University of Zurich^UZH

Thesis August 1, 2018

Publication of linked data streams on the Web

Elias Bernhaut
of Zürich ZH, Switzerland

Student-ID: 14-735-773
elias.bernhaut@uzh.ch

Advisor: **Daniele Dell'Aglio**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I thank Dr. Daniele Dell'Aglio for his support throughout the process of the thesis. I am glad that he had an open ear and helped me with his guidance. Further, I thank Natálie Libosková for her help, review and caring words and finally I also thank Raffael Kallis for his support.

Zusammenfassung

Der Zugang zu Informationen ist ein wichtiger Faktor für fundierte Entscheidungen, sei es im Rahmen von Abstimmungen oder auch Geschäftsentscheidungen. Open Data ist der Baustein für die Veröffentlichung von Informationen, einschliesslich Datensätzen, die von Regierungen veröffentlicht werden. Open Government Data (OGD) ermöglicht Bürgern den Zugang zu Informationen und bildet die Grundlage für neue Applikationen, welche Regierungsdaten als Datenquelle nutzen.

Parallel zur Open Data Bewegung erweitert das Semantic Web den globalen Open Linked Data Graphen zum Aufbau der Linked Open Data Cloud (LOV). Linked Data bietet den Vorteil global identifizierbarer Einheiten, die durch Beziehungen miteinander verbunden sind. Anwendungen mit Zugang zum Web können auf den Graphen der LOV zugreifen, um Informationen aus verschiedenen Quellen abzurufen.

TripleWave ist ein Framework für die Veröffentlichung von Linked Data als Datenströme im Gegensatz zu statischen Datensätzen. Es kann gestreamte Eingangsdaten miteinander verknüpfen und so in global verknüpfte Datenströme umwandeln.

Da TripleWave gestreamte Eingangsdaten transformieren kann, fehlt die Möglichkeit, statische Datensätze, die häufig aktualisiert werden, in verknüpfte Datenströme zu transformieren. In dieser Arbeit analysiere ich die Anforderungen an eine tragfähige Lösung und wähle eine Mapping-Sprache mit Hilfe der Untersuchung von OGD-Datensätzen, welche die Grundlage für die Auswertung bilden. Ich zeige, dass keine vorhandene Mapping-Sprache die Anforderungen erfüllt. Eine neue Mapping-Sprache ist daher notwendig und so stelle ich ein neues, an der Mapping-Sprache RML orientiertes Mapping-Modul namens JRML vor. JRML ist ein Javascript-Modul für Datenmappings zu Linked Data mit einer integrierten, Pull-basierten Datenabrufstrategie, die von einem Scheduler gesteuert wird. Ich zeige, wie JRML die Anforderungen für die Transformation der OGD-Datensätze der Umfrage erfüllt, wie ich JRML implementiere und in TripleWave integriere. Schliesslich veröffentliche ich eine Reihe von transformierten Datensätzen, um das Ergebnis zu präsentieren und damit die Anzahl der Linked Open Data Streams im Web zu erhöhen.

Abstract

The access to information is an important factor for making informed choices, for example in the context of votings or business decisions. Free information is published on the Web as Open Data including free datasets published by governments. The so called Open Government Data (OGD) empowers the citizens with the access to information and builds the basis for new applications using government data as datasource.

Parallel to the Open Data movement, the Semantic Web expands the global graph of Open Linked Data building the Linked Open Data Cloud (LOV). Linked Data comes with the advantage of globally identifiable entities interlinked through relationships. Applications with access to the Web are able to access the graph of the LOV for the retrieval of information from various sources.

Linked Open Data datasets are majorly published as static datasets which means they don't change over time in contrast to dynamic datasets. An ongoing research has the goal to publish dynamic datasets as data streams. A notable example for a framework approaching the publication of Linked Data as data streams is TripleWave. It can interlink streamed input data and thus transform it to Linked Data streams.

As TripleWave can transform streamed input data, it is missing the ability to transform static datasets which are updated frequently into Linked Data streams. Throughout this thesis, I analyse the OGD datasets and identify the requirements for their publication as Linked Open Data streams which are not yet covered by TripleWave. I show that no existing mapping language fulfills the requirements for the publication of the OGD datasets. A new mapping language is therefore necessary and thus I introduce a new, RML oriented mapping module named JRML. JRML is a Javascript module for data mappings to Linked Data with an integrated, pull-based data-fetching strategy controlled by a scheduler. I show how JRML meets the requirements for transforming the OGD datasets of the survey, how I implement JRML and how I integrate it into TripleWave. Finally I publish a range of transformed datasets to present the result and thus increase the number of Linked Open Data streams on the web.

Table of Contents

1 Introduction

In the age of information, the access to key information and knowledge is more important than ever. The Open Knowledge International (OKI) mission is to make *open knowledge* a mainstream concept. The vision of OKI is to bring knowledge to the people which brings power to the majority opposed to the minority. According to OKI, the access to data empowers the people to make free choices about life, purchasing and voting.[1]

Following the intent of OKI, governments publish datasets as licensed open data on several platforms including the Open Data Platform of Zürich[2] the Open Data platform of Austria[3] and the Open Data platform of Germany.[4]

Bizer et al. (2009) present the concept of Linked Data. Linked Data is data which is interlinked with other data, building the the Linked Open Data Cloud (LOV) – a global graph of interlinked datasets containing more than a thousand datasets including data from OGD portals. The usage of URIs as identifiers in Linked Data makes it possible to link data on a global level. Open Government Data (OGD) can greatly improve its value and improve interoperability through the presentation of the data as Linked Data. Using the Resource Description Framework (RDF) is the preferred format for publishing Linked Data (Bizer et al., 2009). Publishing OGD data in RDF format allows clients to query linked data accross multiple datasets on the Web.

An upcoming research interest is targeted to Linked Data publications as data streams on the Web. Data streams represent an endless flow of information and thus, data streams are used when the data is of continuous nature. Barbieri and Valle (2010) propose a concept for the publication of Linked Data streams. Mauri et al. (2016) provide TripleWave, a framework which achieves the publication of Linked Data data streams. TripleWave has the ability to map input data streams to RDF streams or replay a series of RDF graphs with time annotations to build an RDF stream. TripleWave outputs RDF stream can be accessed through pull- and push-based strategies.

Up to now, there is no solution integrated to TripleWave for publishing as RDF streams Web datasets which are frequently updated, e.g., continuously updating OGD datasets with time annotations. Muntwyler (2017) provides guidelines for the publication of open data through the usage of TripleWave. The presented solution relies on external data

[1] *https://okfn.org/about/* (accessed 13.07.2018)
[2] *https://data.stadt-zuerich.ch* (accessed 13.07.2018)
[3] *https://www.data.gv.at* (accessed 13.07.2018)
[4] *https://www.govdata.de* (accessed 13.07.2018)

retrieval scripts that fed data to Apache Kafka[5] with TripleWave as a transform component reading data from Kafka and supplying the transformed data back to Kafka through connector scripts. This approach consists of a lot of components working together.

To increase the number of Linked Open Data streams on the Web, a dedicated solution is needed for the publication of frequently updating datasets. The publication of Linked Open Data streams includes the retrieval of Open Data from the Web and the mapping of the retrieved data to RDF. The goal is a self-contained solution that is preferably integrated into TripleWave.

The contribution of this thesis is the creation of a streamlined solution for the publication of frequently updating datasets through an extension to TripleWave. Through the addition of a datasource to TripleWave, I integrate the data retrieval and mapping to RDF directly into TripleWave through a newly invented component named JRML. JRML is a data mapping language and processor based on RML (Dimou et al., 2014) which is already used in TripleWave for input stream to RDF stream mappings. JRML extends RML and ports it to a Javascript environment with slight changes to the structure of the mapping declarations.

JRML is able to retrieve the datasets from the Web or locally. It maps them with advanced transformations through which mapping structurally challenging data becomes possible. The whole power of Javascript and its ecosystem is available to be used for transformations within the mapping definitions.

In this thesis, I investigate and analyse OGD datasets from the Austrian and German Open data platforms to extend the list of suitable OGD datasets for the publication as RDF streams created initially by Muntwyler (2017) in Section 3.3. I then analyse how TripleWave must be changed to integrate better into the Javascript environment and to ease the deployment in Section 3.4. In Section 5.1, I evaluate the mapping language I am going to use through an analysis of the requirements with respect to the obtained OGD datasets. I show that no current mapping solution fulfills the requirements and that a new solution must be found. As solution, I propose JRML in Section 5.2, a mapping language based on RML (Dimou et al., 2014) but ported to Javascript. I explain step by step what adaptions and changes I apply to fulfill the requirements. I illustrate the architecture and design desicions of the implementation of JRML and how I deployed a range of five OGD datasets with their specific challenges concerning the mapping (see Chapter 6) before I come to the conclusion and future work in Section 7.

[5] *http://kafka.apache.org* (accessed 13.07.2018)

2 Related Work

In this chapter, I present related works to provide the necessary knowledge base. In the first Section, I explain the term Open Data and show that governments make their data openly available as Open Data. In the following section I talk about Linked Data, the Linked Data Cloud (LOD) and the 5 stars rating for data presentation. Finally, I show approaches to the publication of streams that transport Linked Data.

2.1 Open Data

Data is today a widely common term, for this thesis however, it is necessary to grasp the term *open data*. To explain, it is necessary to focus on the *open* part of the term. Open can have various meanings in different domains. A general definition of openness in the domain of data is given by the Open Knowledge International organization: "Open data and content can be freely used, modified, and shared by anyone for any purpose". In other words, open data refers to a domain of computer science where data is shared across the World Wide Web and stored at various places.[1]

To specify what data on the Web is allowed to be used for, the data has to be provided under a license which formalizes the constraints. There is a wide range of licenses which are applicable to data. Only a few of them are *open data licenses* which ensure the data can be defined as open. A list of licenses which are compatible with open data as defined by the Open Knowledge International organization,[2] as well as the ruleset these licenses have to follow,[3] can be found on *https://opendefinition.org/*.

Open data licenses find application in various contexts. One context in which they are used is related to government data and is usually referred to as Open Government Data (OGD). To explain the need for OGD, the Open Knowledge Foundation refers to the need of citizens to have information about the processes of their government. Therefore, the citizens have to be granted free access to government data and have to be able to share this information with other citizens. In addition, the government can help to create businesses and services by making their data openly accessible. Another principle

[1] *https://opendefinition.org/* (accessed 31.05.2018)
[2] *https://opendefinition.org/licenses/* (accessed 31.05.2018)
[3] *https://opendefinition.org/od/2.1/en* (accessed 31.05.2018)

supporting opening up government data is a better contribution to the decision-making process of the government by being better informed through OGD.[4]

2.2 Linked Data

Linked data, according to Bizer et al. (2009), refers to machine-readable data published on the Web. Linked Data can define links to external datasets which in turn can link back to the same data and/or other external datasets. In the Web, HTML documents are connected through hyperlinks. Linked Data on the other hand makes use of the Resource Description Framework (RDF) format to make statements about the world and establish links (Bizer et al., 2009). This results in what is commonly known as the Web of Data.

Berners-Lee (2006) is concerned with interlinking data on a global level. To bring all data into a global namespace, he proposes a set of four rules for publishing data on the Web.

According to Bizer et al. (2009), these rules have become known as the 'Linked Data Principles' and are now standard ingredients for publishing connected data on the Web.

The rules include that URIs should be used to name things. Bizer et al. (2009) state that URIs are a more generic way to identify real world entities. Commonly known are URLs which are used to address documents and entities located on the Web. Entities identified by a URI can be looked up at the location the URI points to by simply using HTTP to access the resource.

The four rules for publishing data on the Web by Berners-Lee (2006) are:

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

Linked Data is linked together through the URIs of the entities it describes. RDF is used as the data description language and it encodes data in the form of triples. Each triple consists of three parts: a subject, a predicate and an object where the subject and the object are either URIs or a URI and a string literal. The predicate defines the relation between the subject and the object.

Two examples of triples given by Bizer et al. (2009) are:

[4] *https://opengovernmentdata.org* (accessed 01.06.2018)

subject	*http://dig.csail.mit.edu/data#DIG*
predicate	*http://xmlns.com/foaf/0.1/member*
object	*http://www.w3.org/People/Berners-Lee/card#i*
subject	*http://data.linkedmdb.org/resource/film/77*
predicate	*http://www.w3.org/2002/07/owl#sameAs*
object	*http://dbpedia.org/resource/Pulp_Fiction_%28film%29*

Figure 2.1: Example RDF triples (Bizer et al., 2009)

For describing real world entities through RDF and URIs, vocabularies are needed for expressing relationships and identities. Vocabularies are written in RDF as well, usually with the RDF Vocabulary Definition Language (RDFS) (Brickley, 2004) or the Web Ontology Language (OWL) (McGuinness et al., 2004). It is a common practice for data publishers to reuse existing vocabularies instead of creating a new one from scratch every time. But it is no easy task to find the right vocabularies and definitions that fit the data. Vandenbussche et al. (2017) present the Linked Open Vocabularies (LOV), a dataset containing various available RDFS and OWL vocabularies. It is hosted by the Open Knowledge Foundation and allows to search for relevant vocabularies. Generally, well-known vocabularies such as FOAF, Dublin Core, SIOC, SKOS, vCard, DOAP, OAI-ORE or GoodRelations should be preferred over unknown vocabularies. However, the choice of vocabularies is open and various vocabularies can be used in parallel (Bizer et al., 2009).

Because RDF is a datamodel it requires a syntax to be serialized. There are several syntax options for RDF. The most common syntaxes are: RDF/XML, Notation3, its subset Turtle.

Open data published on the Web as linked data is called Linked Open Data (LOD) and the set of LOD sets composes the LOD cloud.[5] The LOD cloud can be seen as the whole linked open data that is published. Together, the data builds a large graph of linked data located in multiple datasets which themselves have links to other datasets.

The creator of the Web, Tim Berners-Lee, presents a scale of stars from 1 to 5 to rate how powerful and easy to use the data in question is (Berners-Lee, 2006). The stars depend on each other, meaning that three star data needs to fulfill the criteria that two star data has to fulfill plus the additional criteria for the third star.

Here I list the stars together with their meanings. Figure 2.2 illustrates the five stars in a visual manner in the form of stairs.

- 1: The first star is attributed to the mere publication of the data on the Web under an open license. The format of the data does not matter for this star.
- 2: For applying the second star, the data has to be published as structured data which enables processing the data through proprietary software for as example calculations and visualizations.

[5] *http://lod-cloud.net* (accessed 01.06.2018)

- 3: The third star can be applied if the data is published in a non-proprietary format, for example CSV instead of XLS. This enables processing the data with free or custom software in any required way.
- 4: The forth star can only be applied if URIs are used to identify things in given data. This lets others to point to the given data items and link their own data to it or bookmark it.
- 5: The last star can be applied when linking data to other data for forming a context. It allows the discovery of more related data and the exploration of the data by learning from the schema.

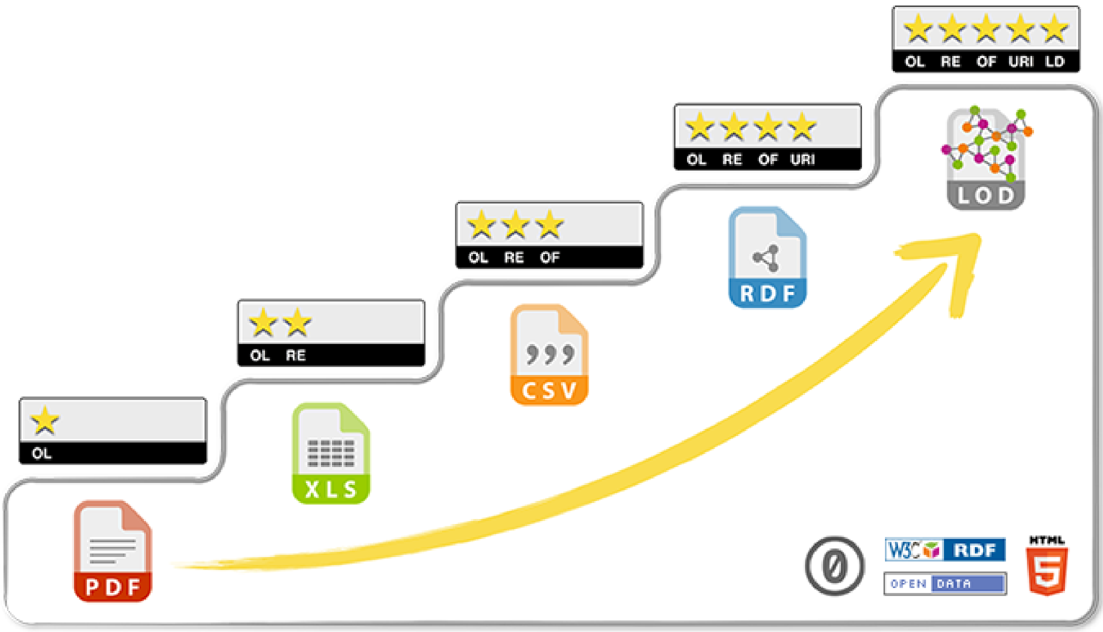


Figure 2.2: The 5 stars of Linked Open Data (source: http://5stardata.info)

2.3 Linked Data mappings for existing datasets

Datasets such as OGD sets are rarely deployed in formats obtaining more than three stars. The reason for this can be found in the additional effort and the convenience of the publication. As a result, most data is available in a structured format that can be processed by a software, such as data stored in CSV, XML, JSON format or in databases.

Because most of the existing data is stored in databases, the World Wide Web Consortium (W3C) created the recommendation for the RDB to RDF Mapping Language (R2RML). The R2RML mapping language describes mappings from relational databases to RDF datasets (Das et al., 2012).

Another mapping language targeting relational databases is D2RQ (Bizer and Seaborne, 2004) which allows querying relational databases as RDF graph through Sparql and the publication of the content of a relational database on the Semantic Web. D2RQ is similar to R2RML because both allow mapping relational database data to RDF. However, it goes one step further because it treats the database itself as an RDF graph.

The D2RQ format, as well as R2RML, only target relational databases and therefore do not define a general mapping solution. More per-format mappings include the usage of XML tools like XSLT, XPath and XQuery (Dimou et al., 2014). None of these solutions provide mappings for multiple source formats.

A more generic solution which allows mappings from multiple source formats to RDF is presented by Dimou et al. (2014). The RDF Mapping Language (RML) is a superset of R2RML. Every R2RML mapping is a valid RML mapping. On top of the R2RML specification, RML specifies additional properties. For example properties for defining the query language used in the predicate-object mappings and for defining the iteration. R2RML and CSV/spreadsheet-to-RDF mapping solutions assumed each row of the dataset is one entity Dimou et al. (2014). With RML, there is now also a way to cross-link data from multiple sources and throughout multiple formats. I.e. in one RML file definition, multiple sources can be defined and mapped. At the same time the mappings define links across the sources (Dimou et al., 2014).

2.4 Stream Publishing

The LOD cloud and most linked data published on the Web provide and describe static data. Streaming data had been neglected and not given much attention (Barbieri and Valle, 2010).

According to Barbieri and Valle's (2010) definition, a datastream is an ordered sequence of pairs where each pair consists of a tuple with its timestamp.

Barbieri and Valle (2010) propose an approach for publishing data streams as linked data. With the development of the C-SPARQL engine as an extension to SPARQL, they provide a way to run continuous queries on streaming data. C-SPARQL can process data streams and RDF streams together with RDF graphs. To represent RDF streams, Barbieri and Valle (2010) propose to use named graphs (Sequeda and Corcho, 2009). Sequeda and Corcho (2009) distinguish between (s-graphs) and Instantanious Graphs (i-graphs). A stream is represented as an s-graph and several i-graphs, one for each timestamp (Barbieri and Valle, 2010). An s-graph contains metadata about the stream content of the current window over the RDF stream. C-SPARQL also supports sliding windows allowing the extraction of the last data elements of the stream.

C-SPARQL enables the consumption and querying of streaming data. However, it does not provide a way to publish streams. Mauri et al. (2016) close this gap and propose TripleWave as a reusable and generic tool for the publication of RDF streams. TripleWave serves as a RDF stream producer and data format transformer. It can consume non-RDF streams from the Web, as well as static, time annotated datasets, transform the data to the RDF format and publish it as RDF streams. It thus supports pull-based and push-based consumption and is able to replay time annotated data, possibly in an endless loop. The RDF stream published by TripleWave publishes its RDF data items in the JSON-LD format[6] and distinguishes between the s-graphs and i-graphs. For the configuration of the data transformation to RDF, Mauri et al. (2016) makes use of RML (see 2.3).

TripleWave is part of a set of technologies named RDF Stream Processing (RSP) and takes its place as an RDF stream publishing service for providing data for stream reasoners (Dell'Aglio et al., 2017). Dell'Aglio et al. (2017) present with their proposal for WeSP, a framework for the exchange of RDF streams, an approach for an improvement and realization of an eco-system of stream engines on the Web.

Sedira et al. (2017) present a Vocabulary of Interlinked Datastreams (VoIS) for describing stream descovery, access, recall and provenance. It allows stream consumers to find relevant streams, accessing them and reason about origin data origin.

Taelman et al. (2016) demonstrate the mapping and publishing of continuous sensor data through RML and NodeJS. They use tessel microcontrollers[7] to obtain sensor data which is mapped by RML and published as RDF stream.

[6] *https://json-ld.org* (accessed 01.06.2018)
[7] *https://tessel.io* (accessed 23.07.2018)

3

Problem analysis

In this chapter I show the current state of TripleWave, its problems and necessary improvements for the publication of OGD datasets. I show the need for a configuration refactoring and a new implementation of the RML mapping which can handle the various forms of publishing OGD data. A survey on OGD datasets provides the basis for the requirements of the implementation which I define in Section 3.4.

3.1 The current state of TripleWave

TripleWave was introduced by Mauri et al. (2016) for the creation of RDF streams and their publication on the Web. In its current state, TripleWave accepts inputs from either a Webstream through a custom connector or from a local file holding RDF data which TripleWave can stream. Currently supported output protocols are WebSocket and MQTT.

Connectors which create an input stream for TripleWave have to be CommonJS[1] modules which NodeJS uses. The expected module export is a NodeJS transform stream. The origin of the streamed data does not matter to TripleWave. In case of OGD datasets, the data is usually retrieved directly from the OGD platforms. In other cases it is generated, stored locally or obtained from databases. The data flowing into TripleWave has to be a in Javascript Object Notation (JSON). A flat JSON object is an object in Javascript Object Notation which has not more than one level of properties and therefore no nested objects or arrays. The connector alone does not suffice for TripleWave to generate a corresponding RDF output. An R2RML mapping file (see Section 2.4) is required in addition to the connector and defines the mapping from the input data to RDF.

The current solution presents problems. The transformation process of TripleWave based on R2RML is a small subset of both: RML and R2RML and neither of those specifications are fully implemented. The mapping directly transforms the input data to the JSON-LD RDF format where the represented triples are directly added to a graph with the identity of the subject given by the mapping. The specification of R2RML (Das et al., 2012) states that if neither a graph nor graphMap is specified, the graph to which

[1] *http://www.commonjs.org/* (accessed 29.07.2018)

the triples are added has to be the default graph. This means TripleWave uses its own interpretation of the mapping which can be confusing for users. In addition, there is no streamlined way to retrieve common datatypes/sources from the Web and feed them into TripleWave. Programming a new connector each time a datasource has to be added is cumbersome and induces a lot of duplication. The connector has to transform the retrieved data to fit the flat JSON structure needed by TripleWave. The configuration of TripleWave is done in *.properties* files which are used mainly in the programming language Java for configurations. While this works, *.properties* configurations are no standard in the Javascript environment. Configurations in Javascript are commonly done in JSON notation either in a *.json* file or as a Javascript object exported from a NodeJS module.

3.2 Approaching a consensus for publication

As mentioned in Section 3.1, a missing feature of TripleWave is to this day an expressive way to point to a datasource/set somewhere on the Web and let TripleWave transform it into an RDF stream.

There are many applications for which this feature is useful. It can for example example be used internally by companies to transform their own datasets to RDF or it can be used to transform openly available datasets on the Web to an RDF stream, provided the datasets contain time annotations or are updated frequently.

TripleWave was built on a set of requirements. Based on a set of analysed datasets which were not considered before, new requirements emerge, which are not covered by the current features of TripleWave.

The following section lists a set of OGD datasets obtained from German OGD platforms. OGD datasets are freely available and follow the Open Data rules, see 2.1. In Section 3.4 I will obtain the requirements for a solution for the transformations explained above.

3.3 A survey on OGD datasets

This section contains a survey on dynamic OGD datasets which can be potentially published as RDF streams. The task of TripleWave is to publish data as RDF streams. OGD provides a lot of datasets that can be freely used and transformed by TripleWave into RDF annotated streams. As the goal of this thesis is the publication of more datasets and the extension of TripleWave to support the publication of frequently updating datasets on the Web, a list of representative datasets is necessary.

3.3.1 Procedure

Muntwyler (2017) analysed a wide range of Swiss datasets. I extend the list in this survey, focusing on the German and Austrian portals.

Streaming data is data connected with specific points in time and is therefore annotated with timestamps. When a data stream should be created out of static data from OGD portals, there is a need for data that has time annotations and is updated frequently. If no time annotation is given and the dataset updates can be considered to happen in real-time, the time of the publication can be used as time annotation.

To narrow down the search through these requirements on the datasets, I used the following keywords for the search:

- RSS
- real time
- *Echtzeit*
- *Messwerte*
- *Messdaten*
- *Parkplätze*
- *aktuell*

Most keywords are German because the datasets are in German. Only "real time" is English, because it is a phrase often used in German instead of *Echtzeit* when talking about real time data. RSS feeds provide datafeeds which act in a way like data streams. Searching for RSS feeds reveals frequently updated dataset and datasets also available in other formats. *Messwerte/daten* are measurements. Measurements are a well-known category of streaming data and are usually supplied with time annotations of the time of the measurement. *Aktuell* means latest or current in english and is a keyword targetting data that is kept up-to-date. *Parkplätze* are parking lots. This search is very specific but the thesis of Muntwyler (2017) has shown that datasets providing data about the availability of parking lots were well fitting as they provide information about what parking lots are available at the moment and are updated frequently. The aim of this keyword is to find other similar datasets.

3.3.2 Results

There are two large Swiss OGD portals - one for the entire Switzerland and one for the city of Zürich.

- *https://opendata.swiss* The dataportal for Open Data of the Swiss Government
- *https://data.stadt-zuerich.ch* The dataportal for Open Data of the city of Zürich

Querying these portals with the selected keywords revealed nothing but the same datasets already found by Muntwyler (2017). Therefore, other German speaking countries are taken into consideration. Namely Austria and Germany.

The Austrian OGD portal is *https://www.data.gv.at*. Querying this portal resulted in 12 new datasets:

- LWAU: Luftmessnetz: aktuelle Messdaten Wien
- AKAU: Abflusswerte Kärnten
- KSAU: Kulturserver Graz RSS feeds
- BSAU: Baustellen
- FAU: Feinstaub PM10
- GAU: Globalstrahlung
- HMAU: Hydrologische Messdaten
- MZAU: Meteorologische Messdaten der ZAMG
- WTAU: Wetterstation Tirol
- NTAU: Niederschlagsdaten Stationen Hydrologischer Dienst Tirol
- WWAU: Wartezeiten in den Magistratischen Bezirksämtern Wien
- AOAU: Aktuelle Ozondaten Österreich

Doing the same on the German portal *https://www.govdata.de* resulted in a smaller range of datasets:

- PKDE: Stadt Kleve: Parkleitsystem Stadt Kleve
- FPDE: Freie Pegeldaten über PEGELONLINE
- RDE: Radmonitore

3.3.3 Obtained datasets

The datasets obtained from the sources above are listed in this section with their relevant metadata.

The *Title* represents the name of the dataset as defined on the OGD platform which provides the dataset.

The *Short Name* is a short abstraction of the name connected with the abbreviation of the country that provides the dataset, which is also given in the row *Country*. In this thesis, I will use the *Short Name* to refer to a dataset.

The *Provider* refers to the organisation that provides the data. Each dataset is published with a *license* that dictates the terms of use. The license is listed with its short name. For example, the full name of the license *CC BY 3.0 AT* is *Creative Commons Attribution Namensnennung 3.0 Österreich*.

The *Link* points to the dataset on the OGD platform and not directly to the data because some datasets are spread over multiple files/have multiple representations. More metadata can be found on the website of the dataset description.

The metadata *Real-Time* shows whether the dataset is considered to be real-time according to the metadata given by the provider. Datasets are considered to be real-time if they are updated continuously.

The *Publication-Frequency* describes how often a new document is published and *Update Frequency* how often the given document is updated with new data. These measurements are not given by all dataset providers. Therefore, the listed measurements are obtained by the inspection of the datasets if no publication/update frequencies are given by the provider. If a new document is published for each update, the update and the publication frequencies are the same. Some datasets show that they are updated continuously, but are not real-time. An example is the dataset LWAU. The provider claims that the data is updated continuously, but the dataset gets updated only every 30 minutes. I do therefore not list this dataset as real-time or as updated continuously.

The *Precition of Time Stamp* defines the granularity of the timestamp the dataset provides.

The *Number of Records* shows how many records are defined in the resource. Where the number of records could not be obtained, I show *N/A* for *Not Available*.

The *Representation* stands for the dataformat in which the data comes. Some datasets are available in multiple formats, such as FPDE. GeoRSS is RSS but follows an additional standard for the description of geodata.[2]

The *Number of Stars* stands for the number of stars of the 5 stars of Linked Open Data (see Section 2.2).

The *Change strategy* stands for the update pattern applied by the publishers. The update pattern can be either static or mixed. If the update pattern is mixed, the file-fill strategy and the file-exchange strategy are listed with the pattern: *mixed/{file-fill}/{file-exchange}*. See a description of the possible strategies below:

1. Static - Each event (whole dataset at a certain time) provides new entries with a common timestamp

2. Mixed - One event holds entries with multiple timestamps. For this variant there are all combinations of the following two possible strategies for this variant:

   The file-fill strategy:

   a) Added - New entries are added to the existing entries.

   b) Fillup - The entries for new data already exist but are not yet filled in.

   The file-exchange strategy:

   a) Everlasting - The same file is used for all entries and is never exchanged. Items are only added.

   b) Exchangable - The file gets replaced once in a while (daily, monthly etc.).

The *Comment* contains additional comments for datasets which are in some way special or unique.

[2] *http://www.georss.org* (accessed 25.06.2018)

Title	**Luftmessnetz: aktuelle Messdaten Wien**	**Abflusswerte Kärnten**	**Kulturserver_ Graz_RSS_Feeds**	**Baustellen**
Short Name	LWAU	AKAU	KSAU	BSAU
Country	Austria	Austria	Austria	Austria
Provider	Stadt Wien	Land Kärnten	Stadt Graz	Land Oberösterreich
License	CC BY 3.0 AT	CC BY 3.0 AT	CC BY 3.0 AT	CC BY 3.0 AT
Link	*https://www.data.gv.at/katalog/dataset/d9ae1245-158e-4d79-86a4-2d9b3defbedc*	*https://www.data.gv.at/katalog/dataset/5459cb0a-8cc3-4056-8717-c60febeafded*	*https://www.data.gv.at/katalog/dataset/f36e3404-c711-4f35-8312-757d6a6691f2*	*https://www.data.gv.at/katalog/dataset/e3ad9e60-9a48-40fd-a5de-ed3351c9222d*
Real-Time	no	yes	no	yes
Publication Frequency	every 30 minutes	continuously	daily	continuously
Update Frequency	every 30 minutes	continuously	daily	continuously
Precision of Time Stamp	second	second	second	second
Number of Records	18	N/A	N/A	N/A
Representation	CSV	GeoRSS	RSS	GeoRSS
Number of Stars	3	3	3	3
Change Strategy	static	mixed/added/ exchangable	static	static
Comment			Multiple RSS endpoints available	

Table 3.1: Obtained datasets with their metadata. Part 1

Title	**Feinstaub PM10**	**Globalstrahlung**	**Hydrographische Messdaten**	**Meteorologische Messdaten der ZAMG**
Short Name	FAU	GAU	HMAU	MZAU
Country	Austria	Austria	Austria	Austria
Provider	Land Niederösterreich	Land Niederösterreich	Land Salzburg	ZAMG
License	CC BY 3.0 AT	CC BY 3.0 AT	CC BY 3.0 AT	CC BY 3.0 AT
Link	*https://www.data.gv.at/katalog/dataset/8b057f32-1312-40ae-ae51-9aa0a0d372ca*	*https://www.data.gv.at/katalog/dataset/f9e40f30-8ac6-43e2-9ee7-f72b712ea9e1*	*https://www.data.gv.at/katalog/dataset/3c589dc0-35ed-48cf-b598-428e4588b19c*	*https://www.data.gv.at/katalog/dataset/9b40a0af-a6fe-47ff-9624-2ea8f40c746f*
Real-Time	no	no	no	no
Publication Frequency	hourly	hourly	every 15 minutes	every hour
Update Frequency	hourly	hourly	every 15 minutes	every hour
Precision of Time Stamp	day	day	minute	hour
Number of Records	21	7	988	10
Representation	CSV	CSV	TXT	CSV
Number of Stars	3	3	3	3
Change Strategy	mixed/fillup/exchangable	mixed/fillup/exchangable	mixed/fillup/exchangable	static
Comment	New file each day. Placeholders for the values for each hour	New file each day. Placeholders for the values for each hour		

Table 3.2: Obtained datasets with their metadata. Part 2

Title	**Wetterstations-daten Tirol**	**Niederschlags-daten Stationen Hydrographischer Dienst Tirol**	**Wartezeiten in den Magistratischen Bezirksämtern Wien**	**Aktuelle Ozondaten Österreich**
Short Name	WTAU	NTAU	WWAU	AOAU
Country	Austria	Austria	Austria	Austria
Provider	Land Tirol	Land Tirol	Stadt Wien	Umweltbundesamt GmbH
License	CC BY 3.0 AT	CC BY 3.0 AT	CC BY 3.0 AT	CC BY 3.0 AT
Link	*https://www.data.gv.at/katalog/dataset/bb43170b-30fb-48aa-893f-51c60d27056f*	*https://www.data.gv.at/katalog/dataset/44720e90-c2de-497b-8162-3810206dd011*	*https://www.data.gv.at/katalog/dataset/e38cdef5-f993-4e6f-919e-ac68d26d727d*	*https://www.data.gv.at/katalog/dataset/8b3b3cdf-2be6-4f0b-8c86-f6be67e5b002*
Real-Time	no	yes	yes	yes
Publication Frequency	every hour	continuously	continuously	continuously
Update Frequency	every hour	continuously	continuously	continuously
Precision of Time Stamp	second	second	minute	second
Number of Records	N/A	1-10K	1	108
Representation	JSON	CSV	JSON	JSON
Number of Stars	3	3	3	3
Change Strategy	static	mixed/added/exchangable	static	static
Comment				

Table 3.3: Obtained datasets with their metadata. Part 3

Title	**Stadt Kleve: Parkleitsystem Stadt Kleve**	**Freie Pegeldaten über PEGELONLINE**	**Radmonitore**	**Parkplätze API (Beta)**
Short Name	PKDE	FPDE	RDE	PADE
Country	Germany	Germany	Germany	Germany
Provider	Stadt Kleve	ITZB Bund	Universitätsstadt Rostock	DB BahnPark GmbH
License	Datenlizenz Deutschland - Zero - Version 2.0	CCBY	CC0	CC BY 4.0
Link	*https://www.govdata.de/web/guest/suchen/-/details/parkleitsystem-stadt-kleve-odp*	*https://www.govdata.de/web/guest/suchen/-/details/pegelonline*	*https://www.govdata.de/web/guest/suchen/-/details/radmonitore*	*https://mcloud.de/web/guest/suche/-/results/detail/parkpltzeapibeta*
Real-Time	yes	yes	no	yes
Publication Frequency	continuously	continuously	daily	continuously
Update Frequency	continuously	continuously	every 15 minutes	continuously
Precision of Time Stamp	second	minute	minute	second
Number of Records	10	N/A	>1M	N/A
Representation	XML	JSON/WSDL/CSV/TXT/WMS/WFS/SOS	JSON/CSV/XLSX	JSON (REST)
Number of Stars	3	3	3	3
Change Strategy	static	static	mixed/added/everlasting	static
Comment		Many different ways to get the data. The current data can be retrieved individually for different stations	Data is available from 2013 to now. All data is in one huge file	Data presented through a REST API

Table 3.4: Obtained datasets with their metadata. Part 4

3.4 Requirements

Publishing the datasets (see Section 3.3) with the current version of TripleWave includes the usage of TripleWave-external scripts for pulling the dataset from the Web, parsing the data and pass it to TripleWave through a data stream. A new way to define datasources and transformations is needed which improves the deployment process of frequently updating datasets.

In this section, I analyse the requirements for an appropriate transformation solution. The wording used in the requirement listings follows the Best Current Practices for the Internet Community defined at *https://www.ietf.org/rfc/rfc2119.txt* (accessed 01.07.2018).

3.4.1 Configuration of TripleWave

The configuration of TripleWave through *.properties* files follows the default for Java applications. In the Javascript community and therefore also the NodeJS community, the accepted default for configurations are *.json* and *.js* files. The current configuration of TripleWave does not fit into the Javascript environment (see Section 3.1).

The explicit requirements for the configuration change are:

- **RC1: The configuration should follow the Javascript conventions.** TripleWave should use the accepted community default for configuring Javascript applications.
- **RC2: The new configuration system must preserve the configuration possibilities.** TripleWave should not loose functionality and dynamics.
- **RC3: The configuration property names should keep their semantics.** Users of TripleWave should not get confused by major property name changes.
- **RC4: The configuration may be extended to enable multiple runs with different configurations.** For the parallel run of multiple TripleWave instances (see Requirement RS4), a configuration which can configure the single instances is necessary.

3.4.2 Startup of TripleWave

TripleWave can be started by running the command line script *start.sh* or *start.bat*. Even though this works fine for starting up TripleWave, the command line script has to be in the current working directory or on the path and the script has to be in the same directory as the source files. Up till now, TripleWave cannot be installed through *npm*.

The explicit requirements for the startup change are:

- **RS1: A *triplwave* command must be available on the system path for starting TripleWave.**

The source files do not have to be in the current working directory from which TripleWave is started.

- **RS2: The startup task must locate the configuration in the current working directory or through a custom path relative to the current working directory.**
  The configuration files do not have to be placed at an exact position in the system, but can be organized by the user.

- **RS3: TripleWave must support global installations through *npm* or *yarn* such that no further code changes are needed for publishing TripleWave on the *npm registry*.**
  TripleWave should follow the NodeJS default for installing command line applications.[3]

- **RS4: TripleWave may allow to start multiple instances simultaneously (In conjunction with RC4).**
  Multiple TripleWave instances can be started simultaneously without the need for multiple configurations with multiple manual startups.

3.4.3 Supported Data Formats

Looking at the amount of datasets in a certain format (see Table 3.5), it can be seen that most of the datasets are available either in CSV, JSON or XML. RSS feeds and WSDL are a vocabulary of XML, so the total of datasets that are available in XML amounts to 5 datasets. Some datasets are also available in other formats.

Plain Text files (TXT) often contain unstructured or semi-structured data. This makes it hard to parse and use them. Transformations for this filetype would be possible but generally only with a lot of processing.

XLSX [4] is hard to parse as it contains a hierarchy of XML files which describe the structure and look of an Excel Stylesheet. This format is therefore cumbersome to handle because it spreads data across multiple files and mixes it with styling definitions.

WMS [5] and WFS [6] are dataservices for Geodata.

[3] *https://docs.npmjs.com/getting-started/installing-npm-packages-globally#how-to-install-global-packages* (accessed 27.06.2018)
[4] *http://officeopenxml.com/anatomyofOOXML-xlsx.php* (accessed 20.06.2018)
[5] *http://www.opengeospatial.org/standards/wms* (accessed 20.06.2018)
[6] *http://www.opengeospatial.org/standards/wfs* (accessed 20.06.2018)

Type	Amount
CSV	7
JSON	6
RSS	3
TXT	2
XML	1
XLSX	1
WSDL	1
WMS	1
WFS	1

Table 3.5: The datatypes of the datasets ordered for the amount of their occurrence

To reduce the scope of the thesis, only the most occurring formats and the most usable formats are considered. These are the three formats CSV, JSON and XML (which includes RSS and WSDL).

The explicit requirements for the supported data types are:

- **RF1: The CSV, JSON and XML datatypes must be supported.**
  TripleWave must accept and process datasets in the most common formats as shown above.
- **RF2: Standardized query languages for the supported datatypes must be supported.**
  The user can refer to values in the datasets with known, powerful and standardized methods.

3.4.4 Input source

Until now, TripleWave has three possible input sources, namely: time annotated RDF datasources, RDF streams and non-RDF streams (Mauri et al., 2016). Looking at the OGD datasources listed in Section 3.3, the datasets get updated frequently but the updates are applied to files rather than published in a stream. In addition, none of the datasets has more than three stars, which means that the data is not linked and data items are not uniquely identifiable. Therefore, this data does not serve as an RDF input and in the current state of TripleWave it can only be passed to TripleWave through the non-RDF stream source input. This requires the data to be a stream but the datasets are static and therefore they cannot serve non-RDF streams. One approach to solve this problem is to create a configuration script for TripleWaves' transform mode (Mauri et al., 2016) which pulls the datasets from the Web and transforms them into a data stream which is passed to TripleWave. Muntwyler (2017) used a similar approach with Apache Kafka as event broker and external connector scripts between the dataset on the Web and Kafka as well as between Kafka and TripleWave. These solutions do not seem to be streamlined and every new dataset needs its own new transformation to a data stream and an R2RML transform definition.

The explicit requirements for the new input source are:

- **RD1: TripleWave must be extended to accept a new input source for frequently updated datasets.**
  No additional retrieval and input scripts will be necessary to feed TripleWave with data from frequently updated datasets located on the Web or locally.
- **RD2: A new transformation configuration must transform the data from the new datasource to an RDF representation (see Section 3.4.5).**
  TripleWave will be able to transform CSV, JSON and XML (see Section 3.4.3) datasources to RDF. The current transformation through the integrated R2RML processor is not expressive enough to handle all forms of datasets (see Sections 3.1 and 3.4.5).
- **RD3: The data retrieval must follow a time-based interval instruction.**
  Frequent changes in a dataset can be pulled in the same intervals as the dataset gets updated, to publish the data as close to the updates as possible. A scheduler, as shown in Figure 5.10, must be created to schedule the request timing.
- **RD4: The data retrieval must be pull-based.**
  As the datasources are static files and do not push change notifications to clients, there is no indication that a dataset has changed. Pulling the datasets in the intervals in which they are supposed to be updated is therefore the best mechanism to get up-to-date data.

3.4.5 A new way of transformation

Because of the shortcomings of the current R2RML transformation implemented in TripleWave (see Section 3.1), a new transformation has to be created (RD2, Section 3.4.4) for handling the complex transformations necessary for publishing frequently updated datasets on the Web such as OGD datasets published in CSV, XML or JSON. Unlike the current solution, the new transformation must be able to handle the complex forms in which the datasets arrive. In case of XML or JSON, this means that nested datastructures have to be easy to access and map to RDF.

Another problem with the current state of TripleWave is that the datasets have to be downloaded, reformed and put into a stream – all through an additional script, for that TripleWave can digest and transform it through the R2RML transform. Declarative mappings targeted directly to the datasets remove the need for an upfront data-transformation through additional, user-defined code. The discussion of the best solution and its implementation can be found in Chapter 4.

- **RT1: The transformation should be expressive enough to transform the listed OGD datasets in Section 3.3 and by Muntwyler (2017).**
  This applies only to the datasets which are represented in the supported formats (see Section 3.4.3).
  A wide number of existing datasets can be transformed with the new solution.

- **RT2: The transformation must identify the data items in the input file (which must be the RDF entities in the output).**
  Entities can be specified explicitly rather than using a convention for inferring the entities from the source data. Conventions, such as each row in a CSV defines an entity, only work if the user is in control of the data format. An RDF entity is the subject to a set of triples.

- **RT3: The transformation must present a way to compute the properties of an entity from the datasource.**
  The properties applied to the entities will be defined explicitly. Additional information can be added and the value format be changed if needed. For example: if the user needs a timestamp from the datasource as a property in the new RDF entity and he wants it in a specific timestamp format, he needs a way to define the format mapping as well as where in the dataset to take the previous timestamp from.

- **RT4: The transformation must create RDF output.**
  The output can be directly streamed out of TripleWave in JSON-LD, the representation of RDF that TripleWave uses in its output streams.

```
module.exports = {
    example: "configuration"
}
```

(a) Javascript

```
{
    "example": "configuration"
}
```

(b) JSON

Figure 4.1: A basic configuration layout in (a) Javascript and (b) JSON

4

Changes to TripleWave

In this chapter, I design a solution for the requirements for the configuration and the startup of TripleWave (see Sections 3.4.1 and 3.4.2 respectively).

4.1 The configuration

The first part of the solution is a new configuration for TripleWave based on the old solution. I explain my choice of the new representation in Section 4.1.1, show the description of the new configuration and how I achieved the mapping of the properties in Section 4.1.2.

4.1.1 Choosing the right representation

The two standard ways of configuration in Javascript and the specification of requirement RC1 are to put the configuration into a *.json* or a *.js* file. The *.json* file holds a serialized version of a Javascript object (JSON) and lends itself well for descriptive configurations. A Javascript file (*.js*) is used in cases where serialized Javascript objects are not enough. It allows for the dynamic creation of the configuration and for passing functions/code as part of the configuration.

Both JSON and Javascript objects have the same structure, however, a Javascript file with a configuration object does not implicitly expose the object. It has to be exported from the Javascript file through the export mechanism of NodeJS: *module.exports*. Figure 4.1a shows an example configuration in Javascript with the module export. On the right, Figure 4.1b shows the same configuration in JSON.

Propertyname	**Description**
port	The port of the TripleWave server
hostname	The hostname of the TripleWave server
path	The path to the root of the server
externaladdress	The external address to the TripleWave server; If not given, the port, hostname and path are used to construct this address
mode	The mode in which TripleWave runs; Available modes are: *endless*, *replay* and *transform*
sources	The source for the streaming data. Possible sources are: *rdfstream* and *triples*
ws_enabled	A flag to enable the websocket
ws_stream_location	The location (path) to the websocket stream
ws_port	The port that points to the stream
ws_address	The full address to the websocket stream (Typically with the ws:// protocol prefix)
mqtt_enabled	A flag to enable mqtt
mqtt_broker_address	The address of the mqtt broker
mqtt_broker_port	The port of the mqtt broker
mqtt_topic	The mqtt topic to which TripleWave should push the streaming data
transform_folder	Used by TripleWave to point to the folder with the transformer and mapping files
transform_transformer	The transformation component – a NodeJS module that exports a stream for the input to TripleWave which should be transformed
transform_mapping	The mapping component – an R2RML mapping that defines mappings on the data coming from the transformer
rdf_file	The file path to the RDF data
rdf_query_folder	The folder which the file is located in
rdf_query_endpoint	The endpoint which queries are sent to
rdf_update_endpoint	The endpoint which updates are sent to
rdf_stream_item_pattern	The pattern used for finding the key and timestamp of the input data
rdf_stream_content_pattern	The pattern used for inserting new triples
rdfstream_file	The file with rdf data that should be replayed in the mode 'rdfstream'.
tbox_stream_location	The tbox stream location

Table 4.1: A list of all possible configuration properties for TripleWave

Both configurations look similar, yet the JSON configuration looks a bit cleaner because there is no need for the `module.exports` = part.

All configuration properties that can be passed to TripleWave in its current state are represented in Table 4.1. These are the options that still have to exist in the new version of the configuration.

All the properties in a *.properties* configuration that TripleWave expects currently are of type text. Therefore, the listed properties must be entered in text format. This includes paths to the code that Triplewave uses as input, namely the property *transform_transformer*. This property has to be given together with *transform_folder* to build the path for a Javascript source file that exports a stream which TripleWave uses as data input.

This configuration could be directly translated to a JSON configuration overtaking the property-names with textual values. However, JSON, unlike *.properties* allows nested structures which makes the grouping of related properties possible. In the case of TripleWave, a good example is the grouping of all configuration properties for the websocket. The same grouping is possible with Javascript objects as JSON represents serialized Javascript datastructures.

The choice between a Javascript object and JSON comes down to the measure in which they differ. This is the dynamic aspect of Javascript. Javascript allows to compute a path instead of hardcoding it into the configuration file. For example, the *path*[1] utility from NodeJS is very handy for this. Another feature added by Javascript is that code can be passed as part of the configuration. It can be seen, in the example about the *transform_folder* above, that Javascript allows a more succinct solution for the configuration. For one, the *transform_folder* property becomes obsolete as the relative and absolute path to the file can be easily computed. In addition, the *transform_transformer* can be changed to take a NodeJS Stream object[2] directly instead of taking a filename that points to a Javascript file which exports one.

The downside of using Javascript objects is that the code assigned to the configuration is ignored when the object is serialized. It follows that it is not possible to send a serialized configuration over the network or store it in a database. Nevertheless, no requirements are calling for a serialization of TripleWave configurations.

4.1.2 Choosing the layout

The properties described in Table 4.1 have to be mapped to a Javascript configuration. New possibilities allow for new structures. Javascript objects can group properties and take references to Javascript values as property values.

For the mapping of the old configuration to Javascript objects, I define the following rules:

1. Use the naming of the old properties

[1] *https://nodejs.org/dist/latest-v10.x/docs/api/path.html* (accessed 26.06.2018)
[2] *https://nodejs.org/dist/latest-v10.x/docs/api/stream.html* (accessed 26.06.2018)

2. Use the underscore (_) as breakpoint for nesting
3. Use nesting only for grouping
4. Group properties with the same prefix
5. Keep the underscore for multipart-names in case no grouping can be created at the breakpoint between the parts of the name (e.g. *stream_item*)
6. If there is a better solution than for the rules above, apply the better solution. (e.g. *rdf_query_folder* and *rdf_query_endpoint* would be mapped according to the above rules but a grouping of *rdf_query_endpoint* with *rdf_update_endpoint* makes more sense)

The result of the mapping is shown in Table 4.2. The *transform_folder* and the *rdf_query_folder* are not needed anymore and are therefore marked with a - (dash). The configuration for *transform_folder* is now implicit in the properties *transform.transformer* and *transform.mapping*, the configuration for *rdf_query_folder* in *rdf.file*.

The main groups are *ws*, *mqtt* and *rdf*. I created only a few nested groups where Rule 4 applies, namely the *mqtt.broker*, the *rdf.endpoint* and the *rdf.stream_item*. The *rdf.endpoint* mapping applies to the example of Rule 6. These are the only names for which the name changed slightly. The semantic meaning of *endpoint.query* and *query_endpoint* is arguably the same as the dot signifies a particular group such that *endpoint.query* is a query of the group endpoint and is thus a query endpoint. No grouping can be established for *stream_location*, *stream_item* and *tbox_stream_location*. In those cases, Rule 5 applies and thus I keep the underscore.

In the following list, I evaluate if the requirements for the configurations are fulfilled:

- **RC1** The requirement RC1 is fulfilled due to the usage of *.js* files for the configuration.
- **RC2** All configuration options from the current version of TripleWave are still available. The removed options for declaring folder paths are now implicit to the options which take absolute paths.
- **RC3** Regarding the discussion above concerning the grouping rules, RC3 is fulfilled as the new names keep the semantics of the current configuration.
- **RC4** The last requirement, RC4, is discussed in Section 4.2 because it is tied tightly to TripleWaves' startup procedure.

4.2 Startup

The refactoring of TripleWave enables installing it globally through *npm* (RS3). Following the *npm* guidelines, a path to an executable has to be added to the *package.json* file with the *bin* entry.[3]

[3] *https://docs.npmjs.com/files/package.json#bin* (accessed 28.06.2018)

Properties	**Javascript object**
port	port
hostname	hostname
path	path
externaladdress	externaladdress
mode	mode
sources	sources
ws_enabled	ws.enabled
ws_stream_location	ws.stream_location
ws_port	ws.port
ws_address	ws.address
mqtt_enabled	mqtt.enabled
mqtt_broker_address	mqtt.broker.address
mqtt_broker_port	mqtt.broker.port
mqtt_topic	mqtt.topic
transform_folder	-
transform_transformer	transform.transformer
transform_mapping	transform.mapping
rdf_file	rdf.file
rdf_query_folder	-
rdf_query_endpoint	rdf.endpoint.query
rdf_update_endpoint	rdf.endpoint.update
rdf_stream_item_pattern	rdf.stream_item.pattern
rdf_stream_item_content_pattern	rdf.stream_item.content_pattern
rdfstream_file	rdfstream_file
tbox_stream_location	tbox_stream_location

Table 4.2: The property mappings of the TripleWave configuration from *.properties* to *.js*

While the currently existing *start.sh* and *bash.bat* could serve as executables, it is more appropriate to use the community default instead of adding another language to the codebase like in this case shell or batch scripts. It's is possible and a convention to use Javascript for the startup script. All needed is an installed version of node visible to the path and a single *shebang* line at the top of the Javascript file that should serve as the command line script: `#!usr/bin/env node`.

The Javascript file can then be executed from the command line like any other command line script. The command line arguments are available in the global variable *process.argv* (*argv* stands for *argument vector*).

At the time of the publication of TripleWave, *npm* reads all informations it has to know about the module from the *package.json* file. One of them, as stated above, is the *bin* entry. Adding the Javascript file with the NodeJS shebang allows *npm* to find the script and install it on the path when the module gets installed globally.

Figure 4.2 shows the *bin* entry for TripleWave. It expects the command line script, following the NodeJS convention, to be located in the *bin* folder relative to the root of TripleWaves' filestructure.

```
"bin": {
    "triplewave": "./bin/triplewave.js"
}
```

Figure 4.2: The *bin* entry for TripleWave

Requirement RS2 targets the configuration parameter passed to the command line script. If TripleWave is called with the command *triplewave*, where should the startup script look for the configuration? In the current version of TripleWave, the convention is that the configuration file is located at the path *./config/config.properties* relative to the root of TripleWave. In addition, the configuration location can be specified as parameter named *--configuration* to the shell or batch script which refers to a configuration relatively to the current working directory. But now, TripleWave can be called from anywhere on the system.

There are not many strategies for a configuration lookup. One possibility is to keep a standard folder like *.triplewave* in the home directory with a list of configurations in it, of which the user can choose from with a command line parameter. A configuration in a standard folder might make sense if he always knows the names of the configurations and does not have to look for them in that particular folder. It is more convenient for a user to pass configurations to an application through a path relative to the current location in the terminal like the current *--configuration* parameter. It would be the best to keep this parameter that is already known to TripleWave users and only add another default behavior. The current default does not work anymore because it looks for the configuration in the source files of TripleWave which should not be accessed in a globally installed TripleWave. Therefore I define a new default behavior. The *--configuration* parameter, if not set explicitly, gets the default value *./tw_config.js*. Like that, the startup script looks for a file named *tw_config.js* in the current working directory if no other path is given. In addition, I add the shortcut *-c* for the configuration parameter for convenience. With this, TripleWave can as example be started with: `triplewave -c another_config.js`.

TripleWaves' new startup script can now start TripleWave from anywhere on the system when the right configuration is given. But at times, users want to start multiple instances of TripleWave, each with a different configuration. This is possible with multiple executions of the *triplewave* command and passing different configuration options to each of them.

To facilitate grouped startups of different configurations, multiple configuration files could be specified in one call to TripleWave. I consider this to be cumbersome and the major usecase is to start up all, or groups of configurations together. Grouping configurations in configuration files can be achieved on file-level. This means that one configuration file holds a group of TripleWave configurations that should start simultaniously. As I explained in Section 4.1, the new TripleWave configuration is done in Javascript files. For multiple configurations in one Javascript file, the configurations can be represented and exported from the Javascript module as an array instead of a Javascript object. The new startup script recognizes that the configuration is an array and forks as many child processes as there are configurations and starts TripleWave instances with the single configurations. Sometimes though, a user could want to run only one configuration of a group, for testing or because only that one is needed. To give the user the possibility to start a single configuration out of the group, I add a new parameter for TripleWave. The new parameter *-i* or *--index* accepts a number which is the (zero-based) index of the configuration in the array.

The following list wraps up this section with an evaluation about the fulfillment of the requirements for the startup:

- **RS1** With the *bin* entry in the *package.json* and the new startup script it is now possible to start TripleWave from the commandline with the command *triplewave* when TripleWave is installed globally on the system. Thus, requirement RS1 is fulfilled.
- **RS2** The *–configuration* parameter together with its new default value allows TripleWave to be started with a configuration relative to the current working directory which fulfills the requirement RS2.
- **RS3** The existing *package.json* delivers all necessary information for TripleWave except the command line script entry, which I created with the *bin* entry. RS1 and RS3 are therefore both fulfilled with the same change.
- **RS4** The requirement RS4 is fulfilled through the possibility to use an array with configurations in the configuration files.
- **RS4** Together with RS4, the requirement RC4 is fulfilled as well.

5

Transforming 3-star data to RDF

The main contribution of this thesis is a solution for retrieving and transforming frequently updated datasets. The requirements for the solution are listed in Section 3.4.

When mapping datasets to other representations, it is crucial to know the domain and what possibilities there are to transform one representation to another. In this thesis, I am concerned with the transformation from CSV, JSON and XML (source) to RDF (target) (RF1). Thus, a transformation from a flat (CSV) or nested (JSON/XML) dataset to a graph-based representation (RDF) is required. In addition, the output RDF must be in JSON-LD format and published as a stream item in the output of TripleWave. The output format of TripleWave describes each stream element as RDF data that is packed in a named graph with a timestamp (Mauri et al., 2016). Named graphs allow talking about graphs and relationship between graphs (Carroll et al., 2005). Barbieri and Valle (2010) propose to use a specific schemata for the IRI of i-graphs (One stream item is an i-graph): *http://ex.org/%stream-name%/URLencode(%timestamp%)*.

During this chapter, I denote an event as one retrieval of the dataset and a stream item as one i-graph for the outgoing data stream.

5.1 Choosing a mapping language

Few solutions for mapping relational data from databases to RDF exist. Two known ones are R2RML (Das et al., 2012) and D2RQ (Bizer and Seaborne, 2004) (see Section 2.3). A solution that builds on R2RML and extends it is RML (Dimou et al., 2014) which adds definitions for accepting more datasources in addition to databases. In the current specification,[1] RML defines the support for CSV, JSON and XML. RML is the only known mapping language (RF2) supporting the mapping from the datatypes CSV, JSON and XML to RDF (Dimou et al., 2014). These datatypes match exactly the requirement RF1 and therefore, I consider RML as the best contestant to fit the requirements. By using RML, both requirements concerning the data formats are fulfilled (see Section 3.4.3). R2RML and D2RQ cannot be used for the transformation because both do not define variants to access or refer to values in nested datastructures such as in JSON or XML.

[1] *http://rml.io/spec.html* (accessed 30.06.2018)

RML is not necessarily a perfect fit for the transformation of the datasets even though it fits the best out of D2RQ, R2RML and RML. I now elaborate what the RML provides that is of use and if RML covers the requirements:[2]

- A means to define a data source (*rml:source*). Supported datasources are relational databases, CSV, XML and JSON.
- A means to define a query language which should be used to refer to values in the dataset (*rml:referenceFormulation*).
- A means to define an iteration instruction through the query language (*rml:iterator*).
- A means to define the graph in which a subject is defined (*rr:graph* / *rr:graphMap*).
- A means to define what the subject of the transformation is (*rr:subject* / *rr:subjectMap*).
- A means to define what properties are assigned to the subject (*rr:predicateObjectMap*).
- A means to interpolate query results into a string for the names of subjects, objects and graphs (*rr:template*).

The prefixes of the predicates like *rr* and *rml* define the namespace (IRI) in which the predicates are defined. The RML mapping language is a specification which makes use of the Turtle syntax (see Section 2.2) – a syntax which can be serialized to RDF graphs.

What RML provides is important to know for the next subsections where I show aspects and problems of the data mapping in connection with the OGD datasets and how RML applies to the observed problems. In Section 5.2, I show a solution to the mapping problems.

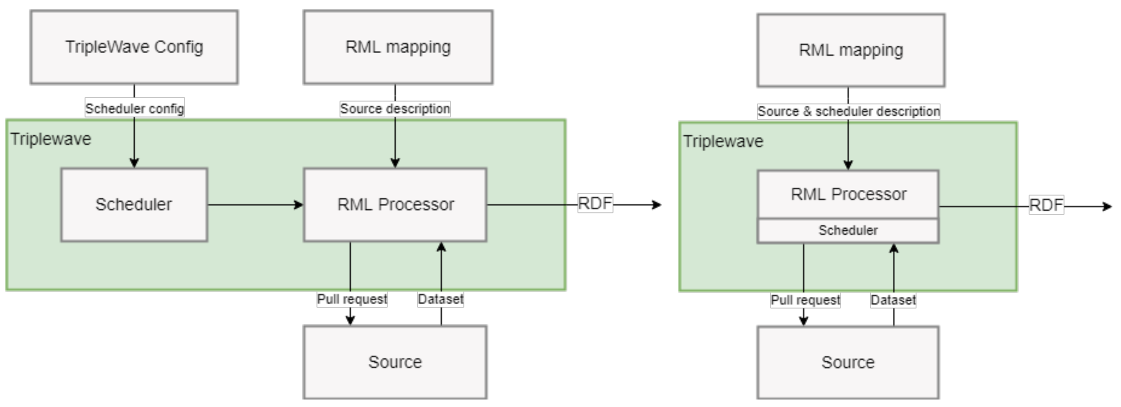
5.1.1 Encoding

OGD datasets have no guidelines or restrictions on their formatting or their encoding. The result is that datasets can come in various encodings. One example dataset is TAQMZH with the encoding *latin1* (*ISO 8859-1*). RML has no means to define the encoding of the input sources. I argue that an automatic detection is possible but is out of scope for this thesis. In addition to a (future) automatic detection, I propose to have the possibility to provide the encoding as a parameter or configuration for the transformation as this is generally a feature users want to control.

5.1.2 Scheduling

The mapping languages mentioned above, for example RML, are created for mapping static datasets and database data to RDF. In RML, the source of a dataset which is the target to a transformation can be defined explicitely as source using an URL through

[2]The prefix *rr* stands for the R2RML namespace (*http://www.w3.org/ns/r2rml#* accessed 02.07.2018), the prefix *rml* for the RML namespace (*http://semweb.mmlab.be/ns/rml#* accessed 02.07.2018)



(a) External scheduler (b) Internal scheduler

Figure 5.1: Two possible designs for the scheduler integration

which the RML processor is able to pull the dataset from the location given as *rml:source*. A transform processor can be triggered any time (Dimou et al., 2014).

Regarding the frequently updated datasets, I need a solution for triggering the processor in predefined intervals or at certain times during the day/month/year. I name that solution a *scheduler*.

RML does not define means for scheduled processing (Dimou et al., 2014). It is possible to create the scheduling as a component that lives outside of the mapping declaration. The difference between a solution where the RML processor is and is not in charge can be seen in Figures 5.1a and 5.1b respectively. The separation of concerns does not become clear when the RML processor is not in charge of the scheduling. This is because the RML processor is responsible for the retrieval of the datasource. When the same component as the scheduler is responsible for the datasource retrieval, the separation of concerns becomes clear. That component is then responsible for the data retrieval and the input of the data to the RML processor. I.e. the best solution is an integrated scheduler in the RML processor which is not subject to the RML specification.

```
{ "IsOpen": false,
  "MBA1_8": 0, "MBA2": 0, "MBA3": 0, "MBA4_5": 0, "MBA6_7": 0,
  "MBA9": 0, "MBA10": 0, "MBA11": 0, "MBA12": 0, "MBA13_14": 0,
  "MBA15": 0, "MBA16": 0, "MBA17": 0, "MBA18": 0, "MBA19": 0,
  "MBA20": 0, "MBA21": 0, "MBA22": 0, "MBA23": 0,
  "Timestamp": "11:36",
  "Wartekreis": 0 }
```

Figure 5.2: The dataset WWAU as an example source

5.1.3 Subject aggregation

Figure 5.2 shows an example of a source in JSON representation. It is the dataset WWAU holding the current waiting time approximation at the pass service counter at different Municipal District Offices (in German: *Magistratische Bezirksämter*, MBA). There are two possibilities to map this dataset to RDF:

1. The whole data entry is considered to be a subject.
   In this case only one stream item is created and the subject name is the timestamp. In other words, all subjects are aggregated into one stream element as shown in Figure 5.3a. The dots (....) are a placeholder for the rest of the elements.

2. Each MBA entry is considered to be a subject.
   In which case 19 stream items are created, each in a graph with the same timestamp but transporting a different subject. In other words, no aggregation of subjects is created and therefore, each subject is presented as a single stream element. A sample of a single stream element is shown in Figure 5.3b.

The general difference between the two approaches is the aggregation. The first approach aggregates the subjects which are in the same graph and originate from the same event. When the subjects are not in the same graph, they cannot be aggregated into one stream item. Both approaches could be wanted by users of TripleWave and therefore, there must be a means to declare on what level an aggregation is wanted.

I propose the creation of three levels of aggregation:

- The level ***all*** aggregates all subjects that are in the same graph into the respective graph. (See Figure 5.3a)

- The level ***subjects*** aggregates all subjects that are generated by one iteration through the source into their respective graph.

- The level ***none*** does not apply any aggregation. All subjects are the single items of their graph and as many stream items as there are subjects are generated. (see Figure 5.3b)

There is no aggregation specification for RML because RML does not work on data streams. RML specifies how to retrieve data and transform it into RDF for producing a set of triples for each source retrieval (Dimou et al., 2014). The only possible aggregation in RDF is the collection of triples in a named graph. Having a data stream of RDF introduces new aggregation possibilities.

When an internal scheduler is added to a RML processor (like in Figure 5.1b), the output of the RML processor must be a data stream. I.e. the aggregation of subjects into stream items must be the responsibility of the RML processor.

```
{
  "http://www.w3.org/ns/prov#
      generatedAtTime": "2018-07-02T11
      :51:26.535+02:00"
  "@id": "http://streamreasoning.org/
      wwau/2018-07-02T16%3A35%3A00%2B01
      %3A00",
  "@graph": [
    {
      "@id": "http://streamreasoning.
          org/MBA10",
      "http://www.w3.org/2006/time#
          hasDuration": [
        {
          "@value": "PT0M",
          "@type": "http://www.w3.org
              /2001/XMLSchema#duration"
        }
      ]
    },
    {
      "@id": "http://streamreasoning.
          org/MBA11",
      "http://www.w3.org/2006/time#
          hasDuration": [
        {
          "@value": "PT0M",
          "@type": "http://www.w3.org
              /2001/XMLSchema#duration"
        }
      ]
    },
    ....
  ]
}
```

(a) Aggregation of subjects

```
{
  "http://www.w3.org/ns/prov#
      generatedAtTime": "2018-07-02T11
      :51:26.535+02:00"
  "@id": "http://streamreasoning.org/
      wwau/2018-07-02T16%3A35%3A00%2B01
      %3A00",
  "@graph": [
    {
      "@id": "http://streamreasoning.
          org/MBA20",
      "http://www.w3.org/2006/time#
          hasDuration": [
        {
          "@value": "PT0M",
          "@type": "http://www.w3.org
              /2001/XMLSchema#duration"
        }
      ]
    }
  ]
}
```

(b) No aggregation of subjects

Figure 5.3: The two different possible outputs for the WWAU dataset

5.1.4 Multiple subjects per iteration

The RML mapping specification allows to define subjects on the level of an iteration such that every iteration step produces one subject (Dimou et al., 2014). For some datasets, this is not enough. One of them is the dataset WWAU shown in Figure 5.2. The RML mapping allows to define an iteration expression which in most cases works well for defining the exact subjects. In the example of WWAU, the iteration expression has to match exactly every key/value pair of all *MBA*{*number*} entries. The default reference formulation (query language) in RML for referencing values in a JSON source is JSONPath[3] (Dimou et al., 2014). It follows that no iteration can be specified with JSONPath for referring to only exactly each *MBA*{*number*} key/value pair in the dataset. In addition, if an iteration could theoretically be defined, back-references to the key for the usage as subject name are impossible.

The creation of dedicated mappings for single values inside an iteration can solve this problem. When each iteration can yield multiple (even different from each other) subjects in every iteration, the case of WWAU can be solved without defining an iteration (respectively one iteration only for the whole dataset). One subject can then be defined for every *MBA*{*number*} key/value pair and thus one iteration yields as many *MBA*{*number*} as needed.

This addition to RML enables the definition of different property mappings for each subject. Nevertheless, there is no dataset in my list for which this feature is needed.

5.1.5 Advanced transformations

The RML (and R2RML) mappings are one-to-one mappings. A value in the source gets mapped to a value in the RDF output, with the optional addition of a language (*rr:language*) or datatype (*rr:datatype*).

OGD data publishers, publish the data in the format they prefer. Because users of TripleWave and therefore users of an integrated RML processor are not in control of the data representation of the OGD datasets, the data is likely published in a format that is either unexpected or not in the format required or wanted for the output.

A reoccurring example are timestamps published in various representations and notations. An example is the previously presented dataset WWAU. The timestamp it delivers shows only two digits for the hours and two digits for the minutes seperated by a colon. This timestamp represents the time of the current day at which the status was last updated. The date part is missing and must be recreated through a reference to the date of the current day. Another dataset, PKDE, delivers the timestamp in the format: *04.07.2018 13:40:00*. The date is given in German notation and the timestamp includes hours, minutes and seconds. I.e. different datasets adopt different time formats. The timestamp in the output stream should be compilant to the proposal of Barbieri and Valle (2010), and be part of the graph: *http://ex.org/%streamname%/URLencode(%timestamp%)*. Direct timestamp mappings result in a range of various formats, different for each dataset. A mapping from an arbitrary timestamp to a standardized output format has requirements exceeding a simple restructurization as the case of WWAU shows. The missing date, seconds and the timezone should be added to the delivered timestamp. The date can be obtained with a lookup of the current date, the seconds set to zero when no seconds are available. The best guess for the timezone is the timezone in which the data was created. In the case of WWAU, the timezone is likely GMT+1 given the dataset was created in Vienna. The timestamp of PKDE misses only the timestamp.

Another example is the FAU dataset. It is published as a CSV file with stations

[3] *http://goessner.net/articles/JsonPath/* (accessed 03.07.2018)

(subjects) on each row and the measurements for particulate matter at different times during the day (hour steps) is presented in the columns. Hourly updates to the dataset fill the measurements into the already existing but not yet filled 24 measurement columns. One column for each hour of the day. Yet unavailable measurements are denoted with the placeholder *-999*. I need from the datasets only the newest measurement of each station. To do so, the last column before *-999* must be found. Another strategy is chosing the column by the current time of retrieval. Here, like in the example with the timestamps above, additional logic to RML is required to achieve a specific selection.

A simple RML direct mapping cannot manipulate, reformat or make additions to the input data. For example, there is no definition in RML for referring to the current date to add it to the timestamp. There is no definition for applying string concatenation or substring replacement. Further, the specific selection of data is not possible with RML when special conditions are needed for which information like the current time of the day is necessary.

Consequently, I need a solution that allows applying arbitrary transformations to the input data if necessary together with a solution for better data selection. A possible solution is an expansion of RML with conditional elements and advanced expressions (for example like in XSL). Another solution is using RML as mapping language and adding an interface to the RML processor for applying hooks through an already existing programming language. Finally, it should be considered moving the RML mapping from a representation in RDF format to a programming language in which the transformations can be directly supplied together with the data query instructions.

5.1.6 Full source transform before mapping

In some cases, the structure of input data can be totally different to expectations about a format. For example a CSV file presenting data in a very particular way. A concrete example is the TAQMZH dataset. The datarows start at row seven, the first column contains the timestamp. When new data comes in, a row is added below. The rest of the columns contain measurement data. The first seven rows contain metadata. The first row shows where the measurement is taken. The second row presents a shortname for the name of the place. The third row contains a long name for the measurement, the forth a short chemical name. Row five shows the name of the file and lastly, row six the measurement unit. The metadata for the measurements is defined on each column and therefore annotates the measurements of that column. A small extract of the dataset is shown in Table 5.1.

The measurement locations are lined up in the columns. Metadata is described on the columns as well, the timestamp on the other hand on the row. The expected RDF output for this dataset is a graph with the timestamp and containing the observations, each with its measurement and metadata as properties of the observation.

An RML processor must be able to parse an input file to represent it in a format in memory through which a query language can refer to values. The processing of the TAQMZH dataset fails at this step because the CSV does not follow the convention of a header now describing the properties. The remaining rows represent the values

Datum	Zürich Stampfenbachstrasse	Zürich Stampfenbachstrasse
	Zch_Stampfenbachstrasse	Zch_Stampfenbachstrasse
	Ozon, höchstes Stundenmittel	Stickstoffdioxid
	O3_max_h1	NO2
	d1	d1
	$\mu g/m3$	$\mu g/m3$
04.06.2018	113.68	23.96
05.06.2018	121.13	20.27
06.06.2018	121.73	14.39
07.06.2018	99.84	15.65

Table 5.1: An extract of the TAQMZH dataset

corresponding to the properties. A mechanism must therefore exist for transforming the dataset to a format interpretable by the processor. The mechanism must be applied when the data is retrieved but before the parsing takes place. Consequently, the mechanism must consist of a middleware taking text input and producing a transformed text output used by the parser of the RML processor.

Most of the problems described in the previous sections can be solved with this approach. It should only be used for the most complex datasets when no other solution exists.

5.1.7 The solution space

The analysis of the datasets and their specific properties unveiled the non-existence of specifications needed to express mappings for the OGD datasets shown in Section 3.3. I shortly presented ways to overcome the detected problems with the datasets.

Other mapping specifications like RML, namely R2RML and D2RQ do not provide a solution. R2RML is a subset of RML and D2RQ is targeted towards relational databases only.

Below, I list a number of possible alternatives to using RML. Not all of them are viable options. I discuss which alternatives are viable and which are not in the following paragraphs:

1. Extend RML with more expressions.
2. Use RML for the mapping and add hooks into the RML processor for a general purpose language.
3. Reuse the RML definitions from the specification but write the mapping in a general purpose language with extensions/adaptions for providing pieces of logic (functions) in the place of queries.

Alternative (1) reuses the RML specification and builds on top of it. Unfortunately, RML is written in RDF and extending it to support advanced transformations is out of scope of this thesis as a solution must be Turing complete due to the needed advanced transformations. An extension of RML to form a Turing complete language is a bigger task than can be covered in a bachelor thesis. Further, RML is a descriptive language and self-contained. For example, referring to the current time of a mapping execution should hardly be described in RDF.

Alternative (2) has the advantage that it does not try to bring features of general purpose languages into a language in that they don't belong. Hooks into a running parsing or mapping process is a well-known concept, e.g. the Simple API for XML (SAX).[4] The downside of Alternative (2) is that the mapping and the processing logic are split. This is a negative factor because the real output cannot be inferred anymore by reading the mapping, as the hooks apply additional transformations. The previously descriptive mapping looses its integrity through the split.

Alternative (3) breaks the RML approach: moving the specification from one language to another requires to change the structure of the mapping definitions. Through the change from RDF to another language, the mapping definitions lose the namespace prefix (see Section 5.2.1). Further, the mapping cannot be serialized anymore as logic (functions) is added to the mapping definition. This is no problem as long as the mapping definitions don't have to be sent through the network in the form of data. The advantage of this approach is the disadvantage of alternative (2): the mapping is self-contained and necessary logic can be added where the mappings are defined.

Due to the discussion above, I choose Alternative (3) as a solution. In the following section, I show how the problems above can be solved with such an alternative.

5.2 JRML

Since TripleWave is built in Javascript, it is natural to create the mapping processor in Javascript as well. Javascript has made its way to a general purpose programming language throughout recent years and runs with NodeJS natively, decoupled from browsers. As discussed in Section 4.1, configurations in Javascript can be written in JSON or Javascript objects. An RML-like mapping allowing advanced transformations (see Section 5.1) can only be defined in a programming language - here Javascript. I.e. JSON cannot be used because it cannot express logic (functions).

I propose JRML, to specify, RML mappings in Javascript. JRML is an extended and slightly restructured version of RML with the structure retained as much as possible, but translated into Javascript objects.

In the following, I describe how I map the RML specification to Javascript objects and how I adapt and extend it to solve the problems described in Section 5.1.

5.2.1 General RML to JRML mapping

The structure of RML is nested and consists of RDF triples. Its hierarchical structure is fairly simple to translate to a Javascript representation. Figure 5.4 shows a straight-

[4] *http://www.saxproject.org* (accessed 05.07.2018)

forward example of a direct mapping from RML to JRML.

The hierarchy directly maps to JRML. It should be noted that the graph description (*#VenueMapping*) is lost in the new mapping, together with the namespace of the predicates. The predicate *rml:logicalSource* for example loses the namespace *http://semweb.mmlab.be/ns/rml#*. The namespaces are used for reference and not important for the functionality. The loss of the graph can in theory be compensated by putting the Javascript object in a named variable or introduce a further nesting layer for graph names wrapping the mapping definition. This is not necessary for the range of this thesis because the purpose of JRML is to map OGD datasets to data streams and cross-linking between datasets is not considered in this thesis. A reference name for the data source is thus not needed.

The namespaces of the predicates are still necessary and can be used in JRML through functions creating named RDF nodes. To do so, it is necessary to import the library *N3*.[5] *N3* implements the RDF Representation specification[6] and is built to enable treating RDF in Javascript.

A structural change is the usage of *predicateObjectMaps* as an array instead of a single *predicateObjectMap*. Opposed to RML, Javascript objects cannot define multiple properties with the same name in the same Javascript object (Javascript object keys are unique). I.e. subsequent *rr:predicateObjectMap* from RML are represented in JRML as an array of *predicateObjectMap* objects. The same change applies to *rr:class* which is an array named *classes* in JRML.

5.2.2 Encoding

An extension of RML is needed for declaring the encoding in which the source is parsed as described in Section 5.1.1. This can be a simple property named *encoding* in the section *logicalSource*. I choose this section because the encoding is part of the metadata characterizing the source. The property value is a string with the name of the encoding. The possible encodings are all encodings accepted by the *toString* method of a NodeJS buffer.[7]

5.2.3 Scheduling

The scheduling mechanism is best integrated in the RML processor as discussed in Section 5.1.2. JRML needs additional definitions to specify in what interval or during what times of the day the source has to be retrieved, transformed and sent out through an RDF stream. The scheduling can be described as a key/value pair on the *logicalSource* like the *encoding*. A key/value pair describing the scheduling consists of the *scheduler* key with the value being a Javascript object. The *scheduler* object accepts three properties:

[5] *https://www.npmjs.com/package/n3* (accessed 05.07.2018)
[6] *http://rdf.js.org* (accessed 05.07.2018)
[7] *https://nodejs.org/docs/latest/api/buffer.html#buffer_buf_tostring_encoding_start_end* (accessed 05.07.2018)

```
@prefix rr: <http://www.w3.org/ns/
    r2rml#>.
@prefix rml: <http://semweb.mmlab.be/
    ns/rml#>.
@prefix ql: <http://semweb.mmlab.be/ns
    /ql#>.
@prefix xsd: <http://www.w3.org/2001/
    XMLSchema#>.
@prefix schema: <http://schema.org/>.
@prefix wgs84_pos: <http://www.w3.org
    /2003/01/geo/wgs84_pos#lat>.
@prefix gn: <http://www.geonames.org/
    ontology#>.

<#VenueMapping>
  rml:logicalSource [
    rml:source "http://www.example.com
        /files/Venue.json";
    rml:referenceFormulation ql:
        JSONPath;
    rml:iterator "$"
  ];

  rr:subjectMap [
    rr:template "http://loc.example.
        com/city/{$.location.city}";
    rr:class schema:City
  ];

  rr:predicateObjectMap [
    rr:predicate wgs84_pos:lat;
    rr:objectMap [
      rml:reference "$.venue.latitude"
    ]
  ];
```

(a) Example RML mapping in turtle format (source *http://rml.io/spec.html#example-JSON*)

```
const N3 = require('n3');
const { namedNode } = N3.DataFactory;

function ql(id) {
    return namedNode("http://semweb.
       mmlab.be/ns/ql#" + id);
}

function schema(id) {
  return namedNode("http://schema.org/
      " + id);
}

function wgs84_pos(id) {
  return namedNode("http://www.w3.org
      /2003/01/geo/wgs83_pos#" + id);
}

module.exports = {
  logicalSource: {
    source: "http://www.example.com/
       files/Venue.json",
    referenceFormulation: ql("JSONPath
       "),
    iterator: "$"
  },
  subjectMap: {
    template: "http://loc.example.com/
       city/{$.location.city}",
    classes: [ schema("City") ]
  },
  predicateObjectMaps: [
    {
      predicate: wgs84_pos("lat"),
      objectMap: {
        reference: "$.venue.latitude"
      }
    }
  ]
}
```

(b) Direct JRML mapping of Figure 5.4a

Figure 5.4: An example direct mapping from RML to JRML (note: this is not final the final JRML version)

- *interval*: it is a number (in milliseconds) defining an interval in which the dataset should be fetched.

- *cron*: it is a cron pattern (string) as specified at *http://crontab.org*. It defines specific times during a day at which the dataset should be refetched.
- *timeZone*: it is a timezone string. The available timezones can be found at *http://momentjs.com/timezone/*.

An *interval* and a *cron* job can be defined at the same time, while a *timeZone* specifies the *cron* value further.

5.2.4 Subject aggregation

Subject aggregation can be achieved in JRML with the new property *aggregate* in the section *logicalSource*. The aggregation levels that I proposed in Section 5.1.3 *all*, *subjects* and *none* can be applied. To rehearse:

- *all* aggregates all subjects living in the same graph into the respective graph.
- *subjects* aggregates all subjects that are generated by one iteration through the source into their respective graph.
- *none* does not apply any aggregation. All subjects are the single items of their graph.

5.2.5 Multiple subjects per iteration

Allowing multiple subjects per iteration comes with a structural change to the previously shown JRML in Figure 5.4. RML and the prevously shown JRML both put the *subjectMap* directly in the root of the mapping. This indicates that only one subject definition is created per source and is applied for every iteration step.

To allow multiple subjects per iteration, one more nesting level is needed since Javascript object keys are unique and multiple subjectMaps cannot be defined on the same level. In addition, *predicateObjectMap* definitions are bound to the subject and must be grouped with the subject they belong to. For this purpose, JRML introduces the new root property *eventItem*. *eventItem* contains the mapping definitions for each event (an event is one request of the data source). An *eventItem* must be an array of objects. Each object consists of a subject and a *predicateObjectMaps* definition. The properties *subject/subjectMap* and *predicateObjectMaps* are not allowed on the root level anymore to prevent from confusion between subjects in the *eventItem* list and the root.

This structural change allows defining multiple subjects for each iteration step and in addition, it enables defining of a custom *predicateObjectMap* for each subject.

5.2.6 Advanced transformations

One of the most interesting changes from RML to JRML is the advanced transformations that can be applied in place of any query definition. As JRML is written in Javascript, the functions for advanced transformations can be defined directly in Javascript.

I illustrate the advanced transformations on the example of the timestamp mapping of the WWAU dataset as discussed in Section 5.1.5. The timestamp input comes in the format *HH:MM* only, the date is missing from the timestamp together with the seconds and the timezone. To receive a standardized format with the additional date-time information included, adanced transformations are needed. The timestamp must be placed as part of the graph URI (*http://ex.org/%stream-name%/URLencode(%timestamp%)*). First of all, TripleWave must know where to define the graph name. This is done in RML in the *graphMap* definition which is part of the *subjectMap*. The same applies to JRML. The *graphMap* takes a *template* for querying elements in the dataset and for string interpolation. The RML approach directly translated to JRML would be `template: sr("wwau/Timestamp")` where the function *sr* is a namespace wrapper (see Figure 5.4). The reference to the timestamp of the dataset is *Timestamp*. The curly braces in the string are the way RML handles string interpolation with an integrated query. JRML has this feature as well, but in the case of the WWAU timestamp, further transformations are necessary to retrieve the missing date-time components as the example mapping translates to: *http://streamreasoning.org/wwau/11:36*. The timestamp is directly mapped and an URI encoding was not applied either. JRML allows to instead use a function as value for *template* as shown in Figure 5.5. The function takes one parameter named *query* which is a function as well. The function *query* takes a string as an argument, which must be a query in the query language defined in *logicalSource*. Therefore, `query('Timestamp')` returns the *Timestamp* entry of the current iteration. The return value of the function passed to *template* is used as value for the RDF graph in which the subject is placed. The code illustrated in Figure 5.5 uses the NPM module momentjs[8] to transform the timestamp from the WWAU dataset. The function creates a moment time object *time* and adds the minutes and hours from the dataset. It sets the seconds to zero and sets the timezone offset (`time.utcOffset(60)`) before the timestamp is placed in a standardized format and URI encoded into the graph string.

As the example illustrates, arbitrary transformations can be applied through this addition to JRML. A query function can be used in JRML at all places where queries can be applied in the original RML plus *predicateObjectMap.predicate* for the option of dynamic predicates. A query function can be used for:

- *logicalSource.iterator*
- *subjectMap.template*
- *graphMap.template*
- *predicateObjectMap.predicate*
- *predicateObjectMap.objectMap.template*

[8] *https://www.npmjs.com/package/moment* (accessed 06.07.2018)

```
graphMap: {
    template: query => {
        const timestamp = query('Timestamp').shift();
        const [h, min] = timestamp.split(':');
        const time = moment();
        time.second(0);
        time.minute(min);
        time.hour(h);
        return sr(
            `wwau/${encodeURIComponent(
                time.utcOffset(60).format()
            )}`
        );
    }
}
```

Figure 5.5: The graphMap template mapping of the WWAU dataset

5.2.7 Full source transform

I explained in Section 5.1.6 why advanced transformations are not enough for very custom datasets like TAQMZH. The addition of the property *transform* in the *logicalSource* section of the mapping definition allows a source transformation before the source is parsed by the JRML processor. The value of *transform* is a Javascript function taking a string as a parameter. The function must return a string. The input string is the source of the dataset as plain string.

In the case of TAQMZH (see Table 5.1), the source can be transformed to a new CSV string through the re-organization of the values: the header must be one line of text with the values *Date*, *Place*, *Description*, *Measurement*, *Dataset*, *Unit* and *Value*. The rows below contain the values for the respective columns. The described format can be handled by JRML. An iteration can be defined for it and references to values in the iteration can be applied as usual.

5.3 Implementation of JRML and design decisions

In this section, I explain how I implemented JRML and discuss my design decisions at code level.

5.3.1 Setup and interface

I realize the JRML processor as a standalone *NPM module*. For the use of the module in TripleWave, I add the JRML processor as an external dependency. The module exposes only one function: the entry point and the same time the startup instruction for the processor. The startup function takes two parameters:

- *config*: the JRML mapping configuration as explained in Section 5.2

- *collector*: a collector object factory function returning a collector which implements the collector interface (see Section 5.3.3)

When called, the startup function returns a NodeJS data stream.[9] The data stream supplies the RDF items generated by the source transforms of JRML.

5.3.2 Architecture and code style

The JRML processor supports NodeJS versions above and including version *8.3*. It makes extensive use of the *ES6* Javascript specification[10] and *async/await* syntax to write and handle promises[11] in Javascript which are part of the upcoming *ES2017* specification.[12] The *async/await* syntax is supported by NodeJS version *8.3* and newer. The current Long Term Support (LTS) version of NodeJS is at the time of writing *8.11.3* and thus, all new NodeJS installations should be able to run the JRML processor.

I built the code base with functional programming patterns opposed to an object oriented approach for simplifying testing and keeping side-effects at a minimum. The code base consists therefore mainly of plain functions which do not rely on their context and are self-contained. The functions work only on plain Javascript datastructures excluding class constructs with the only exception of NodeJS data streams.

The internals of the JRML processor are shown in Figure 5.6 as a sequence diagram with the functional components on listed on top. The diagram shows how one data source retrieval is processed after the scheduler triggers. The datasource component retrieves the dataset from the web and transforms (parses) it to a format which it can use. The source data is piped into the iteration stream which is a NodeJS Transform stream.[13] When the iteration stream receives a dataset, it starts the iteration through triggering the iterator with the given datasource. The iterator iterates through the dataset according to the instructions given by the mapping declaration. For each iteration step, the mapping transformer is called for processing the active iteration. When the mapping transformer issues a new RDF triple, it passes it to the collector. For simplicity and readability, the `push()` call is marked with an asterisk in the figure to indicate that it is possibly called multiple times. When the aggregation level is set to *none*, the mapping transformer calls *group* on the collector which groups the triples issued since the beginning or since the last *group* call. When the aggregation is set to *subjects*, the *group* call is executed by the iterator whenever one iteration is complete. At the end of all iterations, when all triples are written to the collector, *collect* is called to end the processing and retrieve a new representation of the RDF items (depending on the implementation of the collector). The newly obtained RDF items are pushed to the iteration stream ready for the consumer.

[9] *https://nodejs.org/dist/latest-v10.x/docs/api/stream.html* (accessed 06.07.2018)
[10] *https://www.ecma-international.org/ecma-262/6.0/* (accessed 06.07.2018)
[11] *https://promisesaplus.com* (accessed 06.07.2018)
[12] *https://www.ecma-international.org/ecma-262/8.0/#sec-async-function-definitions* (accessed 06.07.2018)
[13] *https://nodejs.org/dist/latest-v10.x/docs/api/stream.html#stream_duplex_and_transform_streams* (accessed 06.07.2018)

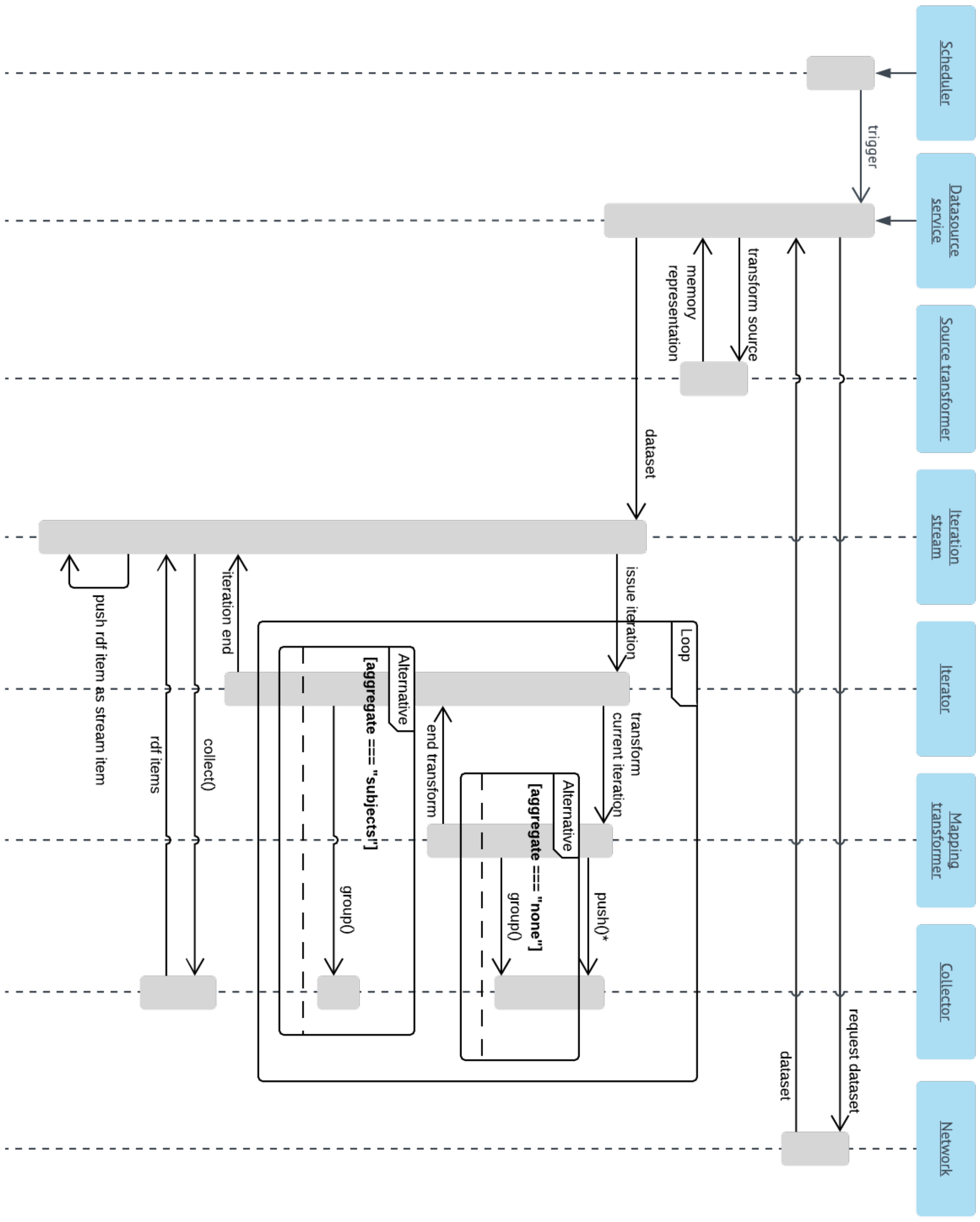


Figure 5.6: The architecture of the JRML processor as sequence diagram.

```
interface Collector {
  push(...args: Term[]): void;
  group(): void;
  collect(): Promise<any>;
}
```

(a) The interface in typescript notation

```
type CollectorFactory = () =>
    Collector;
```

(b) The factory in typescript notation

Figure 5.7: The collector in typescript notation

5.3.3 The collector

A collector is an object that implements the collector interface. The collector is passed as an argument to the JRML processor and can therefore be defined by the user (for example TripleWave). The collector interface is shown in Figure 5.7a (using the typescript interface notation.[14]). The function definition of the collector factory is shown in Figure 5.7b

A *Term* is an object that implements the *Term* interface defined in the RDF Representation specification.[15] A *Promise* is an object that implements the interface[16] for promises that are compatible with the ES6 specification for promises.[17]

An implementation of the collector interface accepts triples or quads through the *push* method and groups them with the method *group*. For keeping track of the pushed triples that should be grouped and later collected, the collector implementation is expected to keep an internal state storing the triples. The interface does not prescribe a representation policy for the internal state as it should not be exposed. The *group* method is expected to group the triples/quads that were pushed since the creation of the collector or since the last call to *group* if available.

5.3.4 The Scheduler

The scheduler is an object internal to the JRML processor and depends on a JRML mapping configuration. The type definitions for the scheduler are shown in Figure 5.8.

The scheduler is responsible for calling a function whenever a defined time interval has passed or a date-time defined by the *cron* string is reached. Therefore, the scheduler is an object which encapsulates the timing logic only and prescribes execution times.

5.3.5 The datasource service

The datasource service consists of a set of functions which depend on a scheduler and the JRML mapping configuration. A datasource is created through the call to the function

[14] *https://www.typescriptlang.org/docs/handbook/interfaces.html* (accessed 09.07.2018)
[15] *http://rdf.js.org/#term-interface* (accessed 09.07.2018)
[16] *https://github.com/Microsoft/TypeScript/blob/master/lib/lib.es2015.promise.d.ts* (accessed 09.07.2018)
[17] *https://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects* (accessed 09.07.2018)

```
interface Scheduler {
  interval?: number;
  cron?: string;
  start(): void;
  resume(): void;
  stop(): void;
}
```

(a) The scheduler interface

```
type SchedulerFactory = ({
  interval?: number,
  cron?: string,
  timeZone?: string
}) => (() => void) => Scheduler;
```

(b) The scheduler factory type

Figure 5.8: The scheduler in typescript notation

which is exported by the datasource module with the JRML mapping configuration, the URL to the source and a scheduler. The function returns a data stream providing new source items whenever the scheduler triggered the service to retrieve a new version of the datasource.

The data service is in charge of applying intitial full source transformations (see Section 5.2.7) and of parsing the input dataset to an internal representation which can be queried:

- JSON is parsed with *JSON.parse*.[18] The internal representation are plain Javascript objects.
- XML is parsed with the npm module *xmldom*.[19] The internal representation is an XML DOM.
- CSV is parsed with the npm module *csv-parse*.[20] The internal representation are plain Javascript objects.

5.3.6 The query mechanism

The query mechanism makes strong use of function currying and partial application. The final query function passed to the JRML mapping configuration must have all necessary information about the query context and iteration available. Partial function application allows a gradual application of context information and works therefore like a multi-level factory function.

The necessary context information for the query function are:

1. The query language defined in the JRML mapping configuration
2. The source that comes from the datasource stream
3. The iteration step

[18] *https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse* (accessed 09.07.2018)
[19] *https://www.npmjs.com/package/xmldom* (accessed 09.07.2018)
[20] *https://www.npmjs.com/package/csv-parse* (accessed 09.07.2018)

4. The list of items that are iterated

5. The query instruction

The first four parameters are gradually applied in the business logic of the JRML processor. Whenever new information is available to the JRML processor, the query function is enriched and the new, enriched query is passed to the next processing steps.

When the query is enriched with all but the last parameters and is called with the query instruction, the query is executed and returns the result.

When a function is defined in the JRML mapping configuration in place of a query instruction, the query function is passed to that function as parameter in the state of the forth enrichment (before applying the query instruction). When instead of a function a string is given, the string is used as query instruction and the query function in its forth enrichment state is executed with the given string as instruction.

5.3.7 The iterator

The iterator consists of a set of functions which depend on the JRML mapping configuration, a query enriched with the query language and a datasource. The iterator retrieves items from the datasource through the iteration instruction defined in the configuration and performs an iteration over those items. A callback function passed to the iterator is called for each iteration step with a context object providing the datasource, the current iteration item, the current iteration step (number) and a query enriched with the iteration step and the list of items iterated.

5.4 Integration of JRML into TripleWave

JRML is a standalone node module accepting a mapping configuration and an optional collector object. The default collector object of JRML provides the stream items in the JSON-LD format, which is the output format of TripleWave and thus, no additional collector is needed.

For the configuration of TripleWave to use JRML, I extend the input possibilities for the *source* key with the additional value of *jrml*. When the *source* is set to *jrml*, TripleWave requires the additional key named *jrml* to be defined. The key *jrml* accepts as value a JRML mapping configuration.

Figure 5.10 shows the dataflow of TripleWave with the integration of JRML in contrast to the current dataflow of TripleWave shown in Figure 5.9. With JRML integrated into TripleWave, a new branch is added to the dataflow including the scheduler which retrieves (pulls) the datasets form the Web and the JRML transform applied before the publication of the RDF output stream.

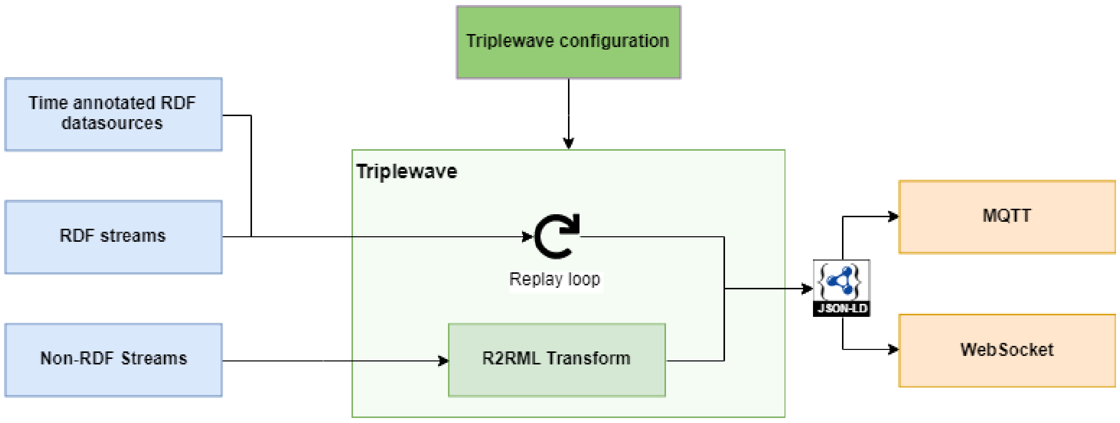


Figure 5.9: The dataflow in the current version of TripleWave

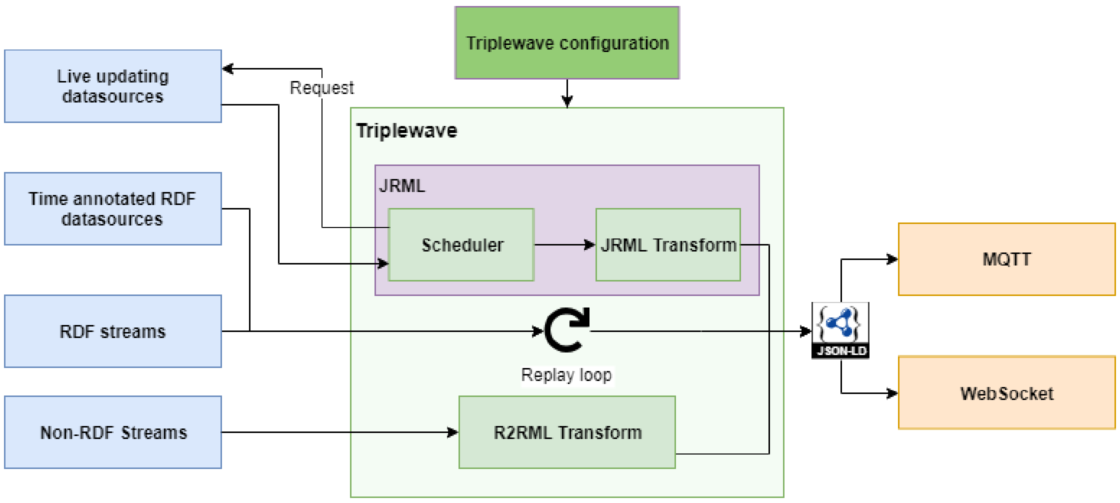


Figure 5.10: The anticipated dataflow after the implementation

In the following list, I evaluate if the state of TripleWave with the addition of JRML fulfills the requirements listed in Sections 3.4.4 and 3.4.5.

- **RD1:** With the integration of JRML into TripleWave, the requirement RD1 is *fulfilled* as TripleWave can now use frequently updated datasets as input source through the use of JRML.
- **RD2:** JRML maps the datasets to RDF, in the case of TripleWave to the RDF representation JSON-LD. The requirement RD2 is therefore *fulfilled*.
- **RD3:** JRML integrates a scheduler that can be configured with an interval or with a cron instruction. The requirement RD3 is therefore *fulfilled*.
- **RD4:** The data retrieval of JRML pulls the datasets from the Web, timed through the Scheduler. The requirement RD4 is therefore *fulfilled*.

- **RT1:** I evaluated and deployed the most complex of the datasets listed in Section 3.3 and listed by Muntwyler (2017) which are published in the formats CSV, JSON or XML. The datasets which are more than 50MB in size can be transformed but if the dataset is updated in intervals shorter than the download speed, the latest stream items must be grouped and the retrieval interval increased. The requirement RT1 is therefore fulfilled.

- **RT2:** Entities in the output stream can be defined through subjects in JRML mapping configurations. JRML allows to define multiple subjects per iteration (see Section 5.2.5). The requirement RT2 is therefore *fulfilled*.

- **RT3:** The properties of an entity can be computed from the input source through predicate object maps in JRML. The advanced transformations (see Section 5.2.6) allow to apply arbitrary transformations on the input data.

- **RT4:** JRML outputs RDF. The RDF representation that JRML generates can be changed through the usage of a custom collector (see Section 5.3.3), but the default format is JSON-LD. The requirement RT4 is therefore fulfilled.

6

Deployment

In this chapter, I present the datasets which I deploy one by one and describe decisions connected to the respective dataset. Further, I show a table with the RDF triples for each dataset.

Every dataset is deployed within an RDF graph following the format proposed by Barbieri and Valle (2010): *http://ex.org/%stream-name%/URLencode(%timestamp%)*. The timestamp must be in the *ISO 8601* format (Klyne and Newman, 2002)

6.1 Choosing datasets for deployment

To filter the datasets for the deployment, I used three criteria:

1. The representation of the published dataset (should be JSON, XML and CSV – which are the formats supported by JRML)
2. How new entries are created in the dataset
3. The special characteristics of the dataset which make a purely declarative mapping solution fail (complex characteristics preferred)

The third filter criterium takes various specialties of the datasets into account which make mapping the dataset through a generalized, declarative, mapping language like RML impossible.

The list of all datasets consist of 41 datasets. 25 datasets were collected by Muntwyler (2017) and 16 datasets by me (see Section 3.3). After applying the first criterium, 35 datasets are left. Six datasets are not published as CSV, XML or JSON. After applying the second criterium, 10 datasets are left: the other 25 follow a static update strategy and for three datasets, no data is available anymore. Three datasets (MAQMZH, TAQMZH and HAQMZH) have the exact same layout and update strategy - the only difference is the time interval between the updates. The overlap in layout and update strategy reduces the number of datasets to eight because I choose only one of the three datasets – which is HAQMZH. I choose HAQMZH out of the three because it is the fastest updating dataset. The datasets FAU and GAU also overlap, NTAU and TCZH show the same layout and update characteristics which reduces the number of datasets to 6. Three of the seven datasets (ADSBB, PTZH and RDE) left are updated frequently by adding new entries to the bottom of the document which is in the CSV format for both datasets. The files instances stay the same but their size keeps growing. All three datasets are above 200 MB in size. The size together with an update strategy of adding new entries at the bottom of the document lead to a nearly unusable datasets when delivering real-time data. Even though they can possibly be used, it is most likely that about 200 MB of unneeded entries have to be loaded everytime a new entry is added, only to put one new item on the stream. This is highly inefficient. For the purpose of this thesis, I exclude datasets with a size bigger than 50 MB for sparing network load on the deployment machine. I am left with three datasets: HAQMZH (CSV), FAU (CSV), NTAU (CSV).

The datasets left have all different layouts and characteristics. It can be seen that most datasets that have special characteristics are published as CSV. There is only one JSON dataset which is not updated statically and no XML dataset.

I include the dataset PKDE in the range of datasets that I deploy to test the XML support. Further, the dataset WWAU does not show special update characteristics but is interesting in terms of the third filter criteria. It is the only dataset for which multiple subjects per iteration are needed.

I end with five datasets that have to be deployed and which show the most complex mappings throughout the datasets. The final list of datasets is: HAQMZH (CSV), FAU(CSV), NTAU(CSV), WWAU(JSON), PKDE(XML)

6.2 The WWAU dataset

I showed the structure of the dataset WWAU and how it is relevant to the outcome of JRML in the Sections 5.1 and 5.2. The main characteristics of the WWAU dataset is its need for multiple subjects per iteration as stated in Section 5.1.4. JRML achieves the mapping of multiple subjects per iteration through an array of mapping definitions defined the *eventItem* entry of JRML (see Section 5.2.5). Each element in the array describes a subject and a *predicateObjectMap*. The subject is one MBA (see Section 5.1.3). The timestamp in the resulting graph must be constructed through the current date taken from the system (see Section 5.2.6) which is achieved through a new advanced transformation mapping (see Figure 5.5). I define the *predicateObjectMap* with the predicate *hasDuration* and an *objectMap* referencing the waiting time in minutes, in the *xsd:duration* format[1] which I add as datatype. The query must be a transformation function for adding the identifiers for the duration format in minutes: `query =>"PT" + query(pred) + "M"`.

Because the dataset is updated continuously with no delay, I choose to use the *interval* option of the scheduler with a value of 2 seconds (2000 milliseconds).

Only one property (*hasDuration*) is defined in the mapping. The *MBA identifier* refers to the name of the MBA as given by the dataset. Table 6.1 shows the mapping of the property formally.

[1] *http://www.datypic.com/sc/xsd/t-xsd_duration.html* (accessed 10.07.2018)

To describe data with no vocabulary definitions, I create new definitions in the namespace *http://streamreasoning.org/{datasetname}/* where the *dataset name* is replaced with the name of the dataset in lower-case – here *wwau*.

Subject	**Predicate**	**Object**
srwwau:{MBA name}	time:hasDuration	{waiting duration} ^^xsd:duration

Table 6.1: The mapping of the WWAU dataset. The prefix *srwwau:* refers to the namespace *http://streamreasoning.org/wwau/*, the prefix *time:* to the namespace *http://www.w3.org/2006/time#*.

6.3 The FAU dataset

The FAU dataset holds hourly average values of the particulate matter PM10 measured through the air quality monitoring network of north-east Austria (NUMBIS).[2]

The speciality of the dataset FAU is that new entries are added on the columns. There is a preset of 24 entries (columns), one for each hour of the day. If no data is given for a certain hour, the field holds a value of *-999*. Each row in the CSV except for the header describes the location at which the measurement is taken.

Two existing vocabularies about air pollution are the OWL vocabulary for weather[3] and the vocabulary presented by Galárraga et al. (2017). The OWL weather ontology defines the property *hasAirPollution* but restricts its domain to a *WeatherState* which the FAU dataset provides no information about. The ontology of Galárraga et al. (2017) enables talking about the air pollution through the notion of an observation. In the FAU dataset, one measurement of the air pollution at one station is one observation. I use the ongology defined by Galárraga et al. (2017) to describe the RDF output of the transformation as listed in Table 6.2

The reference to the measurement in the dataset must make use of the advanced tranformation functions because the current value has to be read for each subject from the column that holds the data of the last hour. The column names have the format *WertHH*, where *HH* is a two-character number for the hour in which the measurement was taken.

The dataset is updated hourly, however, entries are only available each hour from two hours before. The dataset must thus be retrieved each hour and the column with the data of two hours before must be mapped to RDF and published. I thus set the cron job to the same configuration as for the dataset HAQMZH (*2 * * * **).

[2] *https://www.data.gv.at/katalog/dataset/8b057f32-1312-40ae-ae51-9aa0a0d372ca* (accessed 12.07.2018)

[3] *https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/WeatherOntology.owl#hasValue* (accessed 18.07.2018)

Subject	**Predicate**	**Object**
air:observation/AU{timestamp}#{station}	air:schema/ station	srfau:{station}
air:observation/AU{timestamp}#{station}	air:schema/pm10	{measurement} ^^xsd:decimal

Table 6.2: The mapping of the FAU dataset. The prefix *srfau:* refers to the namespace *http://streamreasoning.org/fau/*, the prefix *air:* to the namespace *http://qweb.cs.aau.dk/airbase/data/*.

6.4 The HAQMZH dataset

The HAQMZH dataset holds hourly updated measurements of the air quality of the last seven days at the stations Stampfenbachstrasse, Schimmelstrasse, Rosengartenstrasse and Heubeeribüel in Zürich.[4]

Out of the five datasets for deployment, the HAQMZH dataset needs the most special treatment (see Section 5.2.7 on TAQMZH). TAQMZH and HAQMZH have the same layout and bring therefore the same problems.

The dataset is published in the encoding *latin1* which I can set through the *encoding* property in the *logicalSource* of the JRML mapping. As the dataset is updated each hour (ten minutes after the full hour), the *cron job* for the scheduler must be set to a value right after. I choose every hour 11 minutes after the full hour: *11 * * * ** (in cron job notation).

Following the discussion in Section 5.2.7 about the TAQMZH dataset, the datset HAQMZH needs a full source transform which is defined through the *transform* property in the *logicalSource* that maps the source such that the first row in the new source is the header and the rows are the data that conform to the columns defined in the header.

I choose the same strategy as for the FAU dataset to map the values of the dataset to RDF. The mappings are listed in the Table 6.3.

[4] *https://data.stadt-zuerich.ch/dataset/luftqualitaet-stunden-aktuelle-messungen* (accessed 12.07.2018)

Subject	**Predicate**	**Object**
srhaqmzh:observation/AU{timestamp}#{station}	srhaqmzh: place	srhaqmzh: {place of measurement}
srhaqmzh:observation/AU{timestamp}#{station}	srhaqmzh: description	{description}
srhaqmzh:observation/AU{timestamp}#{station}	srhaqmzh: measuremen-tUnit	{measurement unit}
srhaqmzh:observation/AU{timestamp}#{station}	srhaqmzh: molecule	{measured molecule}
srhaqmzh:observation/AU{timestamp}#{station}	srhaqmzh:value	{value}

Table 6.3: The mapping of the HAQMZH dataset. The prefix *srhaqmzh:* refers to the namespace *http://streamreasoning.org/haqmzh/*, the prefix *air:* to the namespace *http://qweb.cs.aau.dk/airbase/data/*.

6.5 The NTAU dataset

The dataset NTAU holds hourly measurements of the hydrographic service of Tirol. The measurements include waternames, position information and height reference.[5] This dataset must be retrieved after the full hour to get the latest measurements. I choose *2 * * * * ** like for the dataset FAU.

There are two specialities regarding the dataset NTAU. The first speciality is that it has a structural error. The first linebreak in the document is missing which makes the first row with data end up on the same line with the header. Through this error, the dataset cannot be parsed without a full source transform applied with the *transform* property on the *logicalSource* which automatically inserts the missing linebreak before parsing.

The second speciality is the addition of new entries to the dataset at various places in the document. For each station/water for which the dataset shows the water level exist measurements from different times during the last two days. The stations are organized in groups and the newest entries are inserted at the top of the entries which group they belong to.

To iterate only the newest entries in the dataset, the iterator must be a function which groups the entries belonging to one station together and gets the newest entry for each group. The found elements must be returned by the iterator for that the iteration process iterates only those items.

I could not find any existing vocabulary matching the requirements for describing the

[5] *https://www.data.gv.at/katalog/dataset/44720e90-c2de-497b-8162-3810206dd011* (accessed 12.07.2018)

NTAU dataset.

Table 6.4 shows the output triple layout of the mapping.

I choose to use the same strategy as for FAU and denote an observation as the subject to which the measurement value and metadata are linked.

Subject	**Predicate**	**Object**
srntau:observation/AU{timestamp}#{station}	srntau:water	srntau:{water}
srntau:observation/AU{timestamp}#{station}	srntau:rainfall	{rainfall} ^^xsd:decimal
srntau:observation/AU{timestamp}#{station}	srntau: measurementUnit	{unit}
srntau:observation/AU{timestamp}#{station}	srntau:seaLevel	{sea level} ^^xsd:positiveInteger
srntau:observation/AU{timestamp}#{station}	srntau:station	srntau:{station}
srntau:observation/AU{timestamp}#{station}	srntau:easting	{easting} ^^xsd:decimal
srntau:observation/AU{timestamp}#{station}	srntau:northing	{northing} ^^xsd:decimal
srntau:observation/AU{timestamp}#{station}	srntau: EPSGCode	{epsg code}

Table 6.4: The mapping of the NTAU dataset. The prefix *srntau:* refers to the namespace *http://streamreasoning.org/ntau/*.

6.6 The PKDE dataset

The PKDE dataset shows the current occupancy of the parking lots in the city area of Kleve (DE). It is continuously published, so I choose an interval of two seconds for the retrieval to get the updates fast without putting too much load on the network.

The PKDE dataset has itself no speciality but it is the only dataset published as XML. The data queries for the dataset PKDE are XPath expressions. No advanced transformations have to be applied to map the dataset PKDE with the exception of the timestamp. The timestamp is not in *ISO 8601* format and therefore needs to be transformed. An interesting note is when using an advanced transform function, the query function returns XML DOM nodes. A selected textnode has to be transformed to a Javascript string through `.toString()`. The same applies to the transformation of the timestamp.

A vocabulary for describing parking lots exists and is described at *http://vocab.datex.org/terms/#*. I use it for mapping the information of the dataset PKDE as shown in Table 6.5

Subject	**Predicate**	**Object**
srpkde:{parking name}	dtx:parkingName	{parking name}
srpkde:{parking name}	dtx:parking-NumberOfSpaces	{max parking spaces} ^^xsd:positiveInteger
srpkde:{parking name}	dtx:parking-NumberOfVacantSpaces	{number of free parking spaces} ^^xsd:positiveInteger
srpkde:{parking name}	dtx:parking-SiteOpeningStatus	{parking open state}

Table 6.5: The mapping of the PKDE dataset. The prefix *srpkde:* refers to the namespace *http://streamreasoning.org/pkde/*, the prefix *dtx:* to the namespace *http://vocab.datex.org/terms/#*.

7 Conclusions

This thesis builds on the research of Muntwyler (2017) and targets the transformation of 3-star datasets to RDF for their publication as data streams on the web. My focus is on OGD datasets: I examined a list of 16 datasets in addition to the datasets found by Muntwyler (2017). As Muntwyler (2017) focused on the Swiss OGD portal, I extend the list of datasets of the Austrian and German OGD portals.

For making the publication of Linked Data as data streams on the web more accessible and to provide a streamlined solution, I changed the TripleWave configuration from using *.properties* files to using *.js* files and prepared TripleWave for global installations through *yarn* and *npm*. It is now possible to run multiple TripleWave instances in parallel through the definition of an array of configurations in the TripleWave configuration file.

I extend TripleWave such that 3-star datasets can be defined as source together with a mapping declaration which maps values from the source to an RDF output. TripleWave receives an integrated solution for publishing Linked Data as data streams through this extension.

I found that existing Linked Data mappings rely on declarative definitions which makes them language-independant. The drawback is limited expressivity: source data can only be mapped as-is into the target source or interpolated into new values.

The published OGD datasets I inspected in this thesis varied greatly in layout and structure. The multitude of structures, especially in CSV datasets, complicated finding a concensus for declarative mapping definitions to a degree that replacing a declarative solution with a solution in a programming language became a valid and necessary option. The OGD datasets built the basis for the requirements which a mapping solution has to meet.

I introduce JRML, an RDF mapping solution based on RML. JRML uses the same definitions as RML and makes slight changes to the structure. Instead of using the declarative RDF syntax *Turtle*, JRML is written in Javascript which allows for the application of advanced transformations.

Opposed to RML, the output of JRML is a data stream which is controlled by an integrated scheduler. JRML mapping definitions include a scheduler definition which configures the scheduling mechanism. The source encoding can be defined through a new *encoding* definition, subject aggregation is made possible through an *aggregation* definition. Multiple subjects per iteration can be defined due to using an array of subjects with their predicate-object mappings. The additional property *transform* allows source text transforms before the internal parsing takes place.

The main feature of JRML is the possibility to define a Javascript function in place of a query definition with which advanced transformations can be applied. The exact layout of the output values can be controlled through using transformation functions where RML allowed only string interpolation through embedding queries into a string.

The use of JRML is limited to 3-star data. Data which is not available as structured text cannot be parsed and currently, only JSON, CSV and XML are supported. Not supported are data joins as specified in RML due to the usage of JRML in data streams. Data items in data streams are bound to time which makes joining datasets with different expansion in time unnecessary. Further, linking different datasets together is only possible by applying identifying URIs.

JRML meets the requirements and provides a tool for mapping even the most challenging datasets. Even though it cannot be implemented the same way in other programming languages, it serves the use-case better than a declarative mapping like RML. The usage of a self-contained mapping declaration instead of a declarative mapping together with external transformation hooks through the programming language prevents from unexpected side-effects and improves the readers interpretation of the mapping.

I deployed five datasets with JRML through TripleWave, all with different characteristics and challenges concerning the mapping solution. An interesting observation is that the publishers of the OGD datasets do not have guidelines on the presentation of their data which makes it possible that datasets have various structures. In one instance, I even encountered a structural error in the CSV file of the dataset NTAU which could only be fixed with a full, textual source transform.

With JRML and its integration into TripleWave, more Open Data datasets can be published with much less effort and with the first dedicated solution for mapping frequently updated datasets to an RDF data stream. With the deployment of five OGD datasets, I increased the number of datasets on the web and more datasets can be added.

In the future, JRML can be extended to support more query languages for the supported data types, for example JMESPath for JSON.[1] I created JRML for mapping frequently updating datasets to RDF data streams. Further work could test a usage of JRML outside of the field of data streams, for example for the mapping of static 3-star datasets. The JRML processor can be used in place of RML with an extension to JRML which implements RML-like data joins and the integration of SQL as query engine. The advanced transformations of JRML can allow more expressive mappings than RML not only in the field of data streams but also for non-continuous mappings of 3-star datasets. An integration of JRML on top of TripleWave into deployment pipelines of software systems as an intermediate mapping transformer could leverage the publication of Linked Data as part of automated deployments. The addition of a file-watcher and stream subscriptions to complement the scheduler would allow the integration into enterprise systems.

[1] *http://jmespath.org* (accessed 21.07.2018)

References

Barbieri, D. F. and Valle, E. (2010). A Proposal for Publishing Data Streams as Linked Data. In *Linked Data on the Web Workshop*.

Berners-Lee, T. (2006). Design Issues: Linked Data. *http://www.w3.org/DesignIssues/LinkedData.html.*

Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked Data – the Story so Far. *International journal on semantic web and information systems*, 5(3):1–22.

Bizer, C. and Seaborne, A. (2004). D2RQ-Treating Non-RDF Databases as Virtual RDF Graphs. In *Proceedings of the 3rd international semantic web conference (ISWC2004)*, volume 2004. Proceedings of ISWC2004.

Brickley, D. (2004). RDF Vocabulary Description Language 1.0: RDF Schema. *http://www.w3.org/TR/rdf-schema/.*

Carroll, J. J., Bizer, C., Hayes, P., and Stickler, P. (2005). Named Graphs, Provenance and Trust. In *Proceedings of the 14th international conference on World Wide Web*, pages 613–622. ACM.

Das, S., Sundara, S., and Cyganiak, R. (2012). R2RML: RDB to RDF Mapping Language, W3C Recommendation 27 september 2012. *Cambridge, MA: World Wide Web Consortium (W3C)(www. w3. org/TR/r2rml).*

Dell'Aglio, D., Le Phuoc, D., Le-Tuan, A., Ali, M., and Calbimonte, J.-P. (2017). On a Web of Data Streams. In *Proceedings of the ISWC2017 workshop on Decentralizing the Semantic Web, Vienna, Austria*, pages 21–22.

Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., and Van de Walle, R. (2014). RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *Proceedings of the 7th Workshop on Linked Data on the Web.*

Galárraga, L., Mathiassen, K. A. M., and Hose, K. (2017). QBOAirbase: The European Air Quality Database as an RDF Cube. In *International Semantic Web Conference (Posters, Demos & Industry Tracks).*

Klyne, G. and Newman, C. (2002). Date and Time on the Internet: Timestamps. Technical report.

Mauri, A., Calbimonte, J.-P., Dell'Aglio, D., Balduini, M., Brambilla, M., Della Valle, E., and Aberer, K. (2016). Triplewave: Spreading RDF Streams on the Web. In Groth, P., Simperl, E., Gray, A., Sabou, M., Krötzsch, M., Lecue, F., Flöck, F., and Gil, Y., editors, *The Semantic Web – ISWC 2016*, pages 140–149, Cham. Springer International Publishing.

McGuinness, D. L., Van Harmelen, F., et al. (2004). OWL Web Ontology Language Overview. *W3C recommendation*, 10(10):2004.

Muntwyler, P. (2017). Increasing the Number of Open Data Streams on the Web.

Sedira, Y. A., Tommasini, R., and Della Valle, E. (2017). Towards VoIS: a Vocabulary of Interlinked Streams.

Sequeda, J. F. and Corcho, O. (2009). Linked Stream Data: a Position Paper. In *Proceedings of the 2nd International Workshop on Semantic Sensor Networks, SSN 09*. CEUR-WS.

Taelman, R., Heyvaert, P., Verborgh, R., and Mannens, E. (2016). Querying Dynamic Datasources with Continuously Mapped Sensor Data. In *Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016)*.

Vandenbussche, P.-Y., Atemezing, G. A., Poveda-Villalón, M., and Vatant, B. (2017). Linked Open Vocabularies (LOV): a Gateway to Reusable Semantic Vocabularies on the Web. *Semantic Web*, 8(3):437–452.

A

Appendix

A.1 Run instructions

In this section, I explain how to run TripleWave with JRML as input source and mosquitto as target of the output stream. The instructions are targeted mainly to linux systems. The tools/packages that are required to be installed are:

- NodeJS version >= 8.3.0
- *npm*
- *mosquitto*[1]
- *python* (For building dependencies installed by *npm install*)
- *build-essential* (For building dependencies installed by *npm install*)
- *mosquitto-clients* (For testing / subscribing to the deployed datasets)

Following I line out all steps to get a running TripleWave:

1. `cd` into the folder *TripleWave* and run either `npm install` or `yarn` depending on the package management tool of choice to go sure that all dependencies of TripleWave are installed.
2. Repeat (1) for the folders *TripleWaveConfig* and JRML.
3. `cd` into the folder *jrml* and run `npm link`
4. `cd` into the folder *TripleWave* and run `npm link` to make TripleWave available globally on the system. The linking is, like for step (4), necessary as long as TripleWave is not yet published on the NPM registry.

[1]Find the installation instructions for your system at *https://mosquitto.org/download/* (accessed 14.07.2018).

5. `cd` into the folder *TripleWave* and run `npm link jrml`. The linking is important for establishing the dependency as long as JRML is not yet published on the NPM registry[2]

6. Startup *mosquitto* if it is not yet running. On a Linux system with systemd, *mosquitto* can be started with `sudo service mosquitto start`.

7. *cd* into the folder TripleWaveConfig and run `triplewave`.

If an error appears saying that the module *jrml* was not found, linking JRML to TripleWave has failed. In this case try to repeat steps three to five. If the command *triplewave* is not found on your system, running `npm link` from the directory *TripleWave* has failed. Try to repeat step five.

A TripleWave instance should be running by now. The mosquitto topic to which TripleWave publishes the streams, moquitto can be subscribed to with the command `mosquitto_sub -t 'pkde'` from the package *mosquitto-clients*. In the command shown before, *'pkde'* is the topic to which the subscription applies. The deployed datasets are available by subscribing to the topic with the short name of the dataset.

A.1.1 The command TripleWave

The global command `triplewave` can be called with additional flags. Available flags are:

- **--configuration** or **-c**
  Specifies the configuration file which should be chosen.
  Defaults to *./tw_config.js*

- **--print** or **-p**
  If set, TripleWave prints the stream output to the console

- **--index** or **-i**
  Specifies the index of the configuration with which TripleWave should be started. It refers to the index of the configuration in the array exported by the configuration file.
  If not specified, TripleWave starts all configurations in child processes.

[2] *https://www.npmjs.com* (accessed 14.07.2018)

List of Figures

List of Tables