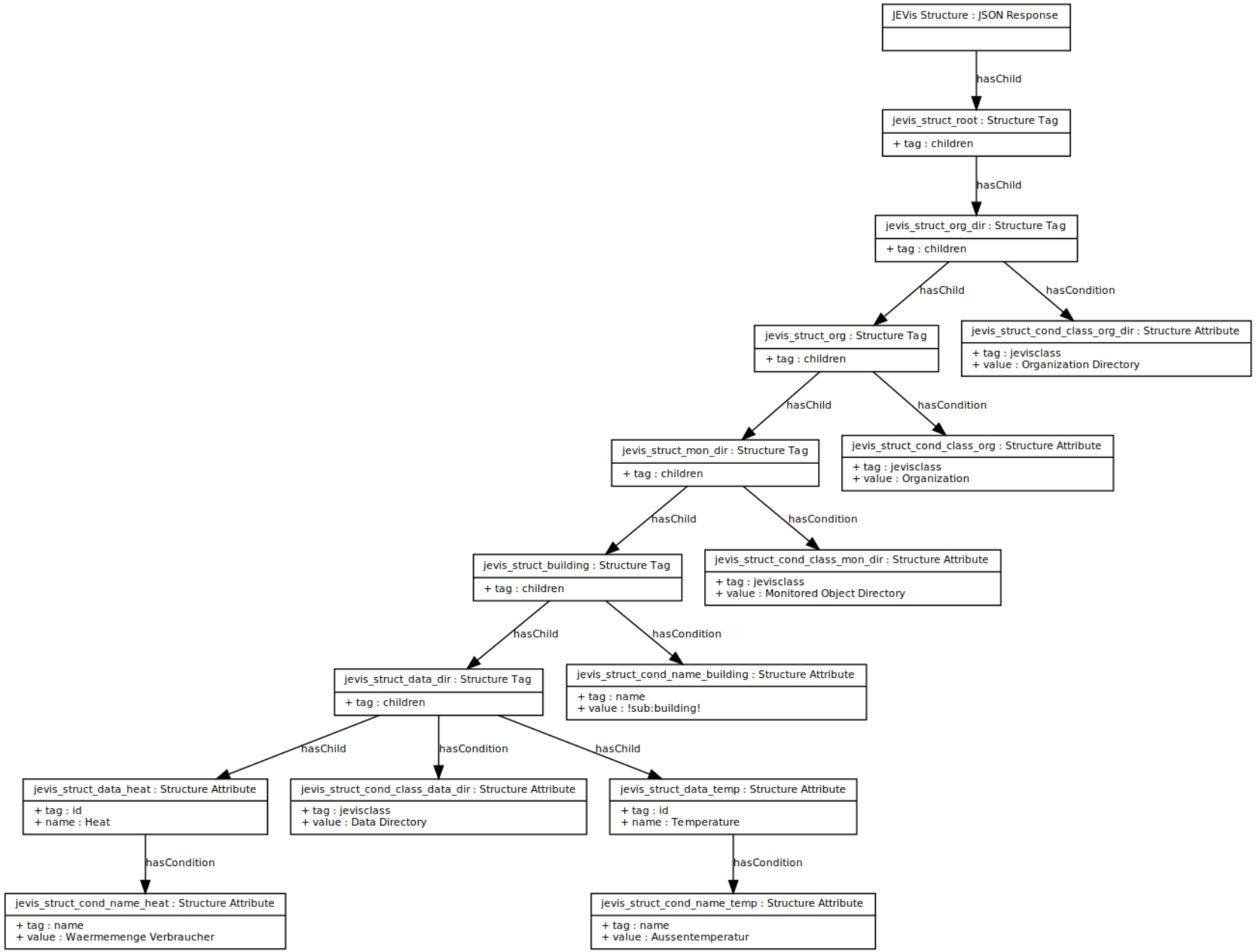


Figure 4.6: JEVis Structure

4.2.3.2 JEVis Data

The unique JEVis IDs can be used to request the data samples from a given time range. The URL to get the samples of a specific ID is `<server>/objects/<id>/attributes/Value/samples?from=<start>&until=<end>`. The datetimes <start> and <end> are in the format '%Y%m%dT%H%M%S'.

An example of the JSON encoded response from JEWebService is shown in Listing 4.5.

Listing 4.5: JSON example of sensor values returned by JEVis

```
{'Sample': [
  {'note': 'Imported by JEDataCollector',
    'value': '23.7653491', 'ts': '2015-09-02T02:00:00.000+02:00'},
  {'note': 'Imported by JEDataCollector',
    'value': '23.0368386', 'ts': '2015-09-02T03:00:00.000+02:00'},
  {'note': 'Imported by JEDataCollector',
    'value': '22.2765468', 'ts': '2015-09-02T04:00:00.000+02:00'},
  {'note': 'Imported by JEDataCollector',
    'value': '21.4838807', 'ts': '2015-09-02T05:00:00.000+02:00'},
  {'note': 'Imported by JEDataCollector',
    'value': '20.6529099', 'ts': '2015-09-02T06:00:00.000+02:00'},
  {'note': 'Imported by JEDataCollector',
    'value': '20.3009703', 'ts': '2015-09-02T07:00:00.000+02:00'}]
}
```

In Figure 4.7 the structure of the response of JEVis samples request is shown. The data samples are contained as JSON-arrays in the JSON object with the tag 'Sample'.

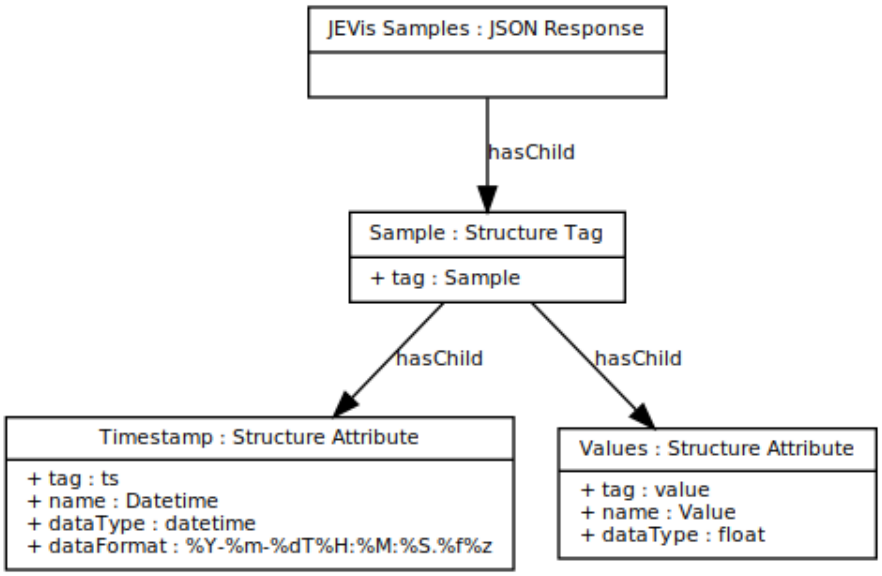


Figure 4.7: JEVis Samples

4.3 SPARQL Queries

So-called *SPARQL Queries* are used to access the information of the created ontology. This section shows the constructed queries used to get the full path from the root of the response to a specified field. Additionally, a way to extract the starting point of the earlier mentioned path is shown.

4.3.1 Root of path

The first SPARQL query checks if there is a path from the specified service to the desired information with a given name. If the service has the required information the first element of the path is returned.

```
PREFIX thesis: <http://example.com/thesis#>

select ?obj ?child ?classlabel
where {
  ?obj rdfs:label "<SERVICE>" .
  ?end thesis:hasName "<NAME>" .

  ?obj thesis:hasChild* ?end .
  ?obj thesis:hasChild ?child .
  ?child thesis:hasChild* ?end .

  ?obj rdf:type [rdfs:label ?classlabel] .
}
```

Before sending the query to the SPARQL endpoint for processing the placeholders <SERVICE> and <NAME> have to be replaced. In this thesis, PyCaster (Section 4.4) is responsible for this subsitution. The service is identified by its attribute 'label'. In this thesis, the following values for <SERVICE> are available:

- `OSM`
- `WU Geolookup`
- `WU Historic`
- `JEVis Structure`

The node is identified by its attribute `hasName`. In the query the placeholder <NAME> needs to be replaced. Examples are "City", "Datetime" and "Value".

Table 4.1 shows the result of the query when replacing "<SERVICE>" with "OSM" and "<NAME>" with "City". Figure 4.3 shows the full ontology instance "OSM". The root node is in the column `obj`. As additional information, the label of the class 'classlabel' and the next element in the path after the root node (in the column `child`) are shown. The `classlabel` can be used to determine the encoding (XML, JSON, CSV) of the response.

obj	child	classlabel
<http://example.com/thesis#OSM>	<http://example.com/thesis#OSM_node>	XML Response

Table 4.1: Root of the path from "OSM" to "City"

4.3.2 Getting the full path

The next query is an extension of the previous one. Instead of just returning the first element of a path the entire path is returned. The placeholders <SERVICE> and <NAME> need to be replaced like before.

```
PREFIX thesis: <http://example.com/thesis#>

select ?obj ?child ?tag ?classlabel ?condTag ?condValue ?dataType ?dataFormat
where {
  ?start rdfs:label "<SERVICE>" .
  ?end thesis:hasName "<NAME>" .

  ?obj thesis:hasChild* ?end .
  ?start thesis:hasChild* ?obj .

  OPTIONAL {
    ?obj thesis:hasChild ?child .
    ?child thesis:hasChild* ?end .
  }
  OPTIONAL {
        ?obj thesis:tag ?tag .
  }

  OPTIONAL {
    ?obj thesis:hasCondition [thesis:tag ?condTag] .
    ?obj thesis:hasCondition [thesis:value ?condValue] .
  }
  OPTIONAL {
    ?obj thesis:dataType ?dataType .
  }
  OPTIONAL {
    ?obj thesis:dataFormat ?dataFormat .
  }
  ?obj rdf:type [rdfs:label ?classlabel] .
}
```

In Table 4.2, the result for the query from "OSM" to "City" is shown. Figure 4.8 visualizes this path by showing only the nodes of the "OSM" model, which are part of the path from "OSM" to "City". The query produces a list of edges with some additional information for the current node listed in the column 'obj'.

obj	child	tag	classlabel	condTag	condValue
<http://example.com/thesis#City_value>		v	Structure Attribute	k	addr:city
<http://example.com/thesis#OSM>	<http://example.com/thesis#OSM_node>		XML Response		
<http://example.com/thesis#OSM_node>	<http://example.com/thesis#OSM_tag>		Structure Tag		
<http://example.com/thesis#OSM_tag>	<http://example.com/thesis#City_value>	tag	Structure Tag		

Table 4.2: Path from "OSM" to "City"

The table shows the path from the node 'OSM', through 'OSM_node' and 'OSM_tag', to 'City_value'. The last node has a 'condTag' of 'k' with a conditional value 'addr:city'. This tells the parser to only access the value if there is a node with a tag of 'k' and a value of 'addr:city' in the XML document.

The columns 'dataType' and 'dataFormat' are emtpy for this particular query and have been left out to save space. These columns indicate to the parser which type the data point is and in the case of a date string its format.

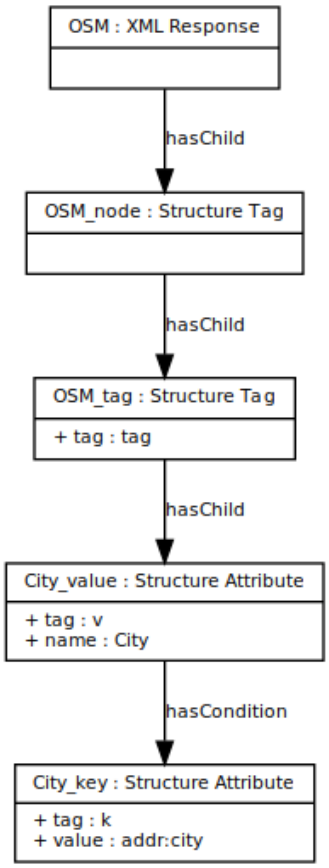


Figure 4.8: Graph visualizing path from "OSM" to "City"

The complexity of analysing the graph, determining the root node and the whole path is offloaded to the SPARQL endpoint using the queries above. This reduces the parser's complexity and thus reduces the likelihood of bugs.

4.4 PyCaster

PyCaster is the heart of the implemented prototype system. It uses the Blazegraph server to access the developed ontologies. Those ontologies are used to extract the required information from the various data sources (OpenStreetMap, Weather Underground, JEVis). The extracted data points are then preprocessed to eliminate obvious errors in the sensor data, resampled, interpolated as well as scaled to be in a range of 0 to 1. The preprocessed data is fed to a Support Vector Machine to learn patterns in the input data. The learned model can then be used to create one day predictions of heat energy load. Those predictions are compared to the actual measured data points. As an accuracy measure, the daily root mean square error between prediction and measurement is calculated. For visual inspection, plots of the errors are created.

PyCaster is written in the scripting language Python [GvR01]. Python is cross platform and provides many tools for easy and fast prototyping. The standard capabilities of Python can be extended by modules. For PyCaster, the following modules are used:

- Requests [req11]

  Requests provides an easy way to do HTTP requests. It is used to communicate with all Web services. It also provides automatic conversion from JSON to Python's dictionary structure.

- scikit-learn [PVG+11]

  Provides simple and efficient tools for data mining and data analysis. For this thesis, the Support Vector Machine (SVM) implemented by this module is used.

- pandas [McK10]

  Provides high performance and easy to use data structures and data analysis tools.

- numpy [WCV11]

  Numerical Python provides powerful N-dimensional arrays and linear algebra capabilities among others. This module is a dependency of the pandas module.

- matplotlib [Hun07]

  This module is another requirement of the pandas module. It provides a 2D plotting library with a MATLAB-like interface.

4.4.1 Data Import

4.4.1.1 Sparql-py

'Sparql-py' is a simple module used to wrap the access to the Blazegraph server. Queries to Blazegraph are HTTP-POST requests to `<server>/blazegraph/namespace/<namespace>/sparql`. In the case of a locally running Blazegraph server and the namespace 'tags', the URL looks as follows `http://localhost:9999/blazegraph/namespace/tags/sparql`.

The actual SPARQL query is in the data field called 'query' of the POST request. Because JSON to python-dictionary transformation is a built in feature of *Requests* the response of Blazegraph is requested to be encoded in JSON. This is done by setting the data field 'format' to the value 'json'.

4.4.1.2 jeapi-py

At the time of writing, no Python wrapper to access the JEVis system was available. Therefore 'jeapi-py' a Python wrapper for JEWebService was created. It provides a Python interface to the API-calls of JEWebService. Further convenience functions like 'getChildren' to get all IDs of children of a specified ID, or 'getSamples' to get all samples for a specified ID, are provided. The formatting of date times for requests is handled by 'jeapi-py'.

The module is available at `https://github.com/AIT-JEVis/jeapipy`.

4.4.2 Data Processing

As a next step, the imported data is processed and used to calculate one day heat energy consumption forecasts. A Support Vector Machine (SVM) provided by the module *scikit-learn* [PVG+11] is used to generate the forecasts. Before the data is usable by the SVM, it first needs to be pre-processed.

- The first step is to resample and interpolate the data at a fixed time step. In this thesis, an interval of one hour is chosen.
- SVMs are not scale invariant in regard to their input data. Therefore, the input data is scaled to be in a range of -1 to 1. The temperature for example is scaled down from a range [-40,40] to [-1,1].
- For a forecast of the heat consumption of the next day, the data of the current day and a temperature forecast of the next day are used.

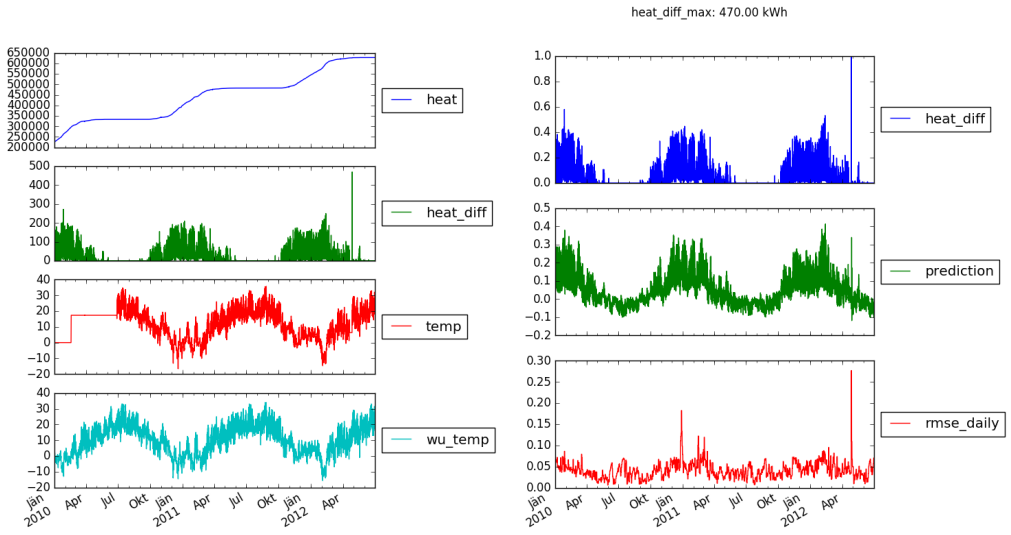
In order to use an SVM, it needs to be trained with training data consisting of input data and the expected output (in this case, the next day hourly heat consumption). Afterwards, the SVM can be tested by calculating forecasts and comparing them with the actual measured heat consumptions.

CHAPTER 5

Evaluation

5.1 Energy Data

The evaluation of the input data is from January 1 2010 until June 30 2012.



(a) Overview over used data (b) Root Mean Square Error of prediction

Figure 5.1: Overview over data and prediction

An overview of the evaluated data input can be seen in Figure 5.1a. The values 'heat' and 'temp' were monitored at the office building 'EnergyBase' located in Vienna. To collect the sensor data from the building, a building management system (Desigo) was used. The first subplot is labeled 'heat'. It shows the accumulated heat consumption of the building 'EnergyBase' in kWh. The second subplot labeled as 'heat_diff' is the

hourly difference of the values of 'heat'. It can be observed, that in the winter months heat energy consumption is high, and in the summer nearly no energy is used. The third subplot is labeled 'temp'. This plot shows the measured outdoor temperature at the roof of the building 'EnergyBase'. The temperature is given in degree Celsius. The last plot labeled as 'wu_temp' shows the measured temperature provided by Weather Underground's Historic service. The temperature is from the weather station located at the airport 'Wien Schwechat'.

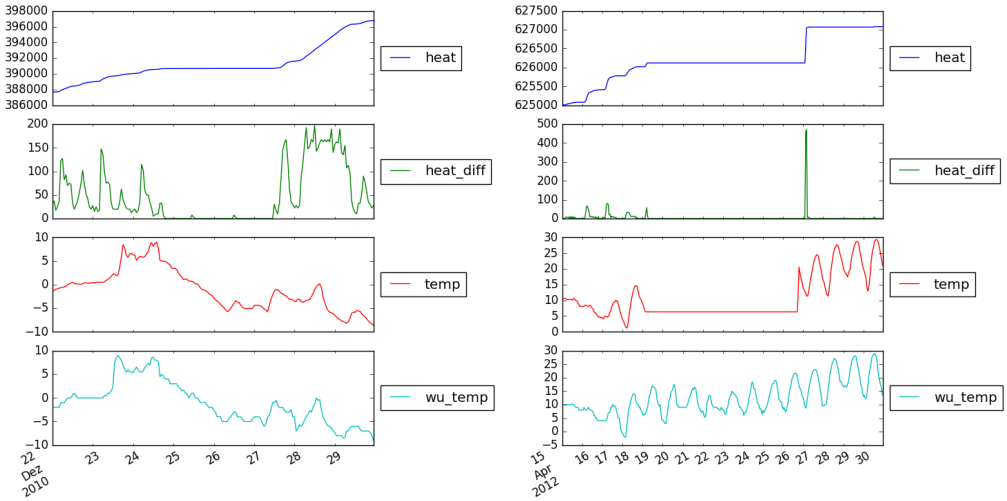
For the forecast, the input values 'heat_diff' and 'wu_temp' are used. The values of 'temp' are used for the later inspection of anomalies. The data from 2010 is used as the training set. Predicted values are calculated for the whole input time range.

Figure 5.1b shows the analysis of the forecast. The first two graphs show the actual 'heat_diff' as measured and the predicted heat difference. The root mean square error (RMSE) was chosen as the error metric. The last graph in Figure 5.1b shows the root mean square error per day.

In the daily RMSE graph two spikes can be seen. Those spikes each indicate an anomaly in the input data.

5.2 Anomalies

After the calculation of a forecast, the calculated predictions are compared to the real heat consumption values. When a high mean square error is encountered the facility manager could use this indicator to investigate further. In this section, the two highest RMSE spikes are investigated.



(a) Anomaly December 2010 (b) Anomaly April 2012

Figure 5.2: Found anomalies

Figure 5.2a shows an anomaly during the last month of 2010. Between December 25 and 27 there was almost no heat consumption. During the previous days a regular heat pattern is observable. Afterwards, there were three days with a high heat consumption throughout the day. A possible explanation is that the winter holidays were used to test the heating system and isolation of the building. Afterwards, the building was brought back to a comfortable level.

Figure 5.2b shows an anomaly at the end of April 2012. From the 20th to 26th, the heat consumption stayed at the same value. On the 27th, the biggest spike of the whole data set occurred. Additionally as seen in the third graph labeled 'temp', the temperature recorded by the BMS did not change for a whole week. The fact that both sensor values ('heat' and 'temp') did not change for six days hints that the system did not operate as planned. Possible explanation are a system outage or a maintenance of the system.

CHAPTER 6

Conclusion

An ontology to model data points in responses from web services was developed. To verify its usefulness, a program to import temperature and heat consumption data of a building from different web services was created. The imported data points were used to create one day heat energy consumption forecasts. In this thesis, the ontology is used to configure the data importing program.

The proposed strategy could be used at different scales, such as room level, floor level, building level or even building complexes. Due to the nature of the available data at EnergyBase, the building layer was chosen as a focus of this thesis.

Apart from directly using the extracted data points the developed response description ontology could be used with other data storing ontologies to model how to import data into the ontology.

The rapid growth of the number of Internet of Things (IoT) devices also increases the number of possible data sources. With this increase the complexity of manually selecting data points and writing specialized importers also rises. The proposed approach could be used in combination with other ontologies and reasoners to aid in their selection. Currently, many data sources are not modeled by ontologies. The proposed ontology provides a way to model the structure of the responses of such data sources. In its current implementation, the data points of the different modeled responses are not linked to each other. The only way to link the data points with the proposes ontology is using the same name for related values.

A proposal for future work is to explicitly model connections between data points or use other ontologies to better describe each data point. The information could be used to automatically find and provide the user with the connected data points from the available sources. Because of the decentralised way of ontologies, other ones may be linked to the response description ontology to model the connections between data points and bridge the gap between data sources and ontologies.

APPENDIX A

Ontology

A.1 response description

In the following the response description ontology developed in this thesis is shown using the turtle syntax.

```
@prefix : <http://webprotege.stanford.edu/project/CVfEBMo9QGaGLnAuzg5AVr#>
    .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix thesis: <http://example.com/thesis#> .
@base <http://webprotege.stanford.edu/project/CVfEBMo9QGaGLnAuzg5AVr> .

<http://webprotege.stanford.edu/project/CVfEBMo9QGaGLnAuzg5AVr> rdf:type
    owl:Ontology .


#################################################################
#
#    Object Properties
#
#################################################################

###  http://example.com/thesis#hasRoot
thesis:hasRoot rdf:type owl:ObjectProperty ;
        rdfs:label "hasRoot" .

###  http://example.com/thesis#hasChild
thesis:hasChild rdf:type owl:ObjectProperty ;
        rdfs:label "hasChild" .
```

```
###  http://example.com/thesis#hasCondition
thesis:hasCondition rdf:type owl:ObjectProperty ;
        rdfs:label "hasCondition" .

#################################################################
#
#    Data properties
#
#################################################################



###  http://example.com/thesis#hasName
thesis:hasName rdf:type owl:DatatypeProperty ;
        rdfs:label "hasName" .

###  http://example.com/thesis#dataType
thesis:dataType rdf:type owl:DatatypeProperty ;
        rdfs:label "dataType" .

###  http://example.com/thesis#dataFormat
thesis:dataFormat rdf:type owl:DatatypeProperty ;
        rdfs:label "dataFormat" .

###  http://example.com/thesis#value
thesis:value rdf:type owl:DatatypeProperty ;
        rdfs:label "value" .

###  http://example.com/thesis#longName
thesis:longName rdf:type owl:DatatypeProperty ;
        rdfs:label "longName" .

###  http://example.com/thesis#tag
thesis:tag rdf:type owl:DatatypeProperty ;
        rdfs:label "tag" .


#################################################################
#
#    Classes
#
#################################################################

###  http://example.com/thesis#Response
thesis:Response rdf:type owl:Class ;
        rdfs:label "Response" ;
        rdfs:subClassOf owl:Thing ,
                [ rdf:type owl:Restriction ;
                        owl:onProperty thesis:longName ;
                        owl:someValuesFrom xsd:string
                ] ,
                [ rdf:type owl:Restriction ;
                        owl:onProperty thesis:hasChild ;
```

```
                owl:someValuesFrom thesis:StructureObject
        ] .

###  http://example.com/thesis#JSONResponse
thesis:JSONResponse rdf:type owl:Class ;
        rdfs:label "JSON Response" ;
        rdfs:subClassOf thesis:Response .

###  http://example.com/thesis#XMLResponse
thesis:XMLResponse rdf:type owl:Class ;
        rdfs:label "XML Response" ;
        rdfs:subClassOf thesis:Response .

###  http://example.com/thesis#CSVResponse
thesis:CSVResponse rdf:type owl:Class ;
        rdfs:label "CSV Response" ;
        rdfs:subClassOf thesis:Response .

###  http://example.com/thesis#StructureObject
thesis:StructureObject rdf:type owl:Class ;
        rdfs:label "Structure Object" ;
        rdfs:subClassOf owl:Thing ,
                [ rdf:type owl:Restriction ;
                        owl:onProperty thesis:hasCondition ;
                        owl:someValuesFrom thesis:StructureAttribute
                ] ,
                [ rdf:type owl:Restriction ;
                        owl:onProperty thesis:tag ;
                        owl:someValuesFrom xsd:string
                ] ,
                [ rdf:type owl:Restriction ;
                        owl:onProperty thesis:hasName ;
                        owl:someValuesFrom xsd:string
                ] .

###  http://example.com/thesis#StructureTag
thesis:StructureTag rdf:type owl:Class ;
        rdfs:label "Structure Tag" ;
        rdfs:subClassOf thesis:StructureObject ,
                [ rdf:type owl:Restriction ;
                        owl:onProperty thesis:hasChild ;
                        owl:someValuesFrom thesis:StructureObject
                ] .

###  http://example.com/thesis#StructureAttribute
thesis:StructureAttribute rdf:type owl:Class ;
        rdfs:label "Structure Attribute" ;
        rdfs:subClassOf thesis:StructureObject ,
                [ rdf:type owl:Restriction ;
                        owl:onProperty thesis:dataType ;
                        owl:someValuesFrom xsd:string
                ] ,
                [ rdf:type owl:Restriction ;
```

```
                owl:onProperty thesis:dataFormat ;
                owl:someValuesFrom xsd:string
        ] ,
        [ rdf:type owl:Restriction ;
                owl:onProperty thesis:value ;
                owl:someValuesFrom xsd:string
        ] .


#################################################################
#
#    Individuals
#
#################################################################

# OSM
# - osm_tag
#  - osm_tag - no name
#   - City - keypair
#    - City_key
#    - City_value

###  http://example.com/thesis#OSM
thesis:OSM rdf:type thesis:XMLResponse ,
        owl:NamedIndividual ;
        rdfs:label          "OSM" ;
        thesis:longName     "OpenStreetMap" ;
        thesis:hasChild     thesis:OSM_node .

###  http://example.com/thesis#OSM_node
thesis:OSM_node rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label          "OSM_node" ;
        thesis:hasChild     thesis:OSM_tag ;
        thesis:hasChild     thesis:OSM_lat ;
        thesis:hasChild     thesis:OSM_lon .

###  http://example.com/thesis#OSM_lat
thesis:OSM_lat rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "OSM_lat" ;
        thesis:hasName      "Latitude" ;
        thesis:tag          "lat" ;
        thesis:dataType     "float" .

###  http://example.com/thesis#OSM_lon
thesis:OSM_lon rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "OSM_lon" ;
        thesis:hasName      "Longitude" ;
        thesis:tag          "lon" ;
        thesis:dataType     "float" .
```

```
###  http://example.com/thesis#OSM_tag
thesis:OSM_tag rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label          "OSM_tag" ;
        thesis:tag          "tag" ;
        thesis:hasChild     thesis:City_value .

###  http://example.com/thesis#City_key
thesis:City_key rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "City_key" ;
        thesis:value        "addr:city" ;
        thesis:tag          "k" .

###  http://example.com/thesis#City_value
thesis:City_value rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        thesis:hasCondition thesis:City_key ;
        thesis:hasName      "City" ;
        rdfs:label          "City_value" ;
        thesis:tag          "v" .


# WU Geolookup
# - wu_geo_root
#  - wu_geo_location                  - location
#   - wu_geo_nearby_weather_stations - nearby_weather_stations
#    - wu_geo_airport                  - airport
#     - wu_geo_station                  - station
#      - wu_geo_station_array           -
#       - wu_geo_city                    - city
#       - wu_geo_icao                    - icao - Airport

###  http://example.com/thesis#wu_geolookup
thesis:wu_geolookup rdf:type thesis:JSONResponse ,
        owl:NamedIndividual ;
        rdfs:label          "WU Geolookup" ;
        thesis:longName     "Weather Underground Geolookup" ;
        thesis:hasChild     thesis:wu_geo_location .

###  http://example.com/thesis#wu_geo_location
thesis:wu_geo_location rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label          "wu_geo_location" ;
        thesis:tag          "location" ;
        thesis:hasChild     thesis:wu_geo_nearby_ws .

###  http://example.com/thesis#wu_geo_station
thesis:wu_geo_nearby_ws rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label          "wu_geo_nearby" ;
        thesis:tag          "nearby_weather_stations" ;
        thesis:hasChild     thesis:wu_geo_airport .
```

```
### http://example.com/thesis#wu_geo_airport
thesis:wu_geo_airport rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label          "wu_geo_airport" ;
        thesis:tag          "airport" ;
        thesis:hasChild     thesis:wu_geo_station .

### http://example.com/thesis#wu_geo_station
thesis:wu_geo_station rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label          "wu_geo_station" ;
        thesis:tag          "station" ;
        thesis:hasChild     thesis:wu_geo_city ;
        thesis:hasChild     thesis:wu_geo_icao .

### http://example.com/thesis#wu_geo_city
thesis:wu_geo_city rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "wu_geo_city" ;
        thesis:tag          "city" ;
        thesis:hasName      "City" .

### http://example.com/thesis#wu_geo_icao
thesis:wu_geo_icao rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "wu_geo_icao" ;
        thesis:tag          "icao" ;
        thesis:hasName      "ICAO␣airport␣code" .


# WU Historic
# - wu_hist_root
# -   wu_hist_temp
# -   wu_hist_datetime

### http://example.com/thesis#wu_historic
thesis:wu_historic rdf:type thesis:CSVResponse ,
        owl:NamedIndividual ;
        rdfs:label          "WU␣Historic" ;
        thesis:longName     "Weather␣Underground␣Historic␣Data" ;
        thesis:hasChild     thesis:wu_hist_datetime ;
        thesis:hasChild     thesis:wu_hist_temp .

### http://example.com/thesis#wu_hist_datetime
thesis:wu_hist_datetime rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "wu_hist_datetime" ;
        thesis:tag          "DateUTC" ;
        thesis:hasName      "Datetime" ;
        thesis:dataType     "datetime" ;
        thesis:dataFormat   "%Y-%m-%d␣%H:%M:%S" .
```

```
###  http://example.com/thesis#wu_hist_temp
thesis:wu_hist_temp rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "wu_hist_temp" ;
        thesis:tag          "TemperatureC" ;
        thesis:hasName      "Temperature" ;
        thesis:dataType     "float" .


#- System (JEVis root)
#  - Organization Directory
#    - Organization - Desigo
#      - Monitored Object Directory
#        - Building - ENERGYBase
#          - Data Directory
#            - Waerememenge Verbraucher
#            - Aussentemperatur

###  http://example.com/thesis#jevis_structure
thesis:jevis_structure rdf:type thesis:JSONResponse ,
        owl:NamedIndividual ;
        rdfs:label          "JEVis Structure" ;
        thesis:longName     "JEVis Tree Structure Description" ;
        thesis:hasChild     thesis:jevis_struct_root .

###  http://example.com/thesis#jevis_struct_root
thesis:jevis_struct_root rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label          "jevis_struct_root" ;
        thesis:tag          "children" ;
        thesis:hasChild     thesis:jevis_struct_org_dir .

###  http://example.com/thesis#jevis_struct_org_dir
thesis:jevis_struct_org_dir rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label          "jevis_struct_org_dir" ;
        thesis:tag          "children" ;
        thesis:hasCondition thesis:jevis_struct_cond_class_org_dir ;
        thesis:hasChild     thesis:jevis_struct_org .

###  http://example.com/thesis#jevis_struct_cond_class_org_dir
thesis:jevis_struct_cond_class_org_dir rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "jevis_struct_cond_class_org_dir" ;
        thesis:tag          "jevisclass" ;
        thesis:value        "Organization Directory" .

###  http://example.com/thesis#jevis_struct_org
thesis:jevis_struct_org rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label          "jevis_struct_org" ;
        thesis:tag          "children" ;
        thesis:hasCondition thesis:jevis_struct_cond_class_org ;
```

```
        thesis:hasChild      thesis:jevis_struct_mon_dir .

###  http://example.com/thesis#jevis_struct_cond_class_org
thesis:jevis_struct_cond_class_org rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label           "jevis_struct_cond_class_org" ;
        thesis:tag           "jevisclass" ;
        thesis:value         "Organization" .

###  http://example.com/thesis#jevis_struct_mon_dir
thesis:jevis_struct_mon_dir rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label           "jevis_struct_mon_dir" ;
        thesis:tag           "children" ;
        thesis:hasCondition  thesis:jevis_struct_cond_class_mon_dir ;
        thesis:hasChild      thesis:jevis_struct_building .

###  http://example.com/thesis#jevis_struct_cond_class_mon_dir
thesis:jevis_struct_cond_class_mon_dir rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label           "jevis_struct_cond_class_mon_dir" ;
        thesis:tag           "jevisclass" ;
        thesis:value         "Monitored Object Directory" .

###  http://example.com/thesis#jevis_struct_building
thesis:jevis_struct_building rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label           "jevis_struct_building" ;
        thesis:tag           "children" ;
        thesis:hasCondition  thesis:jevis_struct_cond_name_building ;
        thesis:hasChild      thesis:jevis_struct_data_dir .

###  http://example.com/thesis#jevis_struct_cond_name_building
thesis:jevis_struct_cond_name_building rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label           "jevis_struct_cond_name_building" ;
        thesis:tag           "name" ;
        thesis:value         "!sub:building!" .

###  http://example.com/thesis#jevis_struct_data_dir
thesis:jevis_struct_data_dir rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label           "jevis_struct_data_dir" ;
        thesis:tag           "children" ;
        thesis:hasCondition  thesis:jevis_struct_cond_class_data_dir ;
        thesis:hasChild      thesis:jevis_struct_data_heat ;
        thesis:hasChild      thesis:jevis_struct_data_temp .

###  http://example.com/thesis#jevis_struct_cond_class_data_dir
thesis:jevis_struct_cond_class_data_dir rdf:type thesis:StructureAttribute
    ,
        owl:NamedIndividual ;
        rdfs:label           "jevis_struct_cond_class_data_dir" ;
```

```
        thesis:tag          "jevisclass" ;
        thesis:value        "Data Directory" .

###  http://example.com/thesis#jevis_struct_data_heat
thesis:jevis_struct_data_heat rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "jevis_struct_data_heat" ;
    thesis:hasName      "Heat" ;
        thesis:tag          "id" ;
        thesis:hasCondition thesis:jevis_struct_cond_name_heat .

###  http://example.com/thesis#jevis_struct_cond_name_heat
thesis:jevis_struct_cond_name_heat rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "jevis_struct_cond_name_heat" ;
        thesis:tag          "name" ;
        thesis:value        "Waermemenge Verbraucher" .

###  http://example.com/thesis#jevis_struct_data_temp
thesis:jevis_struct_data_temp rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "jevis_struct_data_temp" ;
    thesis:hasName      "Temperature" ;
        thesis:tag          "id" ;
        thesis:hasCondition thesis:jevis_struct_cond_name_temp .

###  http://example.com/thesis#jevis_struct_cond_name_temp
thesis:jevis_struct_cond_name_temp rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "jevis_struct_cond_name_temp" ;
        thesis:tag          "name" ;
        thesis:value        "Aussentemperatur" .


# JEVis Sample
# - Sample
#   - Timestamp - ts - '%Y-%m-%dT%H:%M:%S.%f%z'
#   - Value - value

###  http://example.com/thesis#jevis_samples
thesis:jevis_samples rdf:type thesis:JSONResponse ,
        owl:NamedIndividual ;
        rdfs:label          "JEVis Samples" ;
        thesis:longName     "JEVis Samples description" ;
        thesis:hasChild     thesis:jevis_samples_root .

###  http://example.com/thesis#wu_hist_datetime
thesis:jevis_samples_root rdf:type thesis:StructureTag ,
        owl:NamedIndividual ;
        rdfs:label          "Sample" ;
        thesis:tag          "Sample" ;
        thesis:hasChild     thesis:jevis_samples_ts ;
        thesis:hasChild     thesis:jevis_samples_value .
```

```
###  http://example.com/thesis#jevis_samples_ts
thesis:jevis_samples_ts rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "Timestamp" ;
        thesis:tag          "ts" ;
        thesis:hasName      "Datetime" ;
        thesis:dataType     "datetime" ;
        thesis:dataFormat   "%Y-%m-%dT%H:%M:%S.%f%z" .

###  http://example.com/thesis#jevis_samples_value
thesis:jevis_samples_value rdf:type thesis:StructureAttribute ,
        owl:NamedIndividual ;
        rdfs:label          "Values" ;
        thesis:tag          "value" ;
        thesis:hasName      "Value" ;
        thesis:dataType     "float" .

###  Generated by the OWL API (version 3.5.0) http://owlapi.sourceforge.net
```

APPENDIX B

SQL driver

This chapter describes the developed SQL driver to import data from MySQL and MSSQL databases into JEVis. Section B.1 describes the available and mandatory configuration options. In Section B.2 the process of a data import from Desigo to JEVis is shown with the help of a message sequence chart.

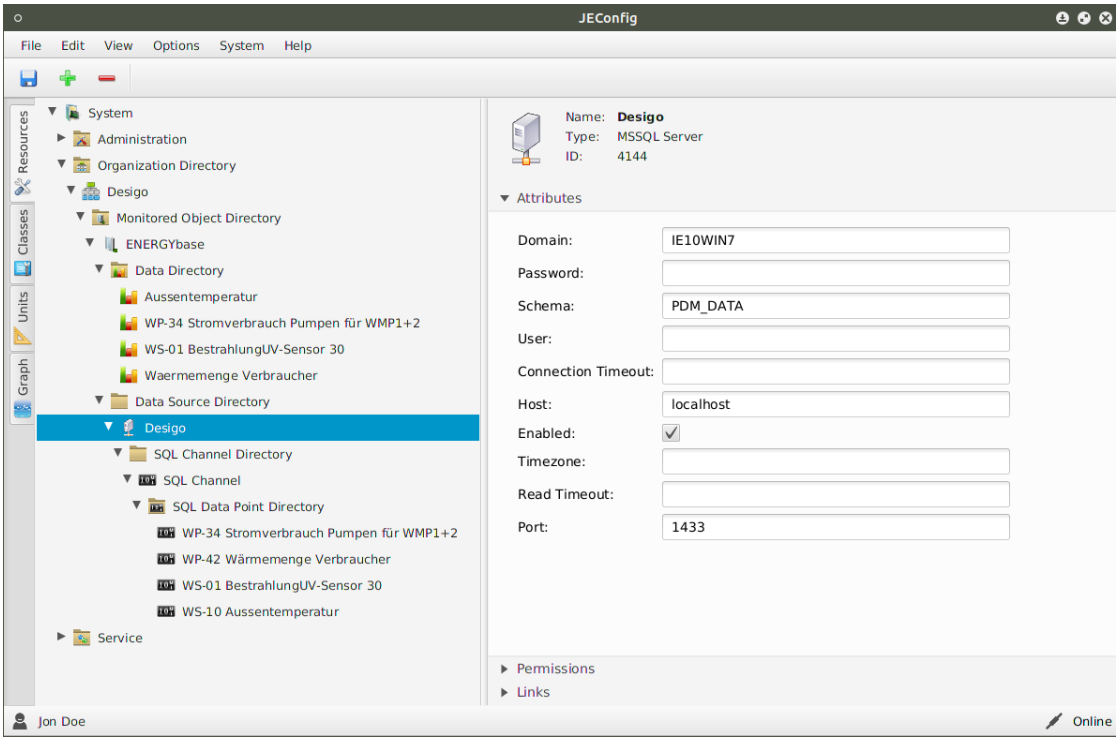


Figure B.1: JEConfig showing SQL DataServer Desigo

B.1 Configuration

The following JEVis classes are used to establish an SQL connection using the specified connection and parsing information.

- *SQL DataServer*

  The *SQL DataServer* contains the information on how to connect to the SQL server.

  - *Host* and *Port*
    These attributes describe where the SQL server is listening for incoming connections. For a local MySQL database, the *Host* would be 'localhost' with the *Port* '3306'.
  - *Schema*

    The name of the database which contains the tables with data samples. An SQL server can have multiple databases.
  - *Domain*, *User* and *Password*
    These attributes describe the credentials to authenticate with the SQL server. If the *Domain* is empty the driver uses regular SQL Server authentication with the provided *User* and *Password*. If the attribute *Domain* is set then the driver uses Windows authentication (NTML) to sign in. In this case, the provided credentials are the domain user and password.

- *SQL Channel Directory*
  A virtual JEVis folder for multiple instances of *SQL Channel*.

- *SQL Channel*

  The *SQL Channel* has attributes to describe the table containing the measurements. This includes the name of the table and the names of the columns for ID, timestamp and value.

  - *Table*
    Identifies the table of the database containing the samples.
  - *Column ID*
    The name of the column containing the ID(s) of the sensor(s). If *Column ID* is left empty the driver assumes that there is no such column and will not limit the query to an ID.
  - *Column Timestamp*
    Identifies the column which holds the timestamp of the samples.
  - *Timestamp Format*
    The format string used to parse the timestamp.
  - *Column Value*
    Identifies the column containing the value of the samples.

- *SQL Data Point Directory*
  A virtual JEVis folder for multiple instances of *SQL Data Point*.

- *SQL Data Point*
  This class specifies the sensor and where the samples are stored in JEVis.

  - *ID*
    The string or number identifying the sensor in the previously specified table.
  - *Target*
    The JEVis internal ID of the data object in which to store the samples.

B.2 Message Sequence chart

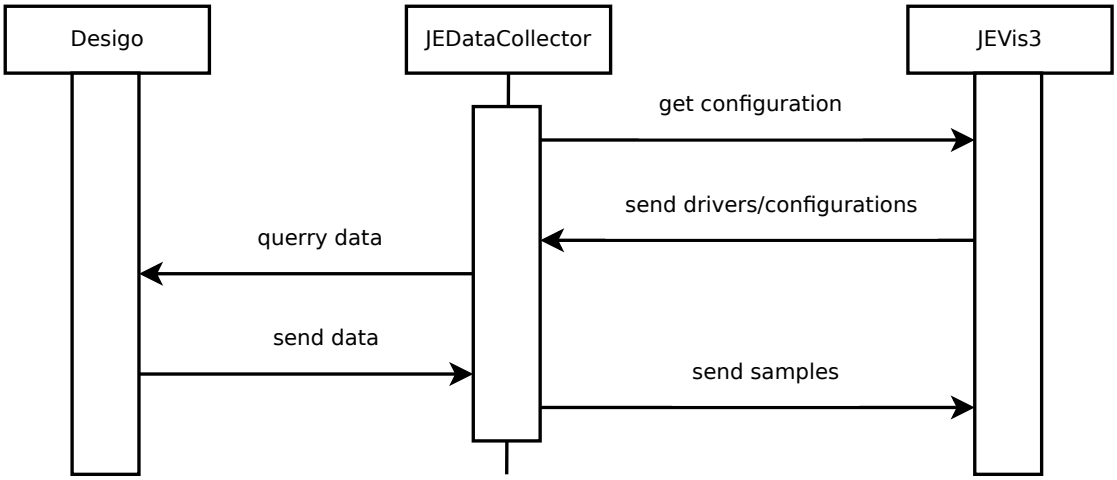


Figure B.2: JEDataCollector Message Sequence Chart

Figure B.2 shows the process of importing data from Desigo into JEVis. In the following, the steps of the process are explained:

- get configuration
  The JEDataCollector starts and connects to the JEVis system to request its configuration.

- send drivers/configurations
  JEVis responds with the needed drivers and configurations.

- query data
  Using the received configurations, JEDataCollector connects to Desigo and requests data samples.

- send data
  Desigo returns the requested data points.
- send samples
  JEDataCollector parses the received samples and sends them to JEVis to be stored.

List of Figures

1.1 Useful energy analysis in Austria [Aus] 2

2.1 Used subset of the Semantic Web Stack [Bra] 9
2.2 RDF triple showing sentence "Alice knows Bob" 11
2.3 RDF 1.1 serialization formats [SR14] 12

3.1 Classes defining the encoding of response 24
3.2 CSV structure classes 25
3.3 XML structure classes 25
3.4 JSON structure classes, a first try 25
3.5 JSON structure classes, simplified 26
3.6 Structure classes, combination 26
3.7 Ontology response classes 27
3.8 Ontology Structure Object classes 28

4.1 Overview of the system and infrastructure around it 32
4.2 JEVis3 overwiev 33
4.3 OpenStreetMaps response ontology 36
4.4 Weather Underground Geolookup ontology 38
4.5 Weather Underground historic weather data 39
4.6 JEVis Structure 41
4.7 JEVis Samples 42
4.8 Graph visualizing path from "OSM" to "City" 45

5.1 Overview over data and prediction 49
5.2 Found anomalies 50

B.1 JEConfig showing SQL DataServer Desigo 65
B.2 JEDataCollector Message Sequence Chart 67

List of Tables

2.1 Quoted Literal shorthands [BBLPC14] 14
2.2 Property paths introduces with SPARQL 1.1 [HSP13] 17

4.1 Root of the path from "OSM" to "City" 44
4.2 Path from "OSM" to "City" 45

Bibliography

[AH11] Dean Allemang and James Hendler. *Semantic web for the working ontologist: effective modeling in RDF and OWL*. Elsevier, 2011.

[AP12] T. Aihkisalo and T. Paaso. Latencies of Service Invocation and Processing of the REST and SOAP Web Service Interfaces. In *2012 IEEE Eighth World Congress on Services*, pages 100–107, June 2012.

[AST+17] S. Ahvar, G. Santos, N. Tamani, B. Istasse, I. Praça, P. E. Brun, Y. Ghamri, and N. Crespi. Ontology-based model for trusted critical site supervision in fuse-it. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, pages 313–315, March 2017.

[Aus] Statistics Austria. Final energy consumption 2013 by fuels and useful energy categories for Austria. `http://statistik.at/`. [Online; accessed 11-February-2016].

[BBLPC14] D Beckett, T Berners-Lee, E Prud'hommeaux, and G Carothers. RDF 1.1 turtle–terse rdf triple language. W3C Recommendation. *World Wide Web Consortium (Feb 2014), available at* `http://www.w3.org/TR/turtle`, 2014.

[BG14] Dan Brickley and R Guha. RDF Schema 1.1. W3C Recommendation (25 February 2014). *World Wide Web Consortium*, 2014.

[BGM04] Dan Brickley, Ramanathan V Guha, and Brian McBride. RDF vocabulary description language 1.0: RDF Schema. W3C Recommendation (2004). *URL http://www. w3. org/tr/2004/rec-rdf-schema-20040210*, 2004.

[BPSM+98] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 16:16, 1998.

[Bra] Steve Bratt. Semantic web, and other technologies to watch. W3C, 2007. `https://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#(24)`. [Online; accessed 11-March-2016].

[Bra14] Tim Bray. The javascript object notation (json) data interchange format. 2014.

[C+12] World Wide Web Consortium et al. OWL 2 web ontology language document overview. 2012.

[C+13] World Wide Web Consortium et al. SPARQL 1.1 overview. 2013.

[CBB+12] Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl García-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, Vincent Huang, Krzysztof Janowicz, W. David Kelsey, Danh Le Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin Page, Alexandre Passant, Amit Sheth, and Kerry Taylor. The ssn ontology of the w3c semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25 – 32, 2012.

[CK04] Jeremy J Carroll and Graham Klyne. Resource Description Framework (RDF): Concepts and Abstract Syntax. 2004.

[Cla97] James Clark. Comparison of SGML and XML: World Wide Web Consortium Note. *World Wide Web Consortium*, 15, 1997.

[CWL14] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 concepts and abstract syntax. *W3C Recommendation*, 25:1–8, 2014.

[DdHR15] Laura Daniele, Frank den Hartog, and Jasper Roes. *Created in Close Interaction with the Industry: The Smart Appliances REFerence (SAREF) Ontology*, pages 100–112. Springer International Publishing, Cham, 2015.

[DuC13] Bob DuCharme. *Learning SPARQL*. O'Reilly Media, Inc., 2013.

[FB96] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. `https://tools.ietf.org/html/rfc2045`, 1996. [Online; accessed 5-June-2016].

[FR14] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. `http://www.rfc-editor.org/info/rfc7231`, 2014. [Online; accessed 5-June-2016].

[FT00] Roy T. Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 407–416, New York, NY, USA, 2000. ACM.

[G+93] Thomas R Gruber et al. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[GCF16] J. Gao, B. Cao, and H. Fan. Point of interest data storage using ontology. In *2016 3rd International Conference on Systems and Informatics (ICSAI)*, pages 1122–1126, Nov 2016.

[GvR01] F.L. Drake G. van Rossum. Python Reference Manual. `http://www.python.org`, 2001. [Online; accessed 5-December-2015].

[HLZ15] C. C. Huang, A. Liu, and P. C. Zhou. Using ontology reasoning in building a simple and effective dialog system for a smart home system. In *2015 IEEE International Conference on Systems, Man, and Cybernetics*, pages 1508–1513, Oct 2015.

[HSP13] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 query language. *W3C Recommendation*, 21, 2013.

[Hun07] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[jso] Introducing JSON. `http://www.json.org/`. [Online; accessed 1-June-2016].

[Kas06] Wolfgang Kastner. Trends in der Gebäudeautomation. *(2006), 11, S. 44; AC00340518*, 2006. Aus Megatech ; 11.

[KKR12] Wolfgang Kastner, Mario Jerome Kofler, and Christian Reinisch. Knowledge representation for the adaptive residential home in the context of smart cities. *e & i Elektrotechnik und Informationstechnik*, 129(4):286–292, 2012.

[Kof14] Mario Jerome Kofler. An ontology as shared vocabulary for distributed intelligence in smart homes, 2014. Wien, Techn. Univ., Diss., 2014.

[LKGZ16] Maxime Lefrançois, Jarmo Kalaoja, Takoua Ghariani, and Antoine Zimmermann. SEAS Knowledge Model. Deliverable 2.2, ITEA2 12004 Smart Energy Aware Systems, 2016. 76 p.

[McK10] Wes McKinney. Data Structures for Statistical Computing in Python . In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[MVH+04] Deborah L McGuinness, Frank Van Harmelen, et al. OWL web ontology language overview. *W3C Recommendation*, 10(10):2004, 2004.

[NM01] Natalya F. Noy and Deborah L. Mcguinness. Ontology development 101: A guide to creating your first ontology. Technical report, 2001.

[PAG06] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In *International semantic web conference*, pages 30–43. Springer, 2006.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[PD09] Addison Phillips and Mark Davis. Tags for identifying languages. Technical report, 2009.

[PS+08] Eric Prud'Hommeaux, Andy Seaborne, et al. SPARQL query language for RDF. *W3C Recommendation*, 15, 2008.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, October 2011.

[req11] Requests: HTTP for Humans. `http://docs.python-requests.org/`, 2011. [Online; accessed 5-December-2015].

[Sch14] Daniel Schachinger. Model-driven engineering for building automation systems, 2014. Wien, Techn. Univ., Dipl.-Arb., 2014.

[Sha05] Yakov Shafranovich. Common format and MIME type for Comma-Separated Values (CSV) files. 2005.

[SR14] Guus Schreiber and Yves Raimond. RDF 1.1 Primer. *W3C Working Group Note*, 2014.

[Sta13] Paul Staroch. A weather ontology for predictive control in smart homes, 2013. Parallelt. [Übers. des Autors] A Weather Ontology for Predictive Control in Smart Homes; Wien, Techn. Univ., Dipl.-Arb., 2013.

[Ste14] Simon Steyskal. Defining an actor ontology for increasing energy efficiency and user comfort in smart homes, 2014. Parallelt. [Übers. des Autors] Defining an Actor Ontology for Increasing Energy Efficiency in Smart Homes; Wien, Techn. Univ., Dipl.-Arb., 2015.

[Str] Tom Strassner. XML vs JSON. `http://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html`. [Online; accessed 1-June-2016].

[WCV11] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[Woo14] David Wood. What's New in RDF 1.1. *W3C Working Group Note*, 2014.