



**University of
Zurich^{UZH}**

Creation of a Catalog of Web Streams

Thesis

July 5, 2019

Deniz Sarici

of Zurich ZH, Switzerland

Student-ID: 13-935-002

deniz.sarici@uzh.ch

Advisor: **Daniele Dell'Aglio**

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

Acknowledgements

I would like to thank Dr. Daniele Dell'Aglione for his guidance and constructive feedback throughout the writing of my thesis. I'm grateful for the support and opportunity to have worked with him and was able to learn a lot during the last few months.

Zusammenfassung

Daten werden vermehrt als Datenstrom veröffentlicht und sind nicht mehr nur in statischer Form verfügbar. Da Daten zusätzlich in immer kürzeren Abständen generiert werden, wird das Speichern und Verarbeiten von Daten erschwert. Im Zusammenhang mit Linked-Data gibt es Bemühungen deren Prinzipien für statische Daten, auf dynamische Daten zu übertragen. In einem ersten Schritt wurden RDF-Streams kreiert, um die Publikation von Streams über das Internet zu modellieren. Dazu wurde das Framework TripleWave entwickelt, um das Streamen von diesen RDF-Daten zu ermöglichen. TripleWave veröffentlicht neben den Daten ebenfalls Metadaten, die dem Anwender Informationen über die gestreamten Datensätze zur Verfügung stellt. Doch bisher hat TripleWave kein standardisiertes Vokabular verwendet. Um diese Lücke zu beheben, integrieren wir das Vokabular VoCaLS in TripleWave. VoCaLS ist ein Ansatz, um Streaming-Services wie TripleWave mit standardisierten Termen beschreiben zu können.

Die Technologie für das Streamen von Datensätzen ist schon reif genug. Jedoch fehlt es an einem Katalog, der das Entdecken von solchen Streams ermöglicht. Deshalb entwickeln wir einen Katalog für Web-Streams und folgen dabei den Linked-Data-Prinzipien. Um die Funktionalität des Kataloges zu demonstrieren, streamen wir Datensätze mit TripleWave und katalogisieren diese. Dazu erweitern wir TripleWave um die programmierbare Interaktion mit dem Katalog.

Abstract

Data is increasingly published as a stream of data and is no longer published as only as a static dataset. Moreover, the data is being published in shorter intervals, making the storage and processing of data more difficult. There's an ongoing effort in aligning Linked Data principles for static data with the streaming dynamic data. RDF streams were created in a first step to model the publication of streams over the web. TripleWave is a framework to publish and manage those RDF streams. TripleWave also exposes metadata about itself and the data being streamed. But TripleWave initially used ad-hoc vocabulary that was not based on existing vocabularies. Therefore, we modify TripleWave to adopt the VoCaLS vocabulary for its metadata. VoCaLS proposes a standardized way for describing services for the publication, consumption and processing of web streams.

The technology for streaming datasets is in place. However, there's a lack of discoverability for web streams. Thus, we create a catalog of web streams that serves as a portal for users and machines to discover streams on the web. To demonstrate the capabilities of the catalog we search and select suitable datasets to be streamed and put them into the catalog. We use TripleWave to publish those streams and extend TripleWave to interface with the catalog.

Table of Contents

1	Introduction	1
2	Related work	3
2.1	Open Data	3
2.2	Linked Data	5
2.2.1	RDF	6
2.2.2	RDF Serialization	7
2.2.3	Applications	9
2.3	Publication	9
2.4	Streaming Linked Data	10
3	Requirements	13
3.1	TripleWave	13
3.1.1	TripleWave Interface	13
3.1.2	Streaming Datasets with JRML	14
3.2	Catalog	15
3.2.1	Functional Requirements	15
3.2.2	Non-functional Requirements	17
4	Architecture	19
4.1	Overview	19
4.2	Publishing data streams	20
4.3	Consuming TripleWave Streams	20
4.4	Message Brokers	21
4.4.1	Kafka Producer	22
4.4.2	Kafka Output	22
4.4.3	MQTT Broker	23
4.5	CKAN	23
5	Changes to TripleWave	25
5.1	Vocabulary	25
5.2	Pushing the metadata to the catalog	28
5.3	TripleWave Config	30
5.3.1	Advertise endpoints	30

5.3.2	Adding Catalog Information	31
5.4	Changes to JRML	31
5.4.1	Discard invalid values	31
5.4.2	Update data source URL	33
6	Creation of the Catalog	35
6.1	Requirements Analysis	35
6.2	Extending CKAN	36
6.3	Creating a catalog entry	37
6.3.1	Creating Additional Fields	37
6.3.2	Mapping to CKAN fields	40
6.3.3	Displaying the Stream Descriptor	40
6.3.4	Machine-readable Version	43
6.3.5	Creating a preview of the Stream	43
6.4	Additional Modifications	45
6.4.1	Accessibility	45
6.4.2	About page	45
6.4.3	Visual Appearance	45
6.5	Deploying the Catalog	47
7	Datasets	49
7.1	Survey	49
7.1.1	Updating previous surveys	49
7.1.2	Adding new Datasets	50
7.2	Deployment	52
7.2.1	Selection	54
7.2.2	Railway Traffic Information	56
7.2.3	Air Quality Zurich (AQZH)	56
7.2.4	Weather Stations Zurich (WSZH)	57
7.2.5	Locations and Availability of Charging Stations (LACS)	57
8	Conclusions	59
A	Appendix	67
A.1	CKAN	67
A.1.1	Install CKAN from source	67
A.1.2	Install the CKAN plugin	67
A.1.3	Create User and API Key	68
A.1.4	Solr	68
A.2	Kafka	70
A.3	Creating the web streams	70
A.3.1	TripleWave and JRML	71
A.3.2	MQTT Broker	71
A.3.3	Kafka Connectors	71

Introduction

The Web of Data is about publishing data on the web and linking points of data together. Such links may connect data points within a dataset or across two different datasets. Connecting data through links allows a person or machine to search for related data (Berners-Lee, 2006). The data can be accessed and traversed by following the HTTP URIs in the resource description. The Resource Description Framework (RDF)¹, serves as a standard format for publishing data on the web. RDF uses URIs to describe the relationship between two resources.

One example for publishing data in this way is the Linking Open Data cloud². It exposes over a thousand interconnected datasets that are managed and published by different organizations (Dell’Aglío et al., 2017). Barbieri and Della Valle (2010) observe that data on the Web is increasingly represented as a stream of data rather than a static dataset. A stream is a sequence of data elements and is often paired with corresponding timestamps to order the described events. Barbieri and Della Valle (2010) regard this kind of data as transient and do not propose to store the data indefinitely. To consume a data stream, the stream is continuously queried over time. Particularly in the context of the Internet of Things (IoT) data elements are created in very short time intervals and often lose their value with time (Dell’Aglío et al., 2017). Mauri et al. (2016) propose the framework TripleWave to publish and manage data streams on the web. TripleWave fetches and transforms data sets into RDF graphs and pushes them to registered consumers.

In parallel, VoCaLS (Tommasini et al. (2018)) defines a vocabulary to describe and advertise such data streams on the web. A common vocabulary for describing access points of web streams and their capabilities allows decentralized and automated discovery. We therefore use VoCaLS to describe streaming services such as TripleWave in a unified way. This is helpful for cataloging and monitoring those services on the web (Tommasini et al., 2018).

The goal of this thesis is the creation of a catalog of web streams for hosting stream descriptions in a central repository. The catalog serves as a portal for users to discover web streams and find out how to connect to them. Each entry in the catalog contains information about the streaming service and metadata about the data being streamed.

¹<https://www.w3.org/RDF/> (accessed 2019-06-03)

²<https://lod-cloud.net/> (accessed 2019-06-03)

The view of the catalog is available as a HTML page and as unformatted, serialized RDF text. To further increase accessibility to the data, we offer a preview of the stream directly through the browser. We present a list of requirements for the catalog and peripheral systems in Chapter 3.

There are already systems for hosting metadata on the market. However, they do lack the capability of handling linked data out of the box and are built for static datasets primarily. Furthermore, we need to make the interface of the catalog accessible to TripleWave. To create the catalog of web streams we extend CKAN³: a tool that's used for hosting Open (Government) Data websites.

In order to populate the catalog with stream entries we use TripleWave to publish the web streams. We modify TripleWave to describe the stream and the dataset itself using VoCaLS and other existing vocabulary. The second feature we to add TripleWave is the automatic registration of the stream through the catalog's API. We summarize the changes to TripleWave in Chapter 5. The creation of the web streams is based on previous work of Muntwyler (2017) and Bernhaut (2018). In the process of selecting datasets to be streamed, we update their surveys on Open Government Datasets and look for newly published datasets that are suitable for streaming. We show the results of the survey and the selection of datasets for streaming in Chapter 7. In order to fetch and transform the data we use the TripleWave module JRML Bernhaut (2018). For the distribution of the web streams we deploy Apache Kafka and reuse some of the connectors developed by Muntwyler (2017). Since TripleWave supports the MQTT protocol, we also set up a MQTT message broker in the backend. We explain the setup in Chapter 4. Finally we deploy TripleWave, the message brokers and the catalog on a virtual machine on the servers of the university.

³<https://ckan.org/about/> (accessed 2019-06-03)

Related work

This section is about the related work. We start by introducing the principles and concepts of open data and then follow up with looking at the open government data initiative. Afterwards we will look at linked data in general and cover its application with open data. The latter is also referred to as linked open data (LOD). The second part will describe the components for publishing and streaming linked data. This includes the vocabularies used and some of the technology to enable the spreading of linked data on the web. We finish with presenting the supporting tools used in this project. Those tools may not have been built to work in conjunction with linked data, but their functionalities match our requirements for the architecture. Amongst the described tools are Apache Kafka, the MQTT protocol and the Comprehensive Knowledge Archive Network (CKAN).

2.1 Open Data

Open data is a concept that “data and content can be freely used, modified, and shared by anyone for any purpose” ([Open Knowledge International, 2015](#)). The essential features of open data are

- **Availability and access:** the data must be easily accessible in modifiable form and should be downloadable via the Internet.
- **Re-use and redistribution:** the terms of use must allow re-use and redistribution of the data, including the mixing and aggregation of different datasets.
- **Universal participation:** anyone can participate in using and distributing the data

An important goal of the definition of open data is interoperability, meaning that data from different datasets can be mixed together in order to create better products and services ([Open Knowledge International, 2015](#)). [Murray-Rust \(2008\)](#) relates this term to a movement in the scientific community for making academic data freely accessible and allow the aggregation of such data without explicit permission required. In his example, scientists can access shared data about molecules and use the combined data to gain

more insights. Another emerging trend is the government publishing information and making it accessible to the public (Maali et al., 2010).

Open government data (OGD) builds on the concept that open data should be freely available to anyone and that the people can use and redistribute the data in any form. Most government owned institutions collect and produce data in the process of doing their work. The large volume of data gathered by the government makes this kind of data particularly interesting.

Nowadays, the data is primarily published on portals that act as entry points to all open datasets of the respective institution. Examples of such portals are data.gov¹ in the US, data.gov.uk² in the UK, opendata.swiss³ in Switzerland and also open data portals of municipalities, e.g. Zurich⁴. In most cases, there is also metadata published. That information may include publishers, the language of the data, further information about the data, its update interval, file formats and more (Maali et al., 2010).

Data published by governments is available for many categories such as administration, agriculture, finances and environment. The data itself comes in many formats, such as CSV, JSON, XML or sometimes in a less reusable format like PDF (Maali et al., 2010). On top of presenting a front end for the user, the portals may also offer (meta)data in formats that enable automatic machine processing, for example, RDFa or JSON-LD embedded into the webpage (Maali et al., 2010).

OGD helps the citizens understand how their governments work, increases transparency of the administrations and holds them accountable for its actions. Furthermore, the publication of data and eased access to it, may lead to further insights about the processes in the government (Shadbolt et al., 2012). Ubaldi (2013) additionally sees OGD supporting economic growth, as an opportunity for entrepreneurship built on the OGD and as a source of social innovation.

In order to increase the quality of OGD, an Open Data Maturity assessment has been started at EU level. The European Data Portal has been assessing and monitoring the various open data portals and policies in the countries of the European Union and some other European countries such as Switzerland (Carrara et al., 2017). Furthermore, the European Data Portal enables access to datasets by searching and filtering datasets directly⁵ and by searching metadata using SPARQL queries⁶.

OGD faces many challenges in making the data accessible. For improving the search over multiple datasets, there need to be common formats to enable such queries. In order to increase the interoperability of those datasets, Maali et al. (2010) propose a standardized interchange format for the machine-readable representation of the data. Such representation helps web crawlers to better interpret the web pages. For example, Google Search uses structured data to understand what a web page is about⁷. The

¹<https://www.data.gov/>

²<https://data.gov.uk/>

³<https://opendata.swiss>

⁴<https://data.stadt-zuerich.ch/>

⁵<https://www.europeandataportal.eu/data/en/dataset> (accessed 2019-03-13)

⁶<https://www.europeandataportal.eu/sparql-manager/en/> (accessed 2019-03-13)

⁷<https://developers.google.com/search/docs/guides/intro-structured-data> (accessed 2019-03-17)

supported formats are JSON-LD, Microdata and RDFa. A common vocabulary also allows the decentralized publishing of catalogs and can also be used as a manifest for digitally archived data (Maali et al., 2014).

A showcase of OGD is the interactive article published in the Swiss newspaper Tages-Anzeiger that shows the punctuality of the public transport in the city of Zurich⁸. The authors make use of the large dataset about arrival and departure times published in Zurich’s open data portal⁹. The datasets lists all target-actual arrival and departure times in a publicly accessible CSV files. Similarly, the Transport for London (TfL) freely releases datasets under the Open Government License for developers and other interested parties on their website¹⁰. Stone and Aravopoulou (2018) show in their case study how the TfL make their data available through public APIs, static files and direct feeds. The data includes for example real-time arrival-time, time tables and network performance. The data can then be consumed by client applications such as Google Maps, Apple Maps and Bus Times London¹¹. The availability of live data helps the TfL’s customers to plan and adjust their routes accordingly. This does not only save time and money, but also helps avoiding congestion in the network by showing capacity and expected utilization of the routes (Stone and Aravopoulou, 2018).

Another use case of OGD is the publication of data from environment sensors. Boyle et al. (2013) survey monitoring sensors across the city of London that publish data containing information about the weather, air quality, river levels, water quality, etc.

2.2 Linked Data

In the previous section we talked about approaches to enable the interoperability between different datasets. One of the solutions is based on principles of Linked Data to connect the datasets. First, we introduce the principles of Linked Data and then cover the related work in the field.

In the web there are lots of documents. Those documents can be accessed by following hyperlinks within the documents. To find relevant data, search engines are valuable resources since they index those documents and apply some algorithms to retrieve and rank the most relevant pages given a user query (Brin and Page, 1998). Whilst the initial web documents did not contain any structure or semantics, the adoption of Linked Data has enabled connecting data inside documents and between different documents (Bizer et al., 2011). Linked Data uses the web to make links between data elements. The data published under Linked Data principles is machine-readable, has a well-defined meaning and can be interconnected to other data elements (Bizer et al., 2011). In order to connect raw data on the web, Berners-Lee (2006) proposes a set of practices for publishing data

⁸<https://www.stadt-zuerich.ch/portal/de/index/ogd/anwendungen/2016/so-pnktlich-ist-ihre-vbz-linie.html> (accessed 2019-03-13)

⁹<https://data.stadt-zuerich.ch/dataset/vbz-fahrzeiten-ogd> (accessed 2019-03-13)

¹⁰<https://tfl.gov.uk/info-for/open-data-users/>. The License enables the users to use the data in their own software and services. Retrieved 2019-03-13.

¹¹<https://www.emeraldinsight.com/doi/full/10.1108/BL-12-2017-0035> (accessed 2019-03-13)

on the web:

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

The Uniform Resource Identifiers (URIs) represent an entity on the web, starting with *http://*. If two entities have the same URI, they represent the same resource. To get more information about a resource, its URI can be resolved over the HTTP protocol. An URI can also be used to identify and name a relationship between two entities (Bizer et al., 2011). URIs may also contain a hash sign (#). The part that is followed by the hash sign is also called *fragment identifier*. The fragment identifier refers to a term that is described in the linked document. In this case the URI references a real-world object, and not the web page itself (Heath and Bizer, 2011).

2.2.1 RDF

In addition to URIs, Linked Data uses the Resource Description Framework (RDF)¹² to describe the entities and their relationships in the web. RDF is graph-based data model and represents data as statements in the form of **subject – predicate – object**. A single statement is also called a triple, and a graph usually consists of a set of triples. The subject is a URI identifying the entity that the statement is about, and the predicate specifies how the subject and the object are related to each other. Finally, the object is either a URI representing a resource or a simple literal value (e.g. a string, a number or a date) (Heath and Bizer, 2011). The latter have an RDF literal as the object. Those kinds of triples can be used to describe the name of a person for example (Listing 2.1). Literals can be further divided into plain and typed. A plain literal is a string and an optional language tag, identifying the language used. Typed literals come with a datatype URI to describe the datatype of the literal (Heath and Bizer, 2011). In the case of RDF links, the triple consists of three URIs. The predicate URI is used to describe the relationship between the resources (Listing 2.2). Heath and Bizer (2011) further distinguish RDF links by the location of the subject and object. An internal RDF link references resources in the same data source, whereas an external RDF link connects the subject to an object in a different data source.

¹²<https://www.w3.org/RDF/> (accessed 2019-06-13)

```
http://example.org/#spiderman
http://xmlns.com/foaf/0.1/name
"Spiderman"
```

Listing 2.1: A literal triple with a plain string

```
http://example.org/#spiderman
http://www.perceive.net/schemas/relationship/enemyOf
http://example.org/#green-goblin
```

Listing 2.2: RDF (internal) Link

Additionally, a subject or an object may also be a blank node. Blank nodes do not identify specific resources (in contrast to URIs and literals) and are limited in scope. They make a statement about a resource that exists, but do not identify any particular thing¹³. Since blank nodes are limited in scope, it's not possible to use them for linking different datasets. It is also difficult to merge data from different datasets, since there is no common key to join on. Hence [Heath and Bizer \(2011\)](#) suggest avoiding the use of blank nodes where ever possible.

To further describe the meaning of objects and relationships, languages such as the RDF Schema Language (RDFS)¹⁴ are used. Such RDFS vocabularies contain definitions about classes and properties, for example the class *Dog* and the property *hasColor*. RDFS can also be used to describe the relationship between classes and between properties. We can for example specify that *Dog* is a subclass of *Animal* or that *hasColor* has the color class as range ([Heath and Bizer, 2011](#)).

[Bizer et al. \(2011\)](#) consider reusing terms from popular RDF vocabularies as a good practice. Some examples are Friend of a Friend (FOAF)¹⁵, SKOS¹⁶ and Dublin Core¹⁷. If two different URIs refer to the same thing, we can use the `owl:sameAs` relation between them¹⁸. [Shadbolt et al. \(2012\)](#) use this to increase linkage between different datasets that use different terms for the same thing.

2.2.2 RDF Serialization

To store and share RDF graphs we then require a serialization format adhering to the RDF syntax. An example of serialization is RDF/XML¹⁹. However, the syntax is regarded as difficult to read for humans.

¹³<https://www.w3.org/TR/2014/REC-rdf11-mt-20140225/#blank-nodes> (accessed 2019-03-17)

¹⁴<https://www.w3.org/TR/2014/REC-rdf-schema-20140225/>

¹⁵<http://xmlns.com/foaf/spec/>

¹⁶<https://www.w3.org/TR/swbp-skos-core-spec/>

¹⁷<http://dublincore.org/>

¹⁸<https://www.w3.org/TR/owl-ref/#sameAs-def>

¹⁹<https://www.w3.org/TR/rdf-syntax-grammar/>

Another format is RDFa²⁰. It allows to embed RDF triples in HTML documents as attributes, without affecting the rendered view the users get. In the following example, the `h2` and `p` tags are annotated with property attributes to give machines further information about the kind of information they enclose. The `about` attribute is used to name the subject of the statements.

```
<div about="http://example.org/posts/trouble"
  <h2 property="http://purl.org/dc/terms/title">The Trouble with Bob</h2>
  <p>Date: <span property="http://purl.org/dc/terms/created">2011-09-10</span></p>
</div>
```

with the resolving triples

```
<http://example.org/posts/trouble> <http://purl.org/dc/terms/title>
  "The Trouble with Bob" .
<http://example.org/posts/trouble> <http://purl.org/dc/terms/created>
  "2011-09-10" .
```

The Turtle²¹ format is used to describe an RDF graph in plain text and comes close to natural language. To avoid typing long URIs repeatedly again, Turtle uses the `@prefix` notation to abbreviate URIs. In this case `somePrefix` gets replaced with the URI specified on the `@prefix` line.

```
@prefix somePrefix: <http://www.perceive.net/schemas/relationship/> .

<http://example.org/#green-goblin> somePrefix:enemyOf
  <http://example.org/#spiderman> .
```

N-triples²² are a subset of Turtle where each line must contain a subject, predicate and object without the use of prefixes. This results in larger file sizes, but has the advantage that each line is parse-able itself (Heath and Bizer, 2011). Another format is JSON-LD²³, a JSON-based serialization for Linked Data. JSON-LD can either be embedded into a website, enclosed by `<script type="application/ld+json">` or just published as serialized JSON. The Internet Movie Database (IMDb) uses this to describe movies on their web page, for example the movie Frozen II²⁴ has the following embedded:

```
<script type="application/ld+json">{
  "@context": "http://schema.org",
  "@type": "Movie",
  "url": "/title/tt4520988/",
  "name": "Frozen II",
```

²⁰ <https://www.w3.org/TR/rdfa-primer/>

²¹ <https://www.w3.org/TR/turtle/>

²² <https://www.w3.org/TR/n-triples/>

²³ <https://www.w3.org/TR/json-ld/>

²⁴ <https://www.imdb.com/title/tt4520988> (accessed 2019-06-13)

```

    "genre": [
        "Animation",
        "Adventure"
    ]
}
</script>

```

2.2.3 Applications

Bizer (2009) attribute the starting point of putting Linked Data into the web to the Linked Open Data (LOD) project. The Linked Data community has since then been identifying datasets and converting them to RDF for publication. Early participants were mainly researchers at universities. Later, larger organizations such as BBC and Thomson Reuters followed foot (Heath and Bizer, 2011). BBC in that case started by publishing data about their programs and about the music that was running on their radios. They also interlinked the data with an open-license music database and with DBpedia (Bizer, 2009).

Auer et al. (2007) built DBpedia as a tool to extract data from Wikipedia and put it into a set of linked data called DBpedia. In DBpedia, the resource identifiers return a RDF description of the resource if a machine accesses it. Otherwise a HTML view of the resource is provided. Furthermore, Morsey et al. (2012) extend DBpedia by processing the wikistream output of Wikipedia live. This synchronizes DBpedia with Wikipedia continuously. Ayala et al. (2017) use DBpedia to identify complementary products, e.g. a cartridge and a printer. Their models manage to predict complementary products solely based on the meta-data of the products (titles, descriptions, categories, etc.) and do not rely on previous customer's transactions.

Valsecchi et al. (2015) provide a map-like visualization to give the user an overview and feeling of the datasets. The tool gives an overview of the dataset, but also allows zooming in further to discover more details²⁵.

2.3 Publication

Shadbolt et al. (2012) identify following challenges for integrating open data into the web of linked data. First, one must find appropriate datasets with an open license for applications. Then there need to be good join points for connecting diverse datasets and integrating them into the linked-data web. At last, there should be client software to use the data in order to create a meaningful representation for a user. There's also the Comprehensive Knowledge Archive Network (CKAN) to provide an entry point to the user. CKAN serves as a platform for publishing and discovering open data²⁶. Neumaier et al. (2017) address those issues in their example of making a linked open data portal. Their work exposes and connects metadata descriptions on the open data portals by

²⁵ <http://wafi.iit.cnr.it/lod/dbpedia/atlas/>

²⁶ <https://ckan.org/>

encoding the metadata in standard vocabularies such as DCAT and schema.org. They further build a SPARQL endpoint for querying the data. SPAQRL is an RDF query language.

2.4 Streaming Linked Data

A data stream is a sequence of digitally encoded signals containing information²⁷. [Sequeda and Corcho \(2009\)](#) introduce the concept of Linked Stream Data in the context of the growing number of sensors publishing data on the web. Those sensors publish data in short time intervals and commonly records numerical values such as time, geolocation, humidity and other physical measurements.

Triples streamed by the sensors may not be stored forever, and need to be processed on the fly. In this case, queries need to be continuously executed on the stream once registered. To continuously query those RDF data streams, [Barbieri and Della Valle \(2010\)](#) propose an extension of SPARQL²⁸ called C-SPARQL. SPARQL itself is a query language for RDF. The C-SPARQL query computes the results in a specified time interval and over a window of time and then generates an RDF stream. [Barbieri and Della Valle \(2010\)](#) define an RDF stream as an ordered sequence of (Triple, timestamp)-pairs. Timestamps may not be unique, but they should not decrease. The streamed data triples are part of some Instantaneous Graph (i-graph) determined by their timestamp. The so-called Stream Graph (s-graph), on the other hand, contains the metadata that describes the stream. Such metadata may contain information about the last update of the stream or pointers to the i-graph instances ([Barbieri and Della Valle, 2010](#)).

In order to increase accessibility, [Barbieri and Della Valle \(2010\)](#) further provide a RESTful interface to control the C-SPARQL engine and its queries. This allows clients to register, start, stop and delete queries on RDF streams processed by the engine. Similar extensions of SPARQL are EP-SPARQL by [Anicic et al. \(2011\)](#) and CQELS by [Le-Phuoc et al. \(2011\)](#).

For linked sensor data, [Taelman et al. \(2016\)](#) propose a less expensive alternative to RDF stream processing query engines such as C-SPARQL. Their solution is tailored towards the processing of sensor data. [Barnaghi et al. \(2010\)](#) present the Sense2Web platform to publish linked-sensor-data to the web. Users can publish and link their data to existing resources on the platform. To perform queries on the data, the platform exposes a SPARQL endpoint. One presented use case is a map application that joins location data with sensor data in the proximity of a location. Unlike [Barbieri and Della Valle \(2010\)](#), updates of the data were not in the scope of the project.

[Balduini et al. \(2013\)](#) report on using the streaming linked data framework (SLD) to analyze information from the social web during city-scale events (e.g. Olympic Games). The different venues and involved people make this particularly interesting. SLD uses RDF to model the data and C-SPARQL to process and analyze the data. For example,

²⁷https://www.its.bldrdoc.gov/fs-1037/dir-010/_1451.htm (accessed 2019-03-18)

²⁸<https://www.w3.org/TR/sparql11-query/>

the live stream of locations can then be used to generate a heat map and visualize the flow of the crowd.

So far the presented approaches have not focused on the publication and offloaded the task of managing the stream publication to the developers. Therefore, [Mauri et al. \(2016\)](#) propose TripleWave as a generic tool to create and distribute RDF streams on the web. TripleWave can either transform non-RDF input into an RDF stream or create time-annotated datasets from RDF streams. The output of TripleWave is a stream in the JSON-LD format. For non-RDF datasets, TripleWave makes use of the RDB to RDF mapping language (R2RML²⁹), which allows to map relational databases to RDF datasets. With the TripleWave extension JRML of [Bernhaut \(2018\)](#) it is also possible to define the mapping in a RML-like notation using JavaScript objects. Regarding RDF datasets, TripleWave supports different running modes, such as replay and endless. A static dataset with time-annotations for example, can be replayed in the order of the triple's timestamps. This can be used to benchmark and test applications that deal with that kind of data. Moreover, it is also possible to loop the replay over and over again ([Mauri et al., 2016](#)).

The output of TripleWave can then be consumed by either a WebSocket client or by a MQTT-Client. This enables the client to choose whether to receive the data by a push or to pull the data from a MQTT broker ([Mauri et al., 2016](#)). The output schema follows [Barbieri and Della Valle \(2010\)](#)'s proposition to put the stream data elements into Instantaneous Graphs (i-graphs) and expose the metadata through a separate graph called stream graph (s-graph). The stream graph is accessible by a HTTP request on the stream endpoint and allows flexible aggregation of stream metadata of the TripleWave instances [Mauri et al. \(2016\)](#).

In the interest of creating a common vocabulary for streaming data catalogs, [Tommasini et al. \(2018\)](#) propose the Vocabulary for Cataloging and Linking Streams (VoCaLS). VoCaLS extends vocabularies such as DCAT, Dcterms and VoID that were primarily created with static data in mind. Thus, VoCaLS aims to bridge the gap between static datasets and streamed datasets. VoCaLS is divided into three modules. The VoCaLS Core module enables the description of streams, such as license, access URL and the output format. In addition, the Service Description module is used to describe the offered services and capabilities of streaming service. The third module, Provenance, allows to track the transformation of the stream, such as describing mapping operators ([Tommasini et al., 2018](#)).

²⁹<https://www.w3.org/TR/r2rml/>

3

Requirements

The following section is divided into requirements for the backend systems for streaming data, and for the catalog serving the metadata of the streams. The backend consists of TripleWave instances with the integrated JRML module [Bernhaut \(2018\)](#) that is responsible for fetching and transforming the datasets on a fixed schedule and finally forwarding it to TripleWave.

In Section [3.1](#) we discuss the requirements for publishing the stream and its metadata to the catalog. In Section [3.2](#) we list the requirements for the catalog. In order to accomplish these goals, we identify the following requirements. The words "must", "should" and "may" are used in accordance with RFC-2119 ([Bradner, 1997](#)).

3.1 TripleWave

Initially we identify the requirements to stream datasets found on open data portals and what interfaces TripleWave must offer to interact with the catalog. The goal is to review whether those requirements are already met by TripleWave and JRML. If not, we will extend TripleWave or JRML.

3.1.1 TripleWave Interface

RT1 TripleWave must create a description of the stream.

The stream description must contain all the relevant metadata (e.g. title, description, license, source, endpoints) of the stream. The vocabulary used should reuse existing vocabulary as much as possible.

RT2 TripleWave must push the description to the catalog.

TripleWave must actively push the stream description to a service or to the catalog directly. Before pushing the stream description, TripleWave must check whether the stream is already registered at the catalog. Based on the catalog's response, TripleWave either sends a create or update catalog entry command to the catalog endpoint.

RT3 TripleWave must serve the stream to multiple clients.

TripleWave or another service must serve the stream to multiple clients. TripleWave must use the address specified in the configuration and include the endpoint addresses in the stream description.

Should another service be used to supply clients with the stream, TripleWave must publish the addresses through the stream description as well.

RT4 TripleWave must grant access to clients outside the intranet.

Clients outside the internal network must be able to access a TripleWave stream.

RT5 The backend setup should be manageable

TripleWave already allows us to spawn a new child process for each data set and keeps the configuration in one file. If new services are introduced, they should be similarly managed.

3.1.2 Streaming Datasets with JRML

The requirements for JRML are explained in detail by [Bernhaut \(2018\)](#). RB1 and RB2 refer to the basic requirements of JRML. We identify two new requirements RB3 and RB4 during the creation of web streams using JRML.

RJ1 JRML must support CSV, JSON and XML data sources

Most open data on the web does not come in a format for linked data. Thus, it is necessary for TripleWave to be able to transform the most common data formats into RDF triples.

RJ2 JRML must fetch data periodically

The creator of the stream must be able to specify a time interval for scheduling the retrieval of the data source.

RJ3 JRML should discard data records containing invalid values

Some datasets contain invalid records that should not be published over the stream. For example, the fine dust measurements dataset on the Austrian portal¹ uses -999 for invalid entries. Since it is impossible to automatically identify all kinds of invalid values, we limit the scope of this requirement to discard only a set of null values that is specified by the developer. Those null values should be passed to JRML before the start-up.

RJ4 JRML should be able to update the data source url

Some data source endpoints may change their URL over time, e.g. generating a new path for each month. More commonly, an endpoint may accept query parameters by appending a query string at the end of the URL.

¹<https://www.data.gv.at/katalog/dataset/8b057f32-1312-40ae-ae51-9aa0a0d372ca> (accessed 2019-06-14)

This allows to apply initial filtering of the dataset on the server side of the data source. Consequently, we reduce network traffic and processing power on our side.

A common use case is the filtering by date, e.g. `?publishDate=today` where `today` gets evaluated just before sending the request to retrieve the original data source.

3.2 Catalog

The catalog requirements should help identifying a suitable software framework to build the web stream catalog. There are already frameworks in use for hosting static datasets and metadata. Thus, we are confident to extend such a framework to fit our requirements and do not need to start from scratch.

We do not explicitly require a neat visual styling of the site. But since our initial survey of frameworks for creating catalogs indicated the possibility to create templates of web pages, we are confident to satisfy visual requirements. The portal must be accessible for automatic and human agents. Therefore, the portal has to supply different views of the catalog. Furthermore, a human agent might want to see a preview of the data stream before accessing the endpoints directly. The platform needs to store the metadata but does not need to store the data itself. Instead the platform should point the clients to the stream endpoints.

We begin with the functional requirements and then list the non-functional ones.

3.2.1 Functional Requirements

In this section we describe the functional requirements for the catalog. First, we shortly define the actors that interact with the catalog.

We name *publisher* the entity that creates a catalog entry. I.e. the publisher pushes (meta)data to the catalog over the network using some sort of API. The *developer* is responsible for adapting and extending the catalog's functionality. For example, defining how the data received from the publisher is parsed or designing and modifying the layout of the web pages. The *administrator* manages the publication of datasets through interfaces provided by the catalog and does not modify the source code of the catalog. Lastly the *user* is the human or machine that reads the catalog and possibly wishes to connect to the stream.

RC1 The catalog must provide an API for registering a stream.

There must be an interface where a publisher can register a stream and push the metadata to the catalog.

RC2 The catalog should provide functionality for updating streams.

It should be possible to update the metadata of the stream via an API. We want to use such a functionality to update the corresponding catalog entry each time TripleWave is restarted.

RC3 The catalog must create static URLs.

When a new page is created for a new dataset, the URL must remain unchanged. This allows a publisher to update the dataset later by using the URL or part of it as an identifier.

RC4 A catalog's page URL must uniquely identify a single dataset

When a new page for a stream is created, a unique URL for that page must be generated or inferred from the data pushed to the catalog. If inferred URL already exists on the catalog an error must be returned to the publisher. Returning an error is important because TripleWave must update the dataset if it already exists.

RC5 The developer must be able to create new data fields.

The schema of the metadata on the catalog side must be modifiable. Should new fields be required that are not available in the catalog, the catalog must be able to create new fields with the corresponding name.

RC6 The catalog must hook into creation of a dataset.

It must be possible to parse the input from a dataset creation or update and map the input data to the catalog's internal representation. E.g. `dct:title` could be mapped to the `title` field of the catalog.

RC7 The catalog must be able to parse data in RDF.

TripleWave already publishes its metadata in an RDF serialization format. To reuse this component, we want to do the mapping of the metadata fields on the catalog's side. Hence, the registering stream does not need to know everything about our catalog's internal mapping.

RC8 The catalog must recognize labels for predicates.

The publisher may define a human-readable version for predicate names. When the catalog shows the metadata to a human user, the catalog must prefer the label over the predicate's URI. This does not affect the machine-readable version of the stream descriptor.

RC9 The catalog should provide human-readable names for common predicates by default.

Some resources appear in every dataset. In order to not repeat the label declarations for each dataset, the catalog should support some sort of default labels. Those labels should have a lower precedence than labels defined by the publisher.

RC10 The catalog must be customisable.

In order to show the metadata in a format suited for web streams, it must be possible to change the content of the web page.

RC11 The catalog must provide means to find datasets.

For exploring the catalog, the catalog must functionality to find datasets. A user must be able to search the catalog by keywords. Furthermore, the catalog must provide a list of datasets on the catalog.

RC12 The catalog must give a preview of the stream.

The catalog should enable the user to see a preview of the data stream. On opening the preview, the catalog must retrieve that latest message and deliver updates as soon as possible.

RC13 The catalog should explain how access web streams

Even though the RDF description of the endpoints are descriptive enough to access those web streams, the user may not know what kind of client software to use to consume those streams. The catalog should therefore provide some sort of manual for accessing web streams.

RC14 The catalog may pull metadata periodically.

The catalog may pull the metadata from TripleWave in a specified interval.

RC15 The developer may create a new page.

The catalog may allow administrators to create new pages rather than only using default pages.

3.2.2 Non-functional Requirements

RN1 The catalog must be open source and free software.

Anyone should be able to freely modify and extend the software.

RN2 The catalog must have documentation online.

There must be a documentation for using, extending and deploying the catalog available.

Architecture

In this chapter we present the overall architecture of the catalog and the systems in the backend for delivering the web streams. Section 4.1 gives an overview of the complete architecture with the catalog and the publishing of the streams. In Section 4.2 we describe the role of TripleWave and JRML in publishing streams and show the consumption of streams in Section 4.3. In Section 4.4 we proceed with explaining the setup of the Kafka and MQTT broker. Finally, in section 4.5 we give a brief overview of CKAN.

4.1 Overview

The core of this project is to create a catalog of web streams. In order to create and publish the catalog we use the Comprehensive Knowledge Archive Network (CKAN), serving the catalog web pages and handling the retrieved metadata from TripleWave. The catalog serves as an entry point for users and machines to discover web streams and access information about those streams. An overview of the architecture is depicted in Figure 4.1.

We use TripleWave to pull, transform and publish linked data as a web stream. The resulting output stream is then forwarded to two different message brokers over the network (a Kafka and an MQTT broker). We use message brokers as intermediaries because they can serve as a single access point for clients. The endpoints of the message brokers are included in the stream description and are listed on the catalog. Muntwyler (2017) used Kafka for the scalability and modularity, allowing the publisher to increase the number of open connections by dynamically adding more brokers to the cluster. Further, using a widely-used broker further reduces the effort required to manage stream consumption, because there are existing libraries handling it. Additionally, we add an MQTT broker to leverage TripleWave's support of the MQTT protocol and to increase the number of access methods for clients. In our implementation TripleWave automatically pushes its stream description to the catalog via the CKAN API¹. The API exposes CKAN's core features and most allows us to create and update data entries of the catalog.

¹<https://docs.ckan.org/en/2.8/api/>

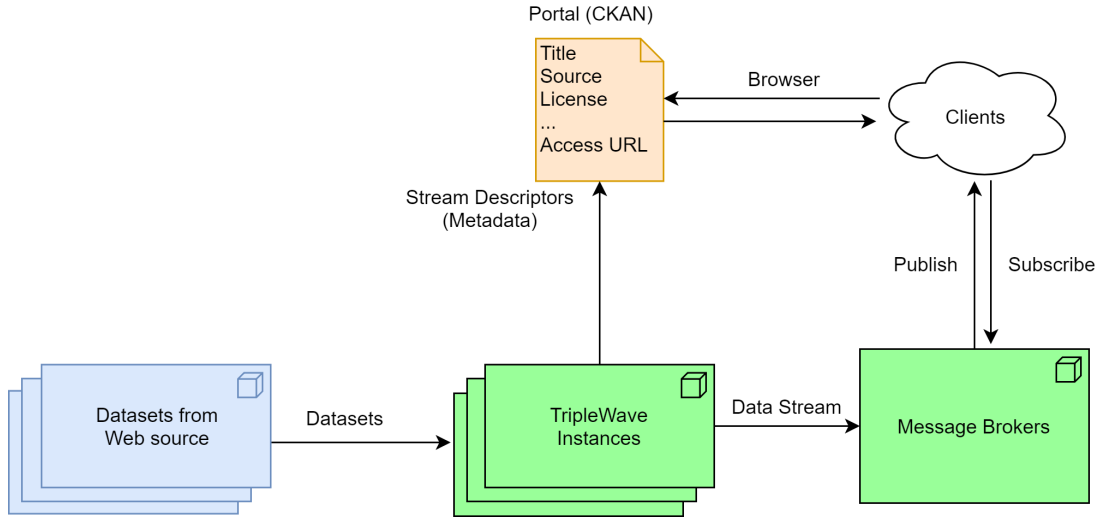


Figure 4.1: Overview of all components

4.2 Publishing data streams

TripleWave is a tool to publish RDF streams to the web. TripleWave may be fed with existing RDF streams or can be used together with JRML to transform non-RDF datasets to an RDF stream. We use TripleWave together with JRML to pull the data from the original source and transform the data to a stream. The creation and publication of web streams is shown more closely in figure 4.2. JRML is a result of Bernhaut (2018)’s thesis and allows to define the transformation of the data in a purely declarative way. It is a Node.js module that maps data to linked data and has an integrated harvesting functionality that fetches and transforms the data on a schedule and feeds it to TripleWave. We decided to use JRML together with TripleWave because it is convenient to use and saves time and effort in mapping the data on the web to linked data. JRML can transform CSV, XML and JSON formatted data into an RDF stream that is then processed and published by TripleWave.

4.3 Consuming TripleWave Streams

Unlike static datasets, the data cannot be downloaded at once. The consumer has to continuously listen the publisher of the data stream. TripleWave’s final output is an RDF stream over HTTP using chunked transfer encoding, over WebSockets and MQTT (Mauri et al., 2016). Chunked transfer encoding divides the data stream a series of disjoint chunks. Those chunks arrive independently from each other at the target location. This is required to keep the connection to the client open and allows to transfer data of unknown size². Alternatively, TripleWave uses the WebSocket protocol

²<https://tools.ietf.org/html/rfc7230#section-4.1>, (accessed 2019-04-19)

and acts as a WebSocket server³. The WebSocket protocol provides a single connection for traffic in both directions. Most importantly, it allows the server to push messages to the client without the client putting a request in first. The MQTT output is published to the MQTT broker listening to the endpoint declared in the TripleWave settings. Clients can then subscribe to web streams through the MQTT broker.

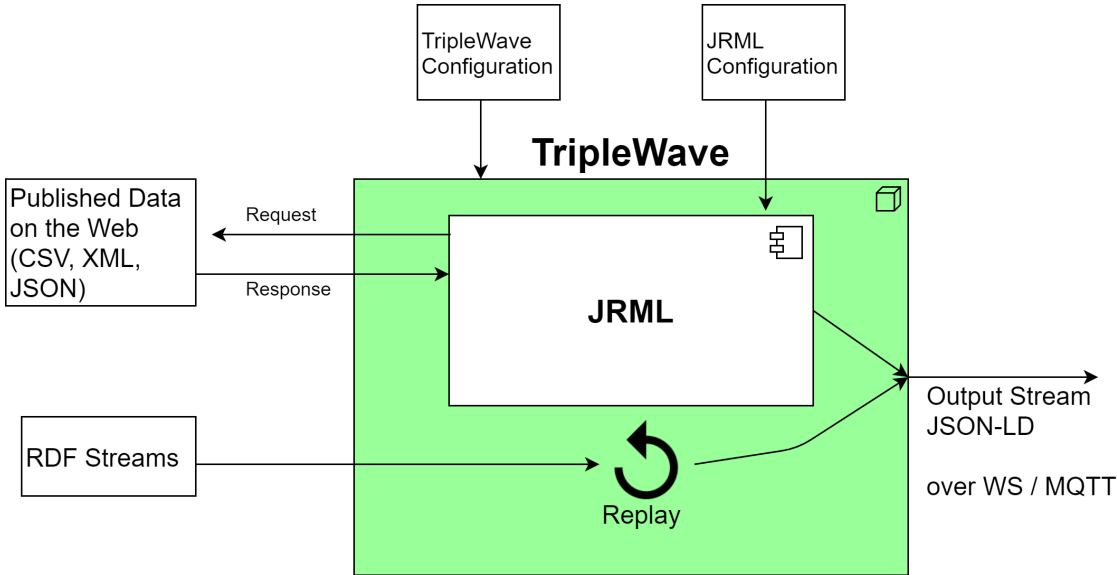


Figure 4.2: Overview of TripleWave (source: Bernhaut (2018))

4.4 Message Brokers

We deploy Apache Kafka⁴, a distributed streaming platform, to achieve higher throughput and to serve multiple streams on the same endpoint (Muntwyler, 2017). This meets requirement RT3 (serving the stream to multiple clients) from Section 3.1. Kafka uses the publish-subscribe mechanism to distribute streams of messages across topics. A topic is a feed name to which messages are published. A message in this case represents a single RDF graph of TripleWave in JSON-LD from a single data retrieval. A client can then choose to subscribe to messages of a specific topic. We assign a unique topic to each web stream in order to let the client decide which web streams to consume.

The red part of Figure 4.3 shows the flow of the stream across Kafka. In Section 4.4.1 we describe the component that forwards TripleWave’s output to the Kafka cluster; then we show the consumer side that listens to the Kafka topics and forwards it to our consumer components in Section 4.4.2. Finally, we introduce the MQTT broker in Section 4.4.3.

³<https://tools.ietf.org/html/rfc6455>, (accessed 2019-04-19)

⁴<https://kafka.apache.org/> (accessed 2019-04-18)

4.4.1 Kafka Producer

The Kafka producer is responsible for forwarding TripleWave’s RDF stream output to the Kafka cluster. It is a Node.js process that listens to the WebSocket servers of each TripleWave instance and publishes the incoming web streams under the corresponding topic on the Kafka cluster.

The WebSocket connections and data transformations are handled by Primus⁵. Primus is a wrapper for managing communication through different WebSocket frameworks. For our needs we only require a simple WebSocket server. We primarily use Primus because of its ability to reconnect to the server and its built-in parser functionality. The parser is needed to stringify the JSON output from TripleWave.

4.4.2 Kafka Output

The WebSocket Consumer component (Figure 4.3) subscribes to requested Kafka topics on behalf of clients and forwards the message streams to the client using the same WebSocket connection from the client request. We use Primus to create a WebSocket server that handles the incoming requests and subscribes to the corresponding topics. The client may request a specific topic by encoding the name of the topic in the request.

As a second access method is the EventSource interface⁶ to handle server-sent events (SSE). We provide this access method because it’s particularly easy to implement on the client side and because it’s designed for continuous data streams.

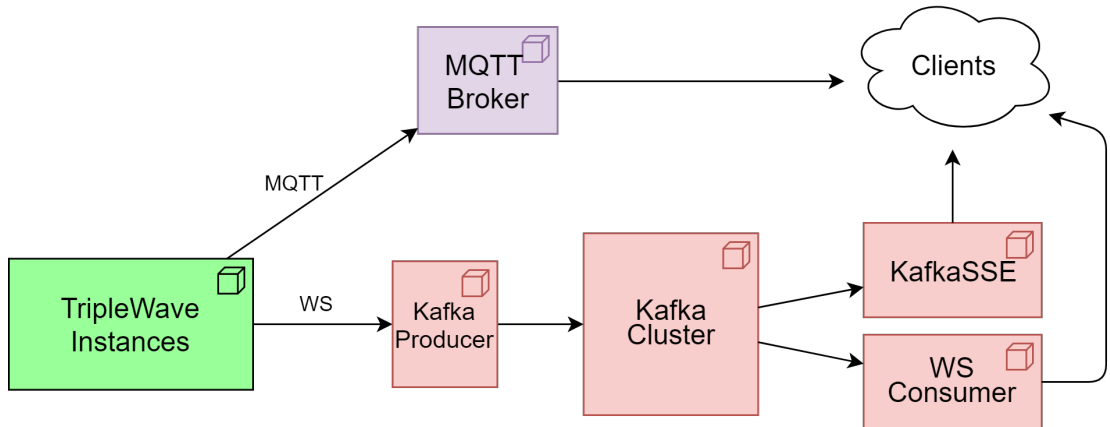


Figure 4.3: Overview of the message brokers

⁵<https://github.com/primus/primus> (accessed 2019-04-19)

⁶<https://developer.mozilla.org/docs/Web/API/EventSource> (accessed 2019-04-19)

4.4.3 MQTT Broker

MQ Telemetry Transport (MQTT)⁷ is a lightweight messaging protocol that allows publishing and subscribing to topics. MQTT supports basic quality of service (QoS) settings for delivering messages to the clients. TripleWave in this case publishes messages at the QoS level 0. This means messages are delivered at best-effort and may arrive once or not at all at their target destination. Level 0 may be used for temperature readings where the latest message matters the most. If all messages must arrive, the publisher must use QoS level 1 that guarantees at-least-once delivery (Hunkeler et al., 2008). Analogous to the Kafka topic, we use the identifying name of the dataset as the topic name.

4.5 CKAN

We use CKAN⁸ to create our platform and publish our metadata. We believe CKAN meets the requirements of our catalog⁹. CKAN is published under the GNU Affero General Public License (AGPL¹⁰). CKAN is built with Python¹¹ and uses the web frameworks Pylons and Flask¹² to serve the web pages. For its database CKAN uses PostgreSQL as a database. CKAN interacts with PostgreSQL through SQLAlchemy (an object-relational mapper).

As shown in Figure 4.1 TripleWave pushes the stream descriptor (the streams metadata in RDF) to CKAN. Even though CKAN has a web user interface to publish data, we do not use it at all. Instead we interact with CKAN exclusively through its API¹³. The CKAN API provides access to all CKAN's core features. The API allows us to read, create and update datasets programmatically. As a result, we do not have to register web streams ourselves and are able to use data that's already available from the TripleWave configuration.

CKAN also has the capability to store data itself¹⁴. Data storage is not a requirement of our catalog, therefore we omit setting up any of the components related to it.

⁷<https://mqtt.org/> (accessed 2019-04-18)

⁸<https://ckan.org/> (accessed 2019-04-21)

⁹<https://ckan.org/> (accessed 2019-04-18)

¹⁰<https://www.gnu.org/licenses/agpl-3.0.html> (accessed 2019-04-18)

¹¹<https://ckan.org/developers/about-ckan/> (accessed 2019-04-21)

¹²<https://github.com/ckan/ckan/wiki/Migration-from-Pylons-to-Flask> (accessed 2019-04-21)

¹³<https://docs.ckan.org/en/2.8/api/index.html> (accessed 2019-04-21)

¹⁴<https://docs.ckan.org/en/2.8/maintaining/filestore.html> (accessed 2019-04-21)

Changes to TripleWave

In this section we introduce the modifications and extensions of TripleWave and the JRML module to meet the requirements in Section 3.1 and Section 3.1.2. The added features enable TripleWave to interact with the catalog. Section 5.1 introduces the changes to the vocabulary that TripleWave uses to describe its stream. In Section 5.2 we then show how TripleWave registers itself at the catalog; in Section 5.3 we adapt the TripleWave configuration according to the added features. In Section 5.4 we list minor changes applied to JRML.

5.1 Vocabulary

Streams on the web require ways to describe their metadata and access points on the web. Unlike static datasets, web streams are more dynamic and offer different kind of mechanism for consuming and processing streams on the web. Therefore, using catalog vocabulary designed for static datasets (e.g. dcat) is not enough (Tommasini et al., 2018). Instead, we need to use vocabulary geared towards streaming linked data. TripleWave previously used its own ad-hoc vocabulary *sld*¹ and switched parts to *rsd* later. To use standard vocabulary and to increase interoperability with other potential services, we adopt Tommasini et al. (2018)’s Vocabulary & Catalog of Linked Streams (VoCaLS).

The use of VoCaLS concerns requirement RB1 (see Section 3.1). Further, we reuse other existing vocabulary as much as possible when encoding the metadata of the web stream. Mauri et al. (2016) refer to TripleWave’s stream description as *sGraph* for stream graph, but we use ”stream description” or ”stream descriptor” to be closer to VoCaLS.

First, we replace *sld* and *rsd* terms with terms from VoCaLS wherever possible. Next, we add new vocabulary that is present in VoCaLS and not in TripleWave to further enrich the metadata. In a third step, we encapsulate the previous stream description (the *sGraph*) in the new stream descriptor as a *dcat:Dataset*.

In dcat terms, a Dataset describes ”A collection of data, published or curated by a single agent, and available for access or download in one or more formats.”². As a result, we can describe the stream descriptor itself and introduce the notion of a catalog

¹<http://streamreasoning.org/ontologies/SLD4TripleWave> (accessed 2019-04-21)

²<https://www.w3.org/TR/vocab-dcat-2/#Class:Dataset> (accessed 2019-04-23)

of stream descriptions (Tommasini et al., 2018). On top of that it allows us to distinguish metadata about the original dataset, and data that has been added by the catalog. For example *dcat:modified* can be used to indicate when the catalog has been updated last³.

VoCaLS is built on top of other vocabularies that are primarily concerned with static and stored data in mind. They do not provide all the terms necessary to adequately describe web streams. VoCaLS seeks to close that gap. We further use the following vocabularies:

- **Data Catalog Vocabulary (DCAT, dcat)** is an RDF vocabulary describing data catalogs on the web (Maali et al., 2010). VoCaLS extends the terms *Distribution*, *Dataset* and *Catalog*.
- **Dublin Core Terms (DCterms, dct)**⁴ is a set of generic metadata terms such as *title*, *description* and *publisher*

From VoCaLS we add the following terms:

- *vocals:StreamDescriptor* that indicates a stream descriptor, i.e. the metadata about a web stream. It is a sub-class of *dcat:Catalog*. The *StreamDescriptor* is exposed by TripleWave and is pushed to our catalog
- *vocals:RDFStream* represents an RDF web stream. It is a sub-class of *vocals:Stream*, which represents any kind of stream. The related *dcat* super-class of a stream is *dcat:Dataset*.
- *vocals:StreamEndpoint* the endpoint that a client can use to consume the data generated by the stream. It is a sub-class of *dcat:Distribution*.
- *vocals:hasEndpoint* is a predicate for adding an endpoint to a web stream
- *vsd:CatalogService* to show that the service is providing catalog metadata.

vocals belongs to the core-vocabulary of VoCaLS and *vsd* provides terms to characterize services (Tommasini et al., 2018). Figure 5.1 shows an overview of our stream descriptor incorporating the VoCaLS vocabulary. In Listing 5.1 we show the example we used as a basis for creating TripleWave’s stream descriptor. The empty prefix resolves to the declared base URI. For example `:TempStream` with base <http://example.org/sensors#> resolves to <http://example.org/sensors#TempStream>.

³<https://www.w3.org/TR/vocab-dcat/#class-dataset> (accessed 2019-04-21)

⁴<http://www.dublincore.org/specifications/dublin-core/dcmi-terms/> (accessed 2019-04-21)

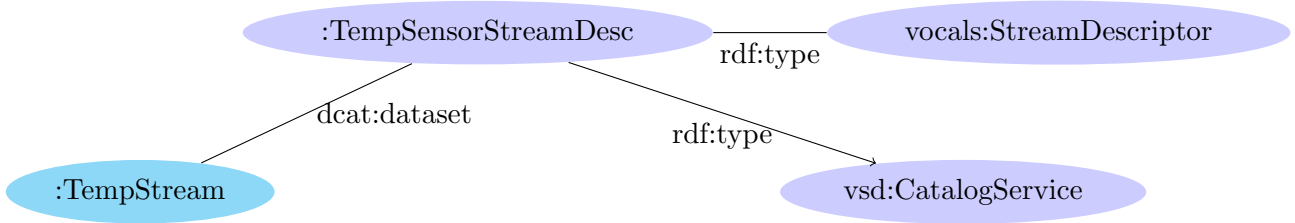


Figure 5.1: The stream descriptor for a stream of temperature sensor data

Figure 5.2 shows the RDFStream endpoint. We add *dct:source* to the RDFStream node to specify the original data source.

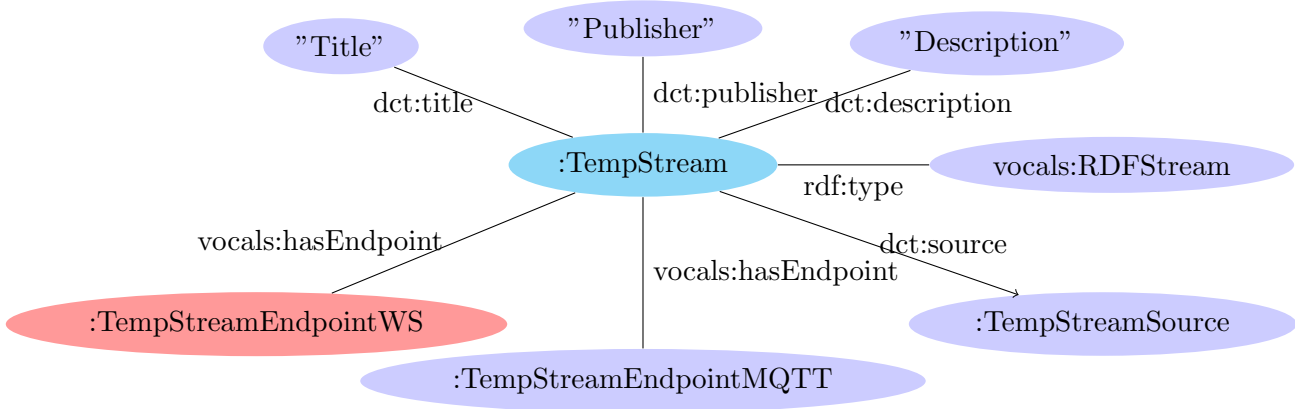


Figure 5.2: An RDFStream containing two different endpoints

In Figure 5.3 we show a stream representing a single endpoint. As shown in the figure, we define the license for each endpoint. This is in agreement with *dcat:Distribution*, even if the catalog's license applies to all distributions in the catalog⁵.

Listing 5.1 shows Tommasini et al. (2018)'s stream description applied to one of our own webstreams in the turtle format. Because Tommasini et al. (2018) do not specify a term for the protocol used, we use *vocalsd:protocol*. There's also no term to define the topic of an MQTT endpoint. Hence we introduce *vocalsd:topic*. For our WebSocket endpoint we do not have to specify a topic because each TripleWave stream has its own address with the topic encoded in the URL.

```
@prefix : <http://example.org/PLS#> .
@base <http://example.org/tw/parking#> .

:ParkingStreamDesc a vocals:StreamDescriptor , vsc:CatalogService ;
```

⁵https://www.w3.org/TR/vocab-dcat/#Property:catalog_license (accessed 2019-04-21)

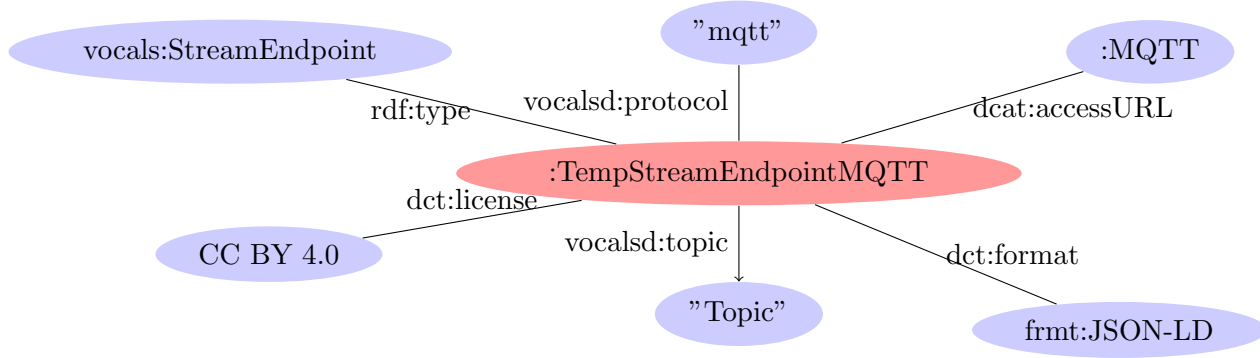


Figure 5.3: A StreamEndpoint

```

dcat:dataset :ParkingStreamZurich .
:ParkingStreamZurich a vocals:RDFStream ;
  vocals:hasEndpoint :ParkingStreamEndpointZurich ;
  dct:title "Parking Zurich" ;
  dct:publisher "PLS Parkleitsystem AG";
  dct:description "Information about parking spaces in Zurich" .
:ParkingStreamEndpointZurich a vocals:StreamEndpoint ;
  dct:license <https://creativecommons.org/publicdomain/zero/1.0/> ;
  dct:format frmt:JSON-LD ;
  vocalsd:protocol "ws" ;
  dcat:accessURL "ws://example.org/plszh/" .

```

Listing 5.1: RDF stream and endpoint description; some prefixes omitted (source: [Tommasini et al. \(2018\)](#)).

5.2 Pushing the metadata to the catalog

To address requirement RT2, TripleWave pushes the stream descriptor to the catalog by creating a dataset entry on CKAN. CKAN stores the stream descriptor as a whole and maps some of the metadata to its own internal representation should there be a matching field (e.g. the title of a dataset). A dataset in CKAN's terms contains metadata and resources which hold the data itself⁶. TripleWave only pushes metadata to CKAN and does not create any resources on the catalog. Because TripleWave's "datasets" on CKAN only contain metadata, we use "dataset" and "catalog entry" interchangeably.

More specifically, TripleWave utilizes the CKAN API methods *package_show*, *package_create* and *package_update*⁷. Package refers to dataset in this case.

TripleWave first checks if there is already an existing dataset entry for its stream by using *package_show*. This method uses the name of the stream as a unique identifier

⁶<https://docs.ckan.org/en/2.8/user-guide.html#datasets-and-resources> (accessed 2019-04-21)

⁷<https://docs.ckan.org/en/ckan-2.7.3/api/index.html> (accessed 2019-04-21)

and returns the stored metadata as a dictionary. If the entry already exists, TripleWave sends a request to the URL of *package_update* with the updated dictionary. Otherwise *package_create* is called. The corresponding sequence diagram is shown in Figure 5.4. The metadata dictionary is named *data_dict* in the figure.

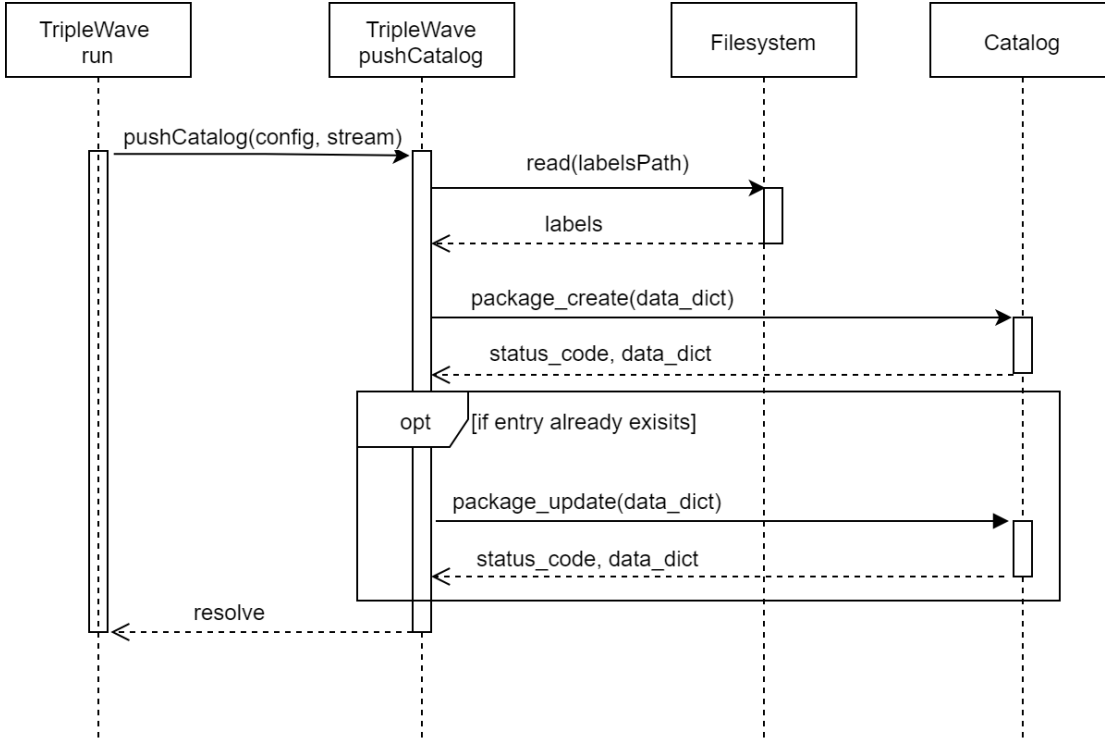


Figure 5.4: TripleWave registering the stream

To create a dataset using CKAN’s API, we require six parameters. The first parameter is the *name* of the dataset. This parameter has to be unique on CKAN’s side and serves as a primary identifier for a dataset on the catalog. The 2nd parameter *streamdescriptor* is the stream descriptor that’s already been generated by TripleWave. The 3rd parameter *labels* holds the labels for the predicates of the stream descriptor and the 4th parameter *use_default_labels* enables TripleWave to use the default labels of the catalog itself. A label definition has the form **R** **rdfs:label** **L** and means that **L** is a human readable version of **R**⁸. We implement the label definitions on the publisher side, because different streams might chose different labels in some cases. However, we aim to keep the labels consistent across all of our published web streams.

The 5th parameter sets the original source of the dataset because that information is not present in the stream description. Finally, the 6th parameter *owner_org* is the name of the organization on CKAN that creates and owns the datasets on CKAN. All parameters are either read from the TripleWave configuration file or generated by

⁸https://www.w3.org/TR/rdf-schema/#ch_label (accessed 2019-04-21)

TripleWave (e.g. the stream descriptor).

We specifically chose not to specify other dataset properties for CKAN (such as license) in the package dictionary that is pushed to CKAN because we handle this on CKAN's side when parsing the stream descriptor. This way, TripleWave does not need to know about CKAN's internal representation of the data. For example CKAN uses three properties to describe the license. *license_id*, *license_title*, *license_url* and finally *isopen* to indicate whether a license is open or not in reference to the open definition⁹. TripleWave on the other hand stores the license as a resource identified by a URL (i.e. a *dct:LicenseDocument*¹⁰).

5.3 TripleWave Config

TripleWave's configuration is stored in a JavaScript file and gets imported by TripleWave as a JavaScript object that is passed to all functions that require it (Bernhaut, 2018). We change two parts of TripleWave's configuration. First we change the way TripleWave stores the endpoints of the stream. The second change provides the option to add more metadata to the stream description.

5.3.1 Advertise endpoints

Originally, TripleWave used the WebSocket and MQTT settings to create the endpoints (i.e. start a WebSocket server with the given settings) and also to advertise them in the stream description for clients on the Internet. This avoids the duplication of settings if both internal and external endpoints are the same. However, we recognize that some endpoints should not be accessible from the outside. For example, we use TripleWave's WebSocket output only internally and provide web access on another endpoint that connects to the Kafka broker. Therefore we add the property *external* that is containing information about all externally accessible endpoints, which are added to the stream description of TripleWave. As shown in Listing 5.2 and 5.3, we adhere to the original schema to describe those endpoints and omit the unnecessary parts. We chose to preserve the structure of the internal server settings although wrapping them into an *internal* may be better for showing intent. This way, we preserve compatibility with older versions of TripleWave.

```
ws: {
  enabled: true,
  stream_location: '/stream'
  port: 8100,
```

⁹<https://opendefinition.org/od/2.1/en/> (accessed 2019-05-11)

¹⁰<http://dublincore.org/specifications/dublin-core/dcmi-terms/2012-06-14/?v=terms#LicenseDocument> (accessed 2019-05-11)

```
        address: 'ws://localhost:8100/tw/stream'
    }
}
```

Listing 5.2: Internal endpoint settings

```
external: {
  ws: {
    port: 8200,
    address: 'ws://example.org:8200/twave'
  }
}
```

Listing 5.3: External endpoint settings

For the catalog we use the external settings to advertise our three endpoints that are not created by TripleWave. (a) MQTT over WebSocket for the preview stream in the browser, (b) the WebSocket consumer that lets clients subscribe to Kafka topics, and (c) the EventSource interface.

5.3.2 Adding Catalog Information

Data about a catalog API endpoint where the request to create an entry is sent to, is new to TripleWave. Thus we put all properties that are related to the catalog into a new top-level property *catalog*. Further, we add fields to add more metadata to the stream description of TripleWave. The catalog property is purely optional and if its not present, TripleWave will not attempt to connect to any catalog. This maintains compatibility with older TripleWave versions. Table 5.1 shows all available settings.

5.4 Changes to JRML

In this section we introduce changes to JRML in order to meet the newly added requirements in Section 3.1.2. We also increased the version of the JRML dependency *jsonld* because the old version caused installation errors on our end. In a final step we update the *readme* file to reflect the changes.

5.4.1 Discard invalid values

This change addresses requirement RJ3, making JRML discard invalid values. Some datasets contain invalid values for data that is not available yet or could not be measured.

Properties	Description
catalog.apiEndpoint	The endpoint of the catalog API for creating and updating entries
catalog.apiKey	The API key for the catalog
catalog.owner_org	Required CKAN field for the owner organization of a dataset
catalog.use_default_labels	Boolean for using the catalog's default labels
catalog.labels	The path to the label file, turtle formatted. This value is eventually replaced with the file's content before being pushed to the catalog.
catalog.name	The name of the stream. The catalog uses this field to generate the URL of the catalog entry. Hence the name must be unique
catalog.title	The title of the stream
catalog.publisher	The publisher of the stream
catalog.description	The description of the stream
catalog.license	The license of the stream
catalog.source	The original source of the dataset being streamed
catalog.format	The format of the stream (e.g. JSON-LD)

Table 5.1: Catalog settings

We decide to exclude such records in our data stream. Consequently we need to reliably filter invalid data. Our main goal is to filter values that have been explicitly set to some kind of designated value by the publisher.

Data publishers use different notations, e.g. "null", "N/A", "", or even designated numbers such as 999. Especially the last example shows that it is difficult to catch and omit all values marked as invalid reliably.

[Bernhaut \(2018\)](#) solves this problem by writing a transformer function that receives a string and then returns the transformed string. The transformer function receives the original data source string and modifies it before passing it to the corresponding parser. Since the transformer function can be defined in the JRML configuration for each data source, this is an efficient way to filter some of the low hanging fruits. This approach works particularly well with CSV formatted data because the individual records are

separated by a newline. Therefore, our transformer can work on the raw string and remove each line that contains an invalid value. For JSON and XML we need to first parse the string and construct an object, filter the parts containing invalid values and then convert the object back to a string again.

We attempted to use the transformer option in our initial attempt. However, we had to write a lot of code for each individual dataset that could possibly contain invalid values.

Hence, we propose to add the option to specify invalid values for each dataset in the JRML configuration as a list of strings. We then delegate the filtering to another component of JRML.

The filtering is implemented in the part where JRML produces the N-Quads¹¹ that get passed to TripleWave. N-Quads serializes RDF in plain text and each triple can be parsed independently because there are no prefixes and base URLs used for compaction.

In the JRML codebase the modification is located at *readPredicateObjectMap* of the *transformer.js* component. This function reads all triple objects and then discards any invalid value in the object position. We do not change the functions reading the subjects and predicates, because we do not expect them to contain invalid values and have valid object values at the same time.

5.4.2 Update data source URL

We change JRML to make it possible to update the data source URL before each request. Some APIs accept queries encoded in the URL in the form "key=value". The query component of an URL begins at the first question-mark ("?") and ends at the number sign ("#") or at the end of the URL (see also RFC3986¹²).

The initial version of JRML only accepts a static string for the location of the data source. Consequently it is not possible to pass dynamic values, such as today's date for example, as a query value to the API.

In order to update the data source URL we extend JRML by adding the possibility to define dynamic values for any query parameter in the URL. We achieve this by letting users define key-value pairs in the configuration. The value may be a function that returns a part of the URL that should be updated between requests. JRML will then evaluate the values before each request and transform the key-value pairs into a valid query string. The generated query string has the form "?key1=value1&key2=value2" and gets appended to the static part of the URL. The generated string starts with a question-mark ("?") if the static URL does not already contain one. Otherwise the string starts with a "&". This change addresses requirement RJ4 from Section 3.1.2.

¹¹<https://www.w3.org/TR/n-quads/> (accessed 2019-04-21)

¹²<https://tools.ietf.org/html/rfc3986#section-3.4> (accessed 2019-05-11)

Creation of the Catalog

In this chapter we present the catalog of web streams. The goal of the catalog is to expose web streams created by a streaming service like TripleWave on the web. It serves as a portal to discover web streams and explains users how to connect to those web streams. The catalog collects the stream metadata it receives from the TripleWave instances and creates a catalog entry for each web stream. The catalog itself is not responsible for streaming and hosting data other than metadata itself. The catalog also does not need any knowledge of the streaming service, because the catalog accepts any data pushed to its API. We separate the concerns of streaming and hosting metadata in order to maintain interoperability of both components in other system configurations.

To create the catalog, we use CKAN as a basis. CKAN is a data management system that is generally used for publishing (meta)data for static datasets. It is widely used for creating Open Government Data portals such as opendata.swiss, data.gov.uk and data.gov due to its extensibility (Shadbolt et al., 2012). As discussed in the requirements chapter in Section 3.2, CKAN already fulfills some of the requirements for creating and updating a catalog entry for metadata. We describe in Section 6.1 which requirements are met, and which require modification and extension of CKAN. Since CKAN publishes data in units called "datasets"¹, we use this word interchangeably with the term "catalog entry". A dataset contains a description of a single web stream that is published by TripleWave or any other streaming service that may use CKAN as a catalog.

The rest of the chapter is structured as follows: in Section 6.2 we describe the CKAN architecture and explain how we chose to extend the code base of CKAN to create a catalog for web streams. We then follow-up with describing the process of creating an entry for a web stream on the catalog in Section 6.3. Creating a preview of the stream is explained in Section 6.3.5. Afterwards we show minor changes to CKAN, such as changing the visual styling of the catalog, in Section 6.4. Section 6.5 concludes this chapter by giving a quick overview of the deployment of the catalog.

6.1 Requirements Analysis

We use the functional requirements from Chapter 3 and compare them to CKAN's base functionality. CKAN meets the following requirements without further modification:

¹<https://docs.ckan.org/en/2.8/user-guide.html#datasets-and-resources> (accessed 2019-05-13)

API for registering a stream (RC1) and for updating a stream (RC2). For those we can use the same API as for creating and updating a static dataset on CKAN. When creating a dataset, CKAN creates a static URL for the dataset. This satisfies RC3. Since CKAN uses the name of the dataset in the URL (supplied by the creation request through the API) and verifies its uniqueness, RC4 is met as well.

The remaining requirements need an extension of CKAN. The rest of the chapter focuses on the implementation of new features that meet those requirements. Creating new metadata fields (RC5) is described in Section 6.3.1. The implementations for hooking into the creation of a dataset (RC6) and parsing the RDF metadata input from TripleWave (RC7) are listed in Section 6.3.2. The procedures for using human readable labels instead of full URIs (RC8, RC9) are drawn out in Section 6.3.3. In Section 6.3.5 we discuss RC12 for providing a preview of the stream. Finally, the requirements RC10 and RC11 are related to rendering the web page and their developments discussed in Section 6.4.

6.2 Extending CKAN

Based on the requirements analysis we develop a design package for CKAN, because the base package provides only little customization features out of the box. Examples are setting the title of the page, adding a site logo and appending custom CSS².

A CKAN extension is a Python package that allows us to override CKAN's default behaviour and add new features³. An overview of CKAN's architecture is shown in Figure 6.1; a package can extend all layers of CKAN. However, we follow CKAN's guidelines for writing extension code and do not touch the model's part at all. For creating a catalog entry on CKAN the package parses the data we get from TripleWave, map it to the correct fields and let CKAN deal with lower layers of data storage and retrieval. When the data is then retrieved from storage, we have access to the data and can manipulate it before it is passed to the HTML template or format it in the template itself. The gray part in Figure 6.1 shows the layers our plugin modifies. We do not modify CKAN's own logic functions: we define functions that are executed on top or instead of them.

Since CKAN already has an API for creating catalog entries, we must hook into the creation of such an entry. The data for the catalog is passed as a Python dictionary. CKAN provides ways to intercept that dictionary before or after certain events, by hooking into the logic calls and modifying the data passed to those logic functions. To do this, we implement the so called "plugin interfaces"⁴. Those implementations are then managed by CKAN and get called at events they are bound to. For example, the **after.create** method of the package controller interface is called each time anyone creates a new package (catalog entry in our case). We use this to map TripleWave's

²<https://docs.ckan.org/en/2.8/sysadmin-guide.html#customizing-look-and-feel> (accessed 2019-05-12)

³<https://docs.ckan.org/en/2.8/extensions/tutorial.html> (accessed 2019-05-12)

⁴<https://docs.ckan.org/en/2.8/extensions/plugin-interfaces.html> (accessed 2019-05-12)

metadata representation to CKAN's own representation. The `package_show` is called each time someone requests a view of the catalog entry through the web page or through the API.

In order to render the view of the web pages, CKAN uses Jinja2⁵ HTML templates. Jinja2 is a template engine for Python and allows us to combine dynamic data with static HTML. By creating a template file with the same name in the plugin, we can override or extend the default templates of CKAN. The templates are structured in a hierarchy and a child template inherits from the parent. This allows us to hide unused CKAN elements and show some web stream specific elements instead on specific parts of the page. Templates also have access to helper functions that can be defined by the plugin. Those functions have access to the metadata stored on CKAN and can be used to modify the data before the web page is rendered to the user. We use this to parse and render some of the data that has not been mapped to the CKAN data model. For example, for displaying the stream description in Section 6.3.3. Figure 6.2 shows the implemented interfaces and helper components of the plugin. The interaction with the CKAN components is omitted in the Figure.

6.3 Creating a catalog entry

In this section we explain how a catalog entry is created (in CKAN terms this is the creation of a dataset). We essentially describe what happens after TripleWave pushes the data in the form of a JSON object. In Section 6.3.1 we list the new fields we created for the catalog and in Section 6.3.2 we handle the mapping to already existing fields on CKAN. Section 6.3.3 describes the parsing and view of the stream descriptor that TripleWave pushes to the catalog.

6.3.1 Creating Additional Fields

In CKAN the metadata is stored and displayed in the form of field-value pairs. CKAN comes with a list of standard fields that can be used to show some of the metadata of the catalog entry, such as the author, publication date and license information. The fields of TripleWave's stream description that have an equivalent on CKAN are directly mapped to CKAN's data model. For example, the title and note fields of a CKAN dataset have the stream description equivalents `dct:title` and `dct:description`. For the other metadata fields, we create new CKAN fields by extending the `DatasetForm` interface that allows us to update and modify the available fields of a dataset. The additional fields are important because we can access them within other CKAN modules. For example, the stream descriptor field is accessed by the module that creates the HTML code for the web page. Fields are also indexed and searched by the search functionality of CKAN. Because our portal is only supposed to show datasets of type "web stream", we override the default dataset schema.

⁵<http://jinja.pocoo.org/> (accessed 2019-06-06)

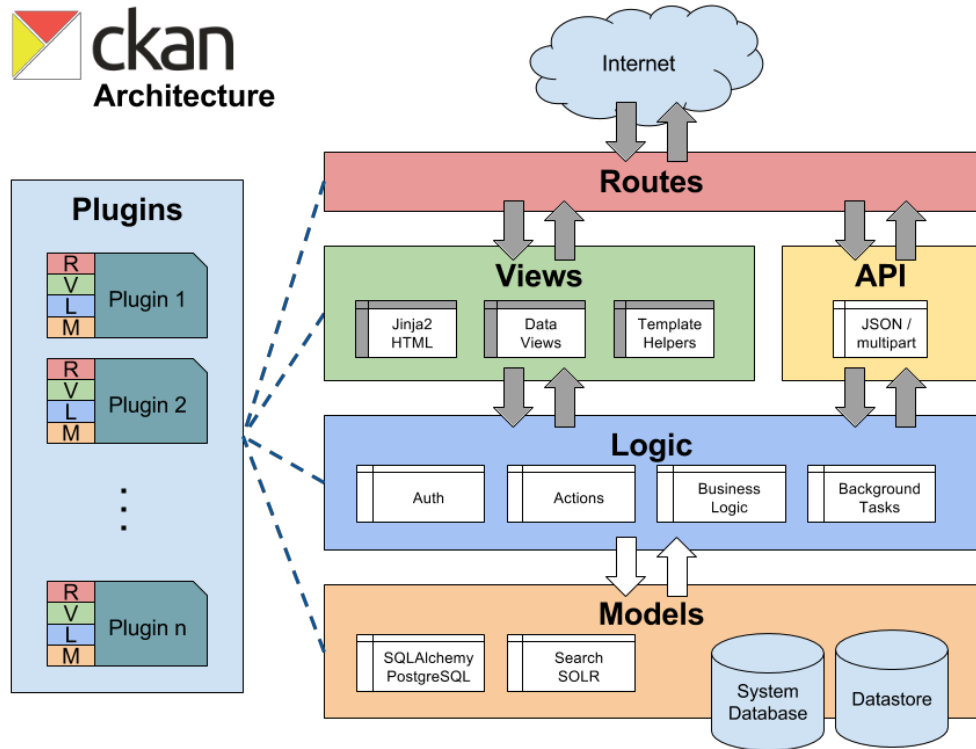


Figure 6.1: CKAN architecture

(source: <https://docs.ckan.org/en/2.8/contributing/architecture.html>)

We do not to map each field in the stream description to CKAN directly, because CKAN stores the fields as field-value pairs represented by a string each. This does not give any room for storing lists or nested data structures as required by our stream description. The stream metadata may contain any number of triples. For example, listing the stream endpoints requires a list of objects, where each object contains a set of triples describing a single stream endpoint. Such a nested structure requires a lot of effort to map to a field-value schema. Therefore, we decide to store the stream description in a CKAN field as a serialized JSON-LD string that is pushed to the catalog. The stream description is then parsed on the fly each time a catalog entry is requested (see also Section 6.3.3). This applies to catalog reads through the browser as well as reads through the CKAN web API.

On the web page we use labels to provide a human-readable version of a predicate URI. The labels provide a human-readable version for predicate names in the RDF triples. For example, <http://w3id.org/rsp/vocals#hasEndpoint> is displayed as "has Endpoint". We add default labels to the catalog for the predicates used to describe our web streams, so we do not have to define them separately in each stream configuration.

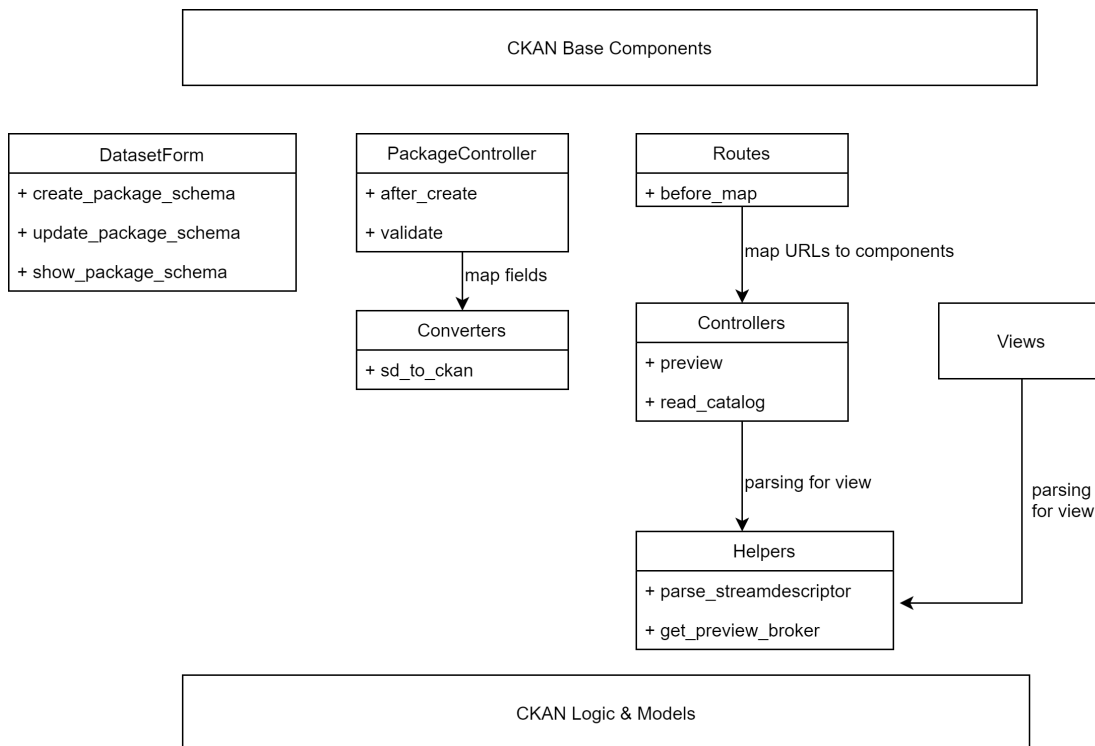


Figure 6.2: Plugin components

But a publisher may want to override those labels or use labels for predicates without a default labels. Thus, CKAN must accept labels from the publisher and store them somewhere. Analogous to storing the stream descriptor, we store the labels as a serialized string in a new field *labels*. To make it possible to completely disable default labels, we add a new boolean field to CKAN: *use_default_labels*.

Lastly, we add an extra field *original_source* to link to the original publication site of the data being streamed. We use this distinctly from the Dublin Core term *dct:source*⁶ because we link to the portal and not directly to the source data itself. Furthermore, it allows a user to search the catalog with such a source URL to check if there is a web stream for the corresponding dataset of an Open Data Portal.

In summary, following dataset fields are created on CKAN:

- **streamdescriptor** the stringified version of the stream descriptor.
- **labels** the stringified version of the labels.
- **use_default_labels** boolean for using the catalog's preset labels
- **original_source** the page for the original data source

⁶http://udfr.org/docs/onto/dct_source.html (accessed 2019-05-13)

6.3.2 Mapping to CKAN fields

In order to map the catalog fields that have a CKAN equivalent, we implement the package controller interface that hooks into the creation and modification of a dataset⁷. When the catalog entry is created or updated, the catalog parses the stream descriptor and maps those elements to the corresponding CKAN fields. The mapping is shown in Table 6.1.

RDF Class	Property	CKAN dataset field
vocals:RDFStream	dct:title	title
vocals:RDFStream	dct:description	notes
vocals:RDFStream	dct:publisher	publisher
vocals:StreamEndpoint	dct:license	license_url

Table 6.1: RDF stream description to CKAN mapping

The title, description and the publisher are extracted from the data the publisher pushes to the catalog. We save the *dct:license* in the *license_url* field because we expect a resource identifier pointing to the license of the stream. On top of the license URL, CKAN provides three additional license fields: (i) *license_id*: for CKAN internal purposes, (ii) *license_title*: a human-readable version of the license and (iii) *isopen*: a boolean to indicate whether the dataset is open according to the Open Knowledge Definition⁸. We develop a function to infer those three fields from the given *license_url* and populates the fields during the creation of a dataset.

6.3.3 Displaying the Stream Descriptor

The stream descriptor (or *sGraph* in Mauri et al. (2016)) contains all the metadata exposed by TripleWave. It is accessed by two different parts of the catalog: when TripleWave registers or updates itself at the catalog, and when the catalog is displayed on the web page or retrieved through the CKAN web API.

Since a stream description does not only contain information about the dataset, but also about the streaming service and about endpoints, we cannot list them in a single metadata table like CKAN does. We thus create a table of field-value pairs for each subject in the stream description. The tables for each subject are displayed on the primary page of the corresponding dataset on CKAN. Since we do not store the other metadata values directly in a CKAN field, we need to parse the stream descriptor each time a dataset page is rendered by the catalog. This might not be optimal from a performance point of view. But this enables the catalog to show more than a fixed set of triples, which would not be possible if we only used CKAN fields. The stream descriptor may for example contain an arbitrary number of endpoints. We parse the serialized

⁷<https://docs.ckan.org/en/2.8/extensions/plugin-interfaces.html#ckan.plugins.interfaces.IPackageController> (accessed 2019-06-10)

⁸<https://docs.ckan.org/en/ckan-2.7.3/api/legacy-api.html> (accessed 2019-05-13)


stream descriptor with the RDF library *rdflib*⁹. The library allows us to navigate the graph and iterate over triples that match a specified pattern of subject, predicate and object. We then aggregate all field-value pairs for each subject separately and send them to the template. We provide an alternative view in RDF, namely in JSON-LD and Turtle (see also Section 6.3.4).


Since the triples have no inherent order, we show the higher-level objects in the JSON-LD formatting of the stream description first. Thus we use the order *vocals.StreamDescriptor* > *vocals.RDFStream* > *vocals.StreamEndpoint* > *the rest*. Since a catalog entry is a stream descriptor that describes an RDF stream that has a set of endpoints, the order is somewhat natural as well.

In the web page view of the catalog entry we prefer labels over URIs for all predicates. Labels pushed by the publisher (e.g. TripleWave) have higher precedence than the default labels we defined in the catalog. The default labels are mostly the terms stripped from their respective prefix URI. For example *vocals:hasEndpoint* becomes just *has Endpoint*. A catalog entry of a selected web stream is shown in Figure 6.3.


⁹<https://github.com/RDFLib/rdflib> (accessed 2019-05-13)

Parking Information Zurich



Organization



webstreams
.. [read more](#)


Original Source

[Link](#)


License

[Creative Commons CCZero](#)

OPEN

DATA

Parking Information Zurich

This stream delivers real-time information about vacant parking spaces in the car parks of Zurich.

[Preview of the Stream](#)

Stream Descriptor

Field	Value
dataset	http://streamreasoning.org/cpzh/
tBoxLocation	http://purl.oclc.org/NET/ssnx/ssn
type	Catalog Service
type	Stream Descriptor

RDF Stream

Field	Value
has Endpoint	Endpoint1
has Endpoint	Endpoint2
has Endpoint	Endpoint3
source	https://data.stadt-zuerich.ch/dataset/parkleitsystem
type	RDF Stream

Endpoint1

Field	Value
access url	mqtt://localhost:1883
format	json-ld
protocol	mqtt
topic	cpzh
type	Stream Endpoint

Figure 6.3: The stream description as shown on the catalog

6.3.4 Machine-readable Version

For the purpose of exposing the catalog data to machines, we provide the original RDF representation of the stream in the JSON-LD and Turtle formats. Contrary to the table view of subjects, we move the RDF representation to a separate web page. The pages can be reached by appending *.ttl* or *.json-ld* to the URL of the dataset page. We set the content-type header to "application/json" and "text/turtle" respectively in order to facilitate the download of the metadata by machines. The final endpoint has the form:

```
https://<ckan-host>/dataset/{dataset_id}.{format}
```

In order to create a new page on CKAN, we must modify the routes as shown in Figure 6.1 by implementing the CKAN routes interface. The routes interface connects a route with a function that then returns the catalog and sets the appropriate headers for the response.

6.3.5 Creating a preview of the Stream

The preview of the stream is supposed to give the user an idea of the data being streamed and also gives away the kind of triples that are being streamed. It is directly accessible through the web browser on the catalog. The preview also serves as an easy access point for users unfamiliar with building WebSocket and MQTT clients. The user is free to pause or stop the stream at any time and can cycle through past messages. Because the broker stores the last message, the user does not have to wait for the next event message to arrive. We use the MQTT broker because we have direct access to it (unlike Kafka that requires a connector) and because the broker makes it easy to retrieve the last message. Retrieving the last message is especially helpful for streams that publish data in long intervals. Since TripleWave includes a timestamp in the stream element, it is still possible to distinguish between old messages and messages that have just arrived.

The preview of the stream is a JavaScript client that connects to the MQTT broker in the backend. This way we move some of the processing required to the client and off the server. We use a MQTT library that works in the browser and subscribes to our MQTT broker through a WebSocket connection. The JavaScript library used for the client is `mqtt.js`¹⁰. Since the browser cannot open a MQTT connection directly, our broker must be able to communicate over WebSockets. In this case the browser wraps the MQTT packet into a WebSocket packet and lets the WebSocket protocol handle the data transfer.

In the backend we utilize the Node.js module `aedes`¹¹ as our MQTT broker. The broker can handle direct MQTT connections as well as WebSocket connections. We use Aedes because it provides WebSockets off the shelf. The mosquito broker on the other hand would require us to enable WebSocket in the compilation settings and then require

¹⁰<https://github.com/mqttjs/MQTT.js> (accessed 2019-05-08)

¹¹<https://www.npmjs.com/package/aedes> (accessed 2019-05-08)

us to compile our own mosquitto package. Mosquitto's WebSocket support is currently disabled at compile time¹².

Figure 6.4 shows the preview of the stream on the catalog page.

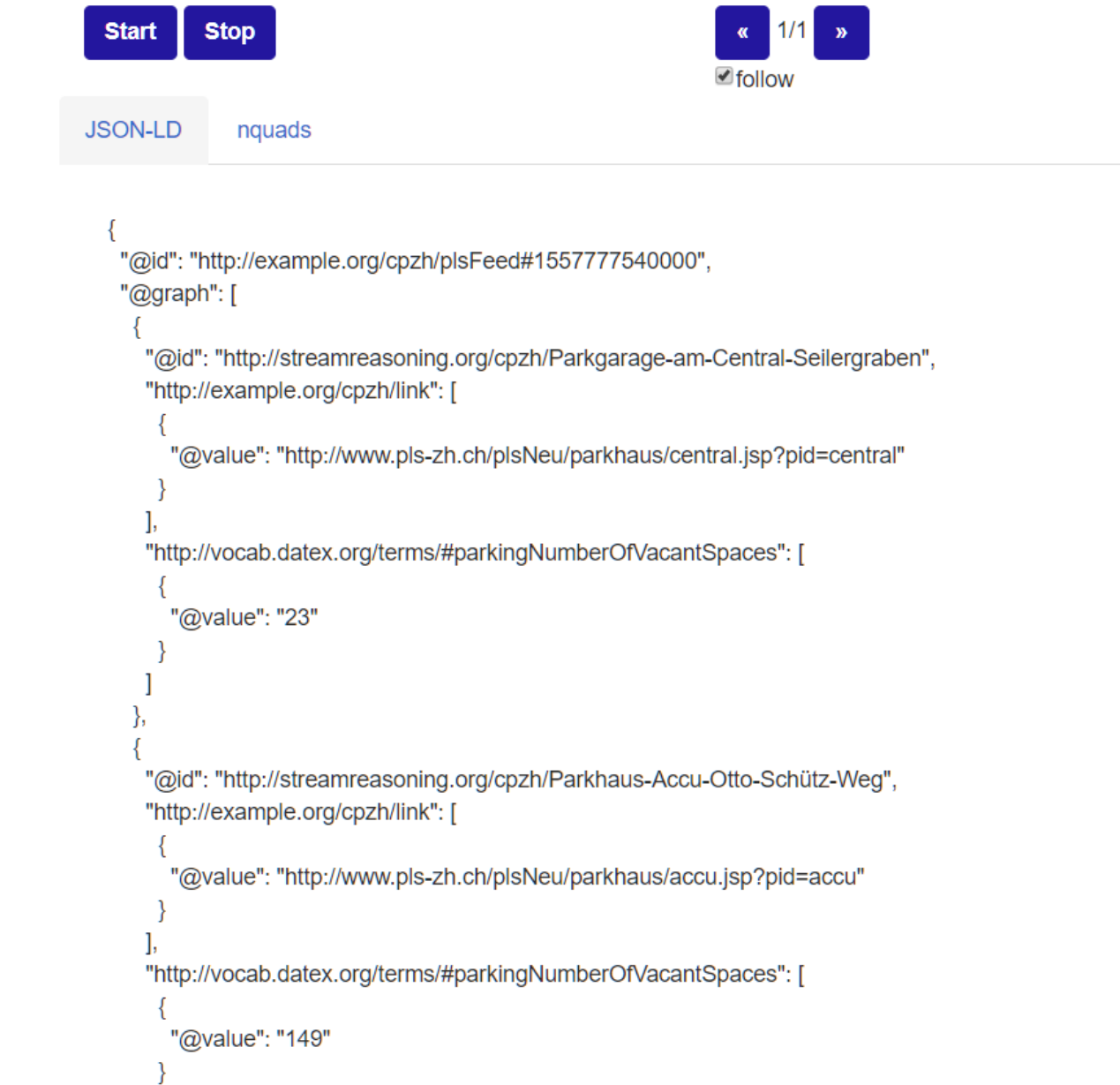


Figure 6.4: Preview of the stream on the catalog

¹²<https://mosquitto.org/man/mosquitto-conf-5.html> (accessed 2019-04-18)

6.4 Additional Modifications

This section lists minor modifications of the catalog web page. We hide unused CKAN features, add an about page, and change the visual appearance of CKAN. All of the changes are done on the template level (see Figure 6.2) by creating template files that override the defaults. The names of the templates can be retrieved from the CKAN repository on Github¹³.

6.4.1 Accessibility

To be compliant with the Open Data principles, we do not require registration for viewing datasets and subscribing to the stream. Thus, we remove the login and register buttons from the CKAN view. Nevertheless, we must leave the login-page accessible because the API key for creating a new catalog entry on CKAN is only retrievable via the web interface¹⁴.

The homepage of the catalog serves as the central portal to explore those datasets available on the site. We leave the homepage of the catalog unmodified for the most part. It shows the most recently added datasets and search box prominently shown in the middle. The modified homepage is shown in Figure 6.5. The user may either search for a dataset using the search box or navigate to a list of all available datasets. In addition, there is a list of selected datasets on the homepage. The most recent datasets are selected by default, but it is possible to show a set of "featured" datasets.

6.4.2 About page

The about page provides a short description of CKAN. We also use this page to describe this project and the related work in Bernhaut (2018) and Muntwyler (2017) in order to give a full overview of the web streams featured in the catalog. Lastly, we give practical advice for accessing the web streams featured in the catalog. We do this by showing an example for available each access method.

6.4.3 Visual Appearance

Figure 6.5 shows the homepage of the catalog. The primary goal of the visual part is to hide CKAN features that are not available to users of our site. Apart from the login and registering we also hide the unused buttons for the manual creation and editing of a dataset to reduce clutter on the web page. Furthermore, we remove the social section (Facebook, Twitter) and follower count for the datasets.

We also want to align the catalog website's visuals to the university's websites, since the catalog is published on the servers of the university. In order to create a similar styling to the universities current website¹⁵, we adopt the color palette of the website

¹³ <https://github.com/ckan/ckan/tree/master/ckan/templates> (accessed 06-28-2019)

¹⁴ <https://docs.ckan.org/en/ckan-2.7.3/api/#authentication-and-api-keys> (accessed 2019-06-04)

¹⁵ <https://www.uzh.ch/> (accessed 2019-05-16)

and include the university's logo on top of each page. For specifying the style, we create our own CSS file and host it within our CKAN instance¹⁶.

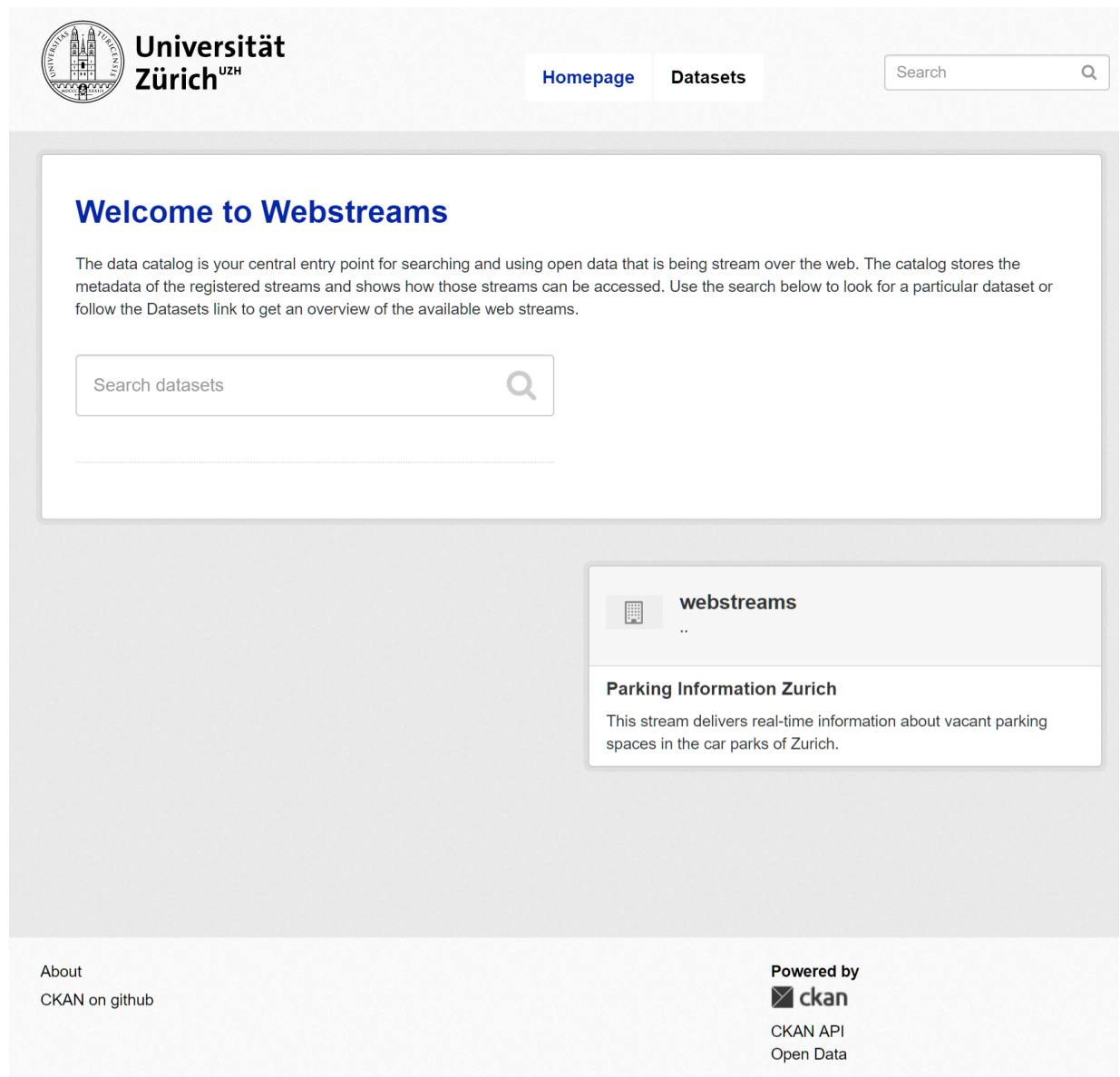


Figure 6.5: The catalog homepage serving as an entrypoint for discovering web streams

¹⁶<https://docs.ckan.org/en/2.8/theming/fanstatic.html> (accessed 2019-05-16)

6.5 Deploying the Catalog

CKAN uses the Web Server Gateway Interface (WSGI) to forward requests on the web server to the CKAN web application. We use the Apache HTTP Server¹⁷ as the web server.

The whole deployment process is documented in detail in the appendix section of this thesis. A description of this and other available deployment methods is accessible on CKAN's documentation page¹⁸.

¹⁷<https://httpd.apache.org/> (accessed 2019-05-16)

¹⁸<https://docs.ckan.org/en/ckan-2.7.3/maintaining/installing/deployment.html> (accessed 2019-05-16)

Datasets

In this chapter we describe Open Government datasets that we selected to be streamed by TripleWave and to be pushed to the catalog. The goal is to demonstrate the capabilities of the catalog and to update the data surveys of the previous work. We focus on Open Government Data because the datasets are published under licenses that allow free distribution and transformation of the data.

Since [Muntwyler \(2017\)](#) and [Bernhaut \(2018\)](#) already created extensive surveys on Open Government Data, we focus our search on datasets that have been added since then. The method and results are described in Section 7.1. Afterwards we select the datasets to be streamed with TripleWave and to be published on the catalog. In Section 7.2 we transform and deploy the datasets to RDF following the 5-star deployment scheme for Open Data ([Berners-Lee, 2006](#)).

7.1 Survey

In this section we go through the list of datasets found in the previous work of [Muntwyler \(2017\)](#) and [Bernhaut \(2018\)](#) and update the metadata of their listed datasets. We then add new datasets to the survey and create a preliminary list of datasets to be streamed with TripleWave and to be included in the catalog.

7.1.1 Updating previous surveys

There were necessary updates to the survey. For example, the license of "Luftmessnetz: aktuelle Messdaten Wien"¹ went from CC BY 3.0 AT to CC BY 4.0 AT. Some resources are no longer available under the listed addresses. All three resource addresses for "Wartezeiten in den Magistratischen Bezirksämtern Wien"² are not longer available. A search on the open data portal did not yield any new address URL either. The

¹<https://www.data.gv.at/katalog/dataset/d9ae1245-158e-4d79-86a4-2d9b3defbedc> (accessed 2019-05-19)

²<https://www.data.gv.at/katalog/dataset/e38cdef5-f993-4e6f-919e-ac68d26d727d> (accessed 2019-05-19)

”Parkplätze API (Beta)”³ changed the endpoint URL and now requires an API key to access the data.

7.1.2 Adding new Datasets

We extend the surveys by searching on the following open data portals.

- <https://opendata.swiss> the central catalogue for Open Government Data made available to the general public.
- <https://data.stadt-zuerich.ch> the data catalogue for the city of Zurich
- <https://opentransportdata.swiss> data on public transport in Switzerland
- <https://data.sbb.ch/pages/home/> the SBB open data portal
- <https://www.data.gv.at> the Austrian open data portal
- <https://www.govdata.de> the German open data portal

We search for new datasets with the keywords on the open data portals listed below:

- real-time
- Echtzeit (real-time)
- Aktuell (latest)
- Messwerte (measurements)
- Arrival, Departure
- General Transit Feed Specification (GFTS)⁴

In our results we do not include datasets with less than three stars according to Berners-Lee’s five stars of open data⁵. This excludes documents in proprietary and formats such as *pdf* and *xls* format as well as data published under a non-open license. We do this to narrow down our candidates for streaming.

We use the last two elements to look for transport information datasets that are frequently updated. Transit Feed Specification (GFTS) is a common format for public transportation schedules linked with geographical information. It is for example used by *maps.google.com* and *map.geo.admin.ch* to display real-time transport information on top of a map. Given the recency and update frequency of such data records, they would make a good web stream.

³<https://mcloud.de/web/guest/suche/-/results/detail/1cc0f0c6-f0fd-45a6-97f7-ca2061dc0eef> (accessed 2019-05-19)

⁴<https://developers.google.com/transit/gtfs/> (accessed 2019-05-19)

⁵5stardata.info (accessed 2019-05-21)

On the Swiss portals we find two new datasets. Table 7.1 shows the metadata for performance measurements of the fiber optic network of the city of Zurich. The measurements are 15 in apart and are published daily. Because of this discrepancy between the publication and the measurement interval we do not consider this dataset for streaming.

Title	Zürich Glasfasernetz Leistungsdaten
Publisher	Elektrizitätswerke des Kantons Zürich (EKZ)
Description	Network Performance Data (Up- and Downstream) in the fiber optic network of the city of Zurich in bits
License	https://opendefinition.org/licenses/cc-zero/
Source	https://data.stadt-zuerich.ch/dataset/ewz_leistungsdaten_zuerinet
Publication Frequency	daily
Time interval of records	15 minutes
Format	CSV

Table 7.1: Zurich fiber optic network performance data

The Rail Traffic Information (Table 7.2) dataset provides a web API for retrieving the most important traveler information on public transport traffic in Switzerland. Examples are cancelled or diverted trains. Since such data is updated frequently and contains pertinent information for travelers, we add the dataset to our catalog of web streams. We verify that the license allows free processing and publication of the data.

Title	Rail Traffic Information
Publisher	SBB Personenverkehr
Description	Rail traffic information that is updated every five minutes.
License	https://data.sbb.ch/terms/terms-of-use
Source	https://data.sbb.ch/explore/dataset/rail-traffic-information/information/
Publication Frequency	every five minutes
Time interval of records	varying
Format	JSON

Table 7.2: Rail traffic information

On the Austrian Open Data portal, we find a dataset providing real-time public transport information in GTFS real-time (Table 7.3) and data gathered by several kind of measurement stations located in the upper regions of Austria (Table 7.4). On the German Open Data portal, we find three new datasets. Locations of public rental bikes in Table 7.5, Charging stations for electrical vehicles shown in Table 7.6 and real-time transport information in Table 7.7.

Title	GTFS Real-Time Vienna
Publisher	City of Vienna
Description	Real-time information about Vienna's public transport.
License	https://creativecommons.org/licenses/by/4.0/
Source	https://www.data.gv.at/katalog/dataset/add66f20-d033-4eee-b9a0-47019828e698
Publication Frequency	continuously
Time interval of records	varying
Format	JSON

Table 7.3: Vienna GTFS

Title	Luftgüte- und meteorologische Messwerte
Publisher	Land Oberösterreich
Description	Current measurements of air pollutants, temperature and other meteorological parameters.
License	https://creativecommons.org/licenses/by/3.0/at/
Source	https://www.data.gv.at/katalog/dataset/1acd5fbf-1f85-4692-847c-45fb2fb0606c
Publication Frequency	30 minutes
Time interval of records	30 minutes
Format	CSV, JSON

Table 7.4: Air quality and meteorological data in Upper Austria

7.2 Deployment

For the datasets deployed by Muntwyler (2017) and Bernhaut (2018) we adopt their proposed mapping and get the metadata from the open data portals. Since the transformation of datasets is not the main part of our thesis, we chose to take some of the existing mappings from Muntwyler (2017) and Bernhaut (2018). Additionally, we create mappings for some of the new datasets from Section 7.1. Each retrieval of a dataset generates a new RDF graph that contains the elements mapped to RDF. We follow Barbieri and Della Valle (2010)'s proposal to map each stream item to the RDF graph with the schema http://example.org/stream_name/timestamp. In JRML this is defined

Title	Real-time location of Rental Bikes
Publisher	Stadtwerke Bonn
Description	Real-time location information about rental bikes in the city of Bonn.
License	http://www.opendefinition.org/licenses/cc-by
Source	https://www.govdata.de/web/guest/suchen/-/details/standorte-von-fahrradern-realtime-fahrradmietsystem
Publication Frequency	continuously
Time interval of records	N/A
Format	XML

Table 7.5: Public bike locations in the city of Bonn

Title	Locations and Availability of Charging Stations
Publisher	Stadtwerke Bonn
Description	Location and real-time availability information about charging stations for electrical vehicles.
License	http://www.opendefinition.org/licenses/cc-by
Source	https://www.govdata.de/web/guest/suchen/-/details/standorte-elektrotankstellen-e-ladesaulen-realtime-verfuegbarkeit
Publication Frequency	continuously
Time interval of records	N/A
Format	JSON

Table 7.6: Locations and availability of charging stations

Title	GFTS Real-Time Rhein-Sieg Transport Network
Publisher	Verkehrsverbund Rhein-Sieg
Description	GFTS real-time data for the Rhein-Sieg transport network and neighbouring networks. Updated every minute.
License	http://www.opendefinition.org/licenses/cc-by
Source	https://www.govdata.de/web/guest/suchen/-/details/echtzeitdaten-opnv-bus-und-bahn
Publication Frequency	every minute
Time interval of records	N/A
Format	Protocol Buffers, txt

Table 7.7: GFTS real-time Rhein-Sieg Transport Network

in the graphMap configuration element. We already described pushing the metadata in Section ?? and Section 6.3.

7.2.1 Selection

In our selection process for streaming datasets we look for following characteristics: First the dataset format should be supported by JRML (namely JSON, XML and CSV). This is technically not necessary since JRML allows full-source transformation, but this would add more complexity to our transformation function in the JRML configuration. JRML allows a transformation by letting us define a function that gets then applied to the retrieved source right before passing it to the transformer. Secondly, the dataset should be updated at least once an hour with preference given to datasets that are updated more frequently.

In order to test our new features devised from the JRML requirements from Section 3.1.2 the dataset should contain invalid values (e.g. "null", "N/A" or "") or support querying the endpoint by encoding the query parameters in the URL. In the table they are labelled as 3a and 3b respectively. Table 7.8 lists the selected datasets. Table 7.9 contains the links to the original catalog entry on the respective Open Data portals.

Dataset	Characteristics
Car Parking Zurich (CPZH) ¹	1, 2, 3a
Zueri Rollt (ZR) ¹	1, 2
Fine Dust PM10 (FAU) ²	1, 2, 3a
Global Radiation (GAU) ²	1, 2, 3a
Rainfall Tirol (NTAU) ²	1, 2
Car Parking Kleve (PKDE) ²	1, 2
Recent Ozone Measurements Austria (AOAU)	1, 2
Rail Traffic Information (RTI)	1, 2, 3b
Air Quality Zurich (AQZH)	1, 2, 3a
Weather Stations Zurich (WSZH)	1, 2, 3b
Locations and Availability of Charging Stations	1, 2

Table 7.8: Selected datasets: some mappings from ¹Muntwyler (2017) and ²Bernhaut (2018)

Dataset	Source
Car Parking Zurich (CPZH)	https://data.stadt-zuerich.ch/dataset/parkleitsystem
Zueri Rollt (ZR)	https://data.stadt-zuerich.ch/dataset/mietvelo-verfuegbarkeit
Fine Dust PM10 (FAU)	https://www.data.gv.at/katalog/dataset/8b057f32-1312-40ae-ae51-9aa0a0d372ca
Global Radiation (GAU)	https://www.data.gv.at/katalog/dataset/f9e40f30-8ac6-43e2-9ee7-f72b712ea9e1
Rainfall Tirol (NTAU)	https://www.data.gv.at/katalog/dataset/44720e90-c2de-497b-8162-3810206dd011
Car Parking Kleve (PKDE)	https://www.govdata.de/web/guest/suchen/-/details/parkleitsystem-stadt-kleve
Recent Ozone Measurements Austria (AOAU)	https://www.data.gv.at/katalog/dataset/8b3b3cdf-2be6-4f0b-8c86-f6be67e5b002
Rail Traffic Information (RTI)	https://data.sbb.ch/explore/dataset/rail-traffic-information/information/
Air Quality Zurich (AQZH)	https://data.stadt-zuerich.ch/dataset/luftqualitaet-stunden-aktuelle-messungen
Weather Stations Zurich	https://data.stadt-zuerich.ch/dataset/sid_wapo_wetterstationen
Locations and Availability of Charging Stations	http://ckan.www.open.nrw.de/dataset/0d701d4a-1255-4301-b362-d08f375e8e1a

Table 7.9: Links to the original data source

7.2.2 Railway Traffic Information

The Railway Traffic Information (RTI) dataset describes incidents on the rail traffic system in Switzerland. Incident examples are interruptions, cancellations, diversions and construction work.

The *subject* for the transformation in this case is the unique record id generated by the publisher and represents a single message about an incident. Updates concerning an already existing incident get a new record id and are published as a new record. We therefore publish only data that's been newly added to the dataset since the last element streamed (now minus five minutes) to avoid duplicate messages and do not need to check for records older than five minutes.

In order to test the dynamic URL generation described in Section 5.4.2 we filter the records by date on the server side of the API. The API does not support filtering by date natively, but allows full text search and comparison on individual data fields. Since the date is in the form *YYYY-MM-DD* the comparison operators can be used to compare dates. Thus, we append the following query parameter to the end of the URL: *?q=(published>{getDate})*. "getDate" in that case is a function that returns the current date minus five minutes right before JRML schedules the GET request.

We use the Dublin Core Terms⁶ for the metadata. For *validitybegin* and *validityend* we did not find a suitable vocabulary and thus used our own. All terms created by us have the form <http://streamreasoning.org/<dataset-name>/>. Table 7.2 shows the mapping of the dataset.

Subject	Predicate	Object
srrti:{recordid}	dct:title	{title}
srrti:{recordid}	dct:description	description
srrti:{recordid}	dct:issued	{record_timestamp}^^xsd:date
srrti:{recordid}	dct:publisher	{author}
srrti:{recordid}	dct:source	{link}
srrti:{recordid}	srrti:validitybegin	{validitybegin}^^xsd:date
srrti:{recordid}	srrti:validityend	{validityend}^^xsd:date
srrti:{recordid}	dct:description	{description_html}

Table 7.10: The mapping of the RTI dataset:

prefix srrti: <http://streamreasoning.org/rti/>

7.2.3 Air Quality Zurich (AQZH)

The AQZH⁷ dataset contains hourly updated measurements of the air quality measured at multiple locations in Zurich. We follow Bernhaut (2018)'s approach and use the

⁶<http://purl.org/dc/terms>

⁷<https://data.stadt-zuerich.ch/dataset/luftqualitaet-stunden-aktuelle-messungen> (accessed 2019-05-27)

QBOAirbase⁸ vocabulary to describe the air quality in this dataset. This ontology proposed by Galárraga et al. (2017) uses an *observation* to denote a measurement of a single pollutant at a specific location at a specific time. We therefore use the QBOAirbase vocabulary to describe the air quality measurements. For the geospatial location of the station we use the Basic Geo Vocabulary (WGS)⁹. The mapping is shown in Table 7.11.

Subject	Predicate	Object
air:observation/AU{timestamp}#{station}	air:schema/station	sraqzh:{station}
air:observation/AU{timestamp}#{station}	wgs84_pos:lat	{latitude}
air:observation/AU{timestamp}#{station}	wgs84_pos:long	{longitude}
air:observation/AU{timestamp}#{station}	air:schema/pm10	{measurement}^^xsd:decimal
air:observation/AU{timestamp}#{station}	air:schema/O3	{measurement}^^xsd:decimal
air:observation/AU{timestamp}#{station}	air:schema/NO2	{measurement}^^xsd:decimal

Table 7.11: The mapping of the AQZH dataset:

prefix sraqzh: <http://streamreasoning.org/aqzh/>

prefix air: <http://qweb.cs.aau.dk/airbase/>,

prefix wgs84_pos: http://www.w3.org/2003/01/geo/wgs84_pos#

7.2.4 Weather Stations Zurich (WSZH)

The WSZH¹⁰ datasets contain measurements of the weather stations in *Mythenquai* and *Tiefenbrunnen*. For fetching the data, we use the provided web API *Tecdottir*¹¹ to query for the most recent measurements for each station. This way we do not have to download the whole dataset repeatedly. We accomplish this by setting the maximum age for the records to be retrieved to ten minutes. In order to pass the current time to the fetch request, we use the query string feature introduced in Section 5.4.2 that sets the *startDate* parameter to *now - 10 min*. We did not find any suitable vocabulary for this dataset. Thus, we create new terms in the <http://streamreasoning.org/wszh/> namespace. The mapping is shown in Table 7.12.

7.2.5 Locations and Availability of Charging Stations (LACS)

This dataset holds information about the locations and availability of charging stations for electrical vehicles close to the city of Bonn. Because each station in this dataset has multiple chargers and each charger has a unique ID, we create a subject for each charger. Therefore, we create a a subject map for the stations and for the chargers called Electric Vehicle Supply Equipment (EVSE)¹². The predicate *hasEVSE* indicates

⁸<http://qweb.cs.aau.dk/qboairbase/> (accessed 2019-05-27)

⁹<https://www.w3.org/2003/01/geo/>

¹⁰https://data.stadt-zuerich.ch/dataset/sid_wapo_wetterstationen (accessed 2019-05-27)

¹¹<https://tecdottir.herokuapp.com/docs/> (accessed 2019-05-27)

¹²<https://support.chargecloud.de/hc/de/articles/115002326585-Was-ist-eine-EVSE-Operator-ID-> (accessed 2019-06-10)

Subject	Predicate	Object
srwszh:{timestamp}#{station}	srwszh:air_temperature	{air temperature} ^^xsd:decimal
srwszh:{timestamp}#{station}	srwszh:air_temperature	{unit}
srwszh:{timestamp}#{station}	srwszh:water_temperature	{water temperature} ^^xsd:decimal
srwszh:{timestamp}#{station}	srwszh:water_temperature_unit	{unit}
srwszh:{timestamp}#{station}	srwszh:wind_speed_avg_10min	{wind speed} ^^xsd:decimal
srwszh:{timestamp}#{station}	srwszh:wind_speed_unit	{unit}
srwszh:{timestamp}#{station}	srwszh:wind_direction	{wind direction} ^^xsd:decimal
srwszh:{timestamp}#{station}	srwszh:wind_direction_unit	{unit}

Table 7.12: The mapping of the WSZH dataset:

prefix srwszh: <http://streamreasoning.org/srwszh/>

to which station a charger belongs. For the address of the charging stations we use the schema.org¹³ vocabulary. For the location we use WGS again. We did not find any vocabulary for electrical vehicle charging stations. Therefore, we create new terms for *hasEVSE* and *status*. Table 7.13 shows the mapping.

Subject	Predicate	Object
srlocs:station/{timestamp}#{id}	schema:name	{name}
srlocs:station/{timestamp}#{id}	schema:streetAddress	{address}
srlocs:station/{timestamp}#{id}	schema:addressLocality	{city}
srlocs:station/{timestamp}#{id}	schema:postalCode	{postal_code}
srlocs:station/{timestamp}#{id}	wgs84_pos:lat	{latitude}
srlocs:station/{timestamp}#{id}	wgs84_pos:long	{longitude}
srlocs:station/{timestamp}#{id}	srecs:hasEVSE	{evse_uid}
srlocs:evse/{timestamp}#{evse_uid}	srecs:status	{status}

Table 7.13: The mapping of the charging stations dataset:

prefix srlocs: <http://streamreasoning.org/lacs/>,

prefix schema: <http://schema.org/>

¹³<https://schema.org>

Conclusions

Data on the web has become more dynamic in the recent years and has been published as a stream of data, rather than a static dataset. Consequently, following the Linked Data principles is no longer enough. Further, the access methods for data have changed and new requirements for the description of web streams have emerged. TripleWave, a streaming framework, addresses this issue and enables streaming linked data on the web. TripleWave makes use of the VoCaLS vocabulary for providing metadata about the stream. With the technology in place, we aim to publish a set of streams on the web and provide a central catalog that hosts the stream descriptions. The catalog serves as a central entry point for discovering web streams and enables sharing data amongst people. This thesis builds on the work of [Muntwyler \(2017\)](#) and [Bernhaut \(2018\)](#) who published web streams with TripleWave among other things. This thesis main contribution is the creation of a catalog of web streams and interfacing TripleWave with the catalog. The catalog itself remains independent from TripleWave, and accepts input from other services as well.

Because one of the evaluated frameworks already met some of the requirements for the catalog, it was not necessary to build a catalog from scratch. We created the catalog by developing an extension for the Comprehensive Knowledge Archive Network (CKAN) that is widely used as a catalog for Open Government Data [Shadbolt et al. \(2012\)](#). On the backend we used TripleWave and its JRML module for fetching, transforming and publishing the original dataset sources.

For creating the catalog, we followed the schema of static datasets that are published on CKAN. Each stream is represented on CKAN by a dataset entry. Such an entry contains all the metadata for a specific stream. The metadata includes information about the stream and about the data itself. Using the catalog, a user can find out how to connect to the stream. TripleWave publishes its metadata as an RDF graph serialized in JSON-LD and pushes it to the catalog. This results in either a new or updated entry in the catalog. We mapped some of the metadata directly to existing CKAN fields (e.g. title, description and license). Because not all metadata did fit directly into CKAN's default field-value schema, we extended the schema and the web page view of the catalog. We did this by creating a table of field-value pairs for each subject in TripleWave's stream description.

We created a stream preview that connects the user's browser directly to the MQTT broker on the backend using a JavaScript library. We did this to lower the barrier for

accessing web streams and to show consumers the schema of the stream without requiring a separate client. Since some streams generate new event items long times apart and consumers should not have to wait for a preview, we stored the latest retrieved message for each stream. The catalog also offers a description on how to connect to the published data streams.

In order to test and showcase the catalog we first searched for relevant datasets to be streamed. The surveys of Muntwyler (2017) and Bernhaut (2018) served as a starting point and were updated if necessary. Looking forward, we searched for new datasets to increase the survey of web streams. On the Swiss portals we found two new datasets that are suitable for streaming. Most of the recently published data has a static kind of nature with long update intervals. Consequently, we focused our search on the Austrian and German Open Government Data portals. In order to select the datasets to be streamed we first defined requirements based on JRML’s transformation capabilities. In the process of transforming datasets, we also identified new requirements for JRML. In order to test the new features of JRML we further favored datasets requiring the new functionality of JRML in our selection.

For the purpose of publishing the datasets through *TripleWave* we extended the vocabulary in two places. First, we introduce the VoCaLS vocabulary constructed by Tommasini et al. (2018). This enables *TripleWave* to describe its stream with a well-defined and published vocabulary that is partially an extension of the dataset catalog vocabulary *dcat*. In this work we focused on the VoCaLS-core module for metadata¹, but it should be noted that VoCaLS also supports other services and provides description for stream provenance as well. Secondly, we constructed the vocabularies for streaming the data itself. We reused existing vocabulary as much as possible. Only when we did not find any suitable terms, we created our owns. To increase the number of published datasets on the catalog we also took over some of the existing mappings of Muntwyler (2017) and Bernhaut (2018) that matched our selection criteria.

In the backend we deployed TripleWave instances. We sourced the output of TripleWave to two message brokers. Each stream item ends up as a JSON document published under a topic associated to a specific dataset. The MQTT broker directly listens to the MQTT output of TripleWave. We also adopted Muntwyler (2017)’s Kafka approach and transform the WebSocket output of TripleWave to the Kafka broker. We use a WebSocket producer to transform and forward the TripleWave output to the Kafka broker. On the client side we provide interfaces for EventSource and a WebSocket connection.

In conclusion, the catalog meets almost all of the requirements listed in Chapter 3. However, the catalog does not pull the metadata from TripleWave periodically (RC14), instead the metadata must be pushed by TripleWave. The catalog could be extended to support continuously polling TripleWave instances to check their liveliness. Further, the documentation for the CKAN plugin (RN1) is limited to this thesis and readme files in the source code folder. Regarding the creation of vocabularies for web streams, we faced the same challenges as Muntwyler (2017) and did not find suitable terms for every dataset, and thus had to construct our own vocabulary. Further, we did not

¹<https://ysedira.github.io/vocals/docs/core/index-en.html> (accessed 2019-05-30)

rigorously test the user experience of the catalog web site. However, CKAN's base templates are adjusted for different screen sizes and browsers. Visiting the pages on the latest Chrome and Firefox with a laptop and a mobile phone did not yield any breaking pages. Nevertheless, we cannot ensure compatibility with all browsers. For example, the preview of the stream requires a lot of text area and could lead to issues on devices with a small display resolution.

So far, the catalog enables discovery and access of web streams. The catalog view orders the known metadata terms in a logical way. Other terms are appended at the end of the page with no inherent order. For the future, the catalog could be extended by adding more common data catalog vocabulary to the catalog to fix the order of appearance. For example, VoCaLS has the terms for describing processing capabilities of a service. In addition, the catalog could be extended to allow the evaluation of simple queries on the web streams available. We think this further lowers the barriers to entry for accessing web streams. Another idea is to support more protocols for data sources. We found several datasets about real-time public transport information in the protocol buffers² format, and thus we think it is beneficial to support such format on TripleWave and/or JRML. We should further think about increasing the quality of the web streams on the catalog and reuse more existing vocabulary mapping the data source to stream items. Ultimately, the catalog depends on the quality of the entries as a whole.

²<https://developers.google.com/protocol-buffers/> (accessed 2019-06-28)

References

- Anicic, D., Fodor, P., Rudolph, S., and Stojanovic, N. (2011). Ep-sparql: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM.
- Auer, S., Berners-Lee, T., Bizer, C., Capadisli, S., Heath, T., Janowicz, K., and Lehmann, J., editors (2017). *Proceedings of the Workshop on Linked Data on the Web (LDOW)*, number 1809 in CEUR Workshop Proceedings, Aachen.
- Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. (2007). Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer.
- Ayala, V. A. A., Cheng, T.-N., Alzoghbi, A., and Lausen, G. (2017). Learning to identify complementary products from dbpedia. In [Auer et al. \(2017\)](#).
- Balduini, M., Della Valle, E., Dell’Aglia, D., Tsytsarau, M., Palpanas, T., and Confalonieri, C. (2013). Social listening of city scale events using the streaming linked data framework. In *International Semantic Web Conference*, pages 1–16. Springer.
- Barbieri, D. F. and Della Valle, E. (2010). A proposal for publishing data streams as linked data—a position paper.
- Barnaghi, P., Presser, M., and Moessner, K. (2010). Publishing linked sensor data. In *CEUR Workshop Proceedings: Proceedings of the 3rd International Workshop on Semantic Sensor Networks (SSN), Organised in conjunction with the International Semantic Web Conference*, volume 668.
- Berners-Lee, T. (2006). Linked data. <https://www.w3.org/DesignIssues/LinkedData.html>. Retrieved 2019-03-08.
- Bernhaut, E. (2018). Publication of linked data streams on the web. Bachelor Thesis, University of Zurich.
- Bizer, C. (2009). The emerging web of linked data. *IEEE intelligent systems*, 24(5):87–92.

- Bizer, C., Heath, T., and Berners-Lee, T. (2011). Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI Global.
- Boyle, D. E., Yates, D. C., and Yeatman, E. M. (2013). Urban sensor data streams: London 2013. *IEEE Internet Computing*, 17(6):12–20.
- Bradner, S. (1997). Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, RFC Editor.
- Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117.
- Carrara, W., Radu, C., and Vollers, H. (2017). Open data maturity in europe 2017. Retrieved from the European Data Portal website: https://www.europeandataportal.eu/sites/default/files/edp_landscaping_insight_report_n3_2017.pdf.
- Dell’Aglío, D., Le Phuoc, D., Lê Tuán, A., Ali, M. I., and Calbimonte, J.-P. (2017). On a web of data streams. In *DeSemWeb@ ISWC*.
- 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)*.
- Heath, T. and Bizer, C. (2011). Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136.
- Hunkeler, U., Truong, H. L., and Stanford-Clark, A. (2008). Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE’08)*, pages 791–798. IEEE.
- Le-Phuoc, D., Dao-Tran, M., Parreira, J. X., and Hauswirth, M. (2011). A native and adaptive approach for unified processing of linked streams and linked data. In *International Semantic Web Conference*, pages 370–388. Springer.
- Maali, F., Cyganiak, R., and Peristeras, V. (2010). Enabling interoperability of government data catalogues. In *International Conference on Electronic Government*, pages 339–350. Springer.
- Maali, F., Erickson, J., and Archer, P. (2014). Data catalog vocabulary (dcat). *W3c recommendation*, 16.
- Mauri, A., Calbimonte, J.-P., Dell’Aglío, D., Balduini, M., Brambilla, M., Della Valle, E., and Aberer, K. (2016). Triplewave: Spreading rdf streams on the web. In *International Semantic Web Conference*, pages 140–149. Springer.
- Morsey, M., Lehmann, J., Auer, S., Stadler, C., and Hellmann, S. (2012). Dbpedia and the live extraction of structured data from wikipedia. *Program*, 46(2):157–181.

- Muntwyler, P. (2017). Increasing the number of open data streamson the web. Bachelor Thesis, University of Zurich.
- Murray-Rust, P. (2008). Open data in science. *Serials Review*, 34(1):52–64.
- Neumaier, S., Umbrich, J., and Polleres, A. (2017). Lifting data portals to the web of data. In Auer et al. (2017).
- Open Knowledge International (2015). The open definition 2.1.
- Sequeda, J. F. and Corcho, O. (2009). Linked stream data: A position paper. *TODO*.
- Shadbolt, N., O’Hara, K., Berners-Lee, T., Gibbins, N., Glaser, H., Hall, W., et al. (2012). Linked open government data: Lessons from data. gov. uk. *IEEE Intelligent Systems*, 27(3):16–24.
- Stone, M. and Aravopoulou, E. (2018). Improving journeys by opening data: the case of transport for london (tfl). *The Bottom Line*, 31(1):2–15.
- 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)*.
- Tommasini, R., Sedira, Y. A., Dell’Aglio, D., Balduini, M., Ali, M. I., Le Phuoc, D., Della Valle, E., and Calbimonte, J.-P. (2018). Vocals: Vocabulary and catalog of linked streams. In *International Semantic Web Conference*, pages 256–272. Springer.
- Ubaldi, B. (2013). Open government data: Towards empirical analysis of open government data initiatives. *OECD Working Papers on Public Governance*, No. 22.
- Valsecchi, F., Abrate, M., Bacciu, C., Tesconi, M., and Marchetti, A. (2015). Dbpedia atlas: Mapping the uncharted lands of linked data. In Bizer, C., Auer, S., Berners-Lee, T., and Heath, T., editors, *Proceedings of the Workshop on Linked Data on the Web (LDOW)*, number 1409 in CEUR Workshop Proceedings, Aachen.

A

Appendix

This section describes the installation and deployment of the catalog and TripleWave. The files are located on the CD in the **repositories** folder and path references in relation to the CD in this section are relative to this path. This has been deployed on a Debian and Ubuntu VM.

A.1 CKAN

The catalog is located at <http://130.60.155.163> and the development version is reachable at <http://130.60.155.163:5000>. In order to install and use the catalog we need to do:

1. Install CKAN from source
2. Install the CKAN plugin and register it
3. Create an admin account on CKAN

A.1.1 Install CKAN from source

For creating the catalog we use CKAN 2.8.2. We follow <https://docs.ckan.org/en/2.8/maintaining/installing/install-from-source.html>. The base folder does not necessarily have to be in the `/usr/lib` folder and can be located anywhere. We also do not use `solr-jetty` and instead use Solr 6.5.0.

A.1.2 Install the CKAN plugin

The plugin is named **webstreamsuzh**.

On the CD the repository is placed in the folder **ckan-stream**. The plugin should be installed with pip to maintain the correct folder structure. If using git or copy from CD: the folder name must be `ckanext-webstreamsuzh` and is in the CKAN **src** folder (e.g. `/usr/lib/ckan/default/src`).

```

. /usr/lib/ckan/default/bin/activate

# IF prod
pip install -e git+https://gitlab.ifi.uzh.ch/dellaglio/ckan-stream/#egg=ckanext

# ELSE IF development
git clone https://gitlab.ifi.uzh.ch/dellaglio/ckan-stream.git
cd ckanext-webstreamsuzh
python setup.py develop
pip install -r dev-requirements.txt

# FINALLY restart CKAN
cd /usr/lib/ckan/default/src/ckan
paster serve /etc/ckan/default/production.ini

```

If the plugin is not registered, check if the `ckan.plugins` in the ini-file `/etc/ckan/default/product` contains the plugging name. E.g. `ckan.plugins = default, ..., webstreamsuzh`

A.1.3 Create User and API Key

Create an admin account within the CKAN virtualenv

```

# Enter the virtual environment
. /usr/lib/ckan/default/bin/activate

paster sysadmin add twave email=twave@localhost name=twave \
-c /etc/ckan/default/production.ini

```

The API key is required to push data to the catalog. Since the login is hidden we must go to `<ckan-host.com>/user/login` to get to the login page. Login with the credentials specified at the user creation above. The API key is shown in the profile page in the bottom left. At this stage we can also create an organization on CKAN, because each dataset must belong to an organization on CKAN.

A.1.4 Solr

We follow the following guide to install and configure Solr. The Solr configuration files are also on the CD in `solr-6.5.0`.

<https://github.com/ckan/ckan/wiki/Install-and-use-Solr-6.5-with-CKAN>.

Since Solr keeps changing the schema for the search, it's important to take a Solr version that is compatible with CKAN's schema (e.g. 6.5.0).

Solr-6.5.0 can be downloaded from <https://lucene.apache.org/solr/>.

Then run the following instructions:

```
tar xzf solr-6.5.0.tgz solr-6.5.0
cd solr-6.5.0/bin

# Create a core for CKAN
./solr create -c ckan
```

We then need to modify the schema for CKAN. The modified files are also on the CD at `solr-6.5.0/ckan/conf`.

```
cd solr-6.5.0/server/solr/ckan/conf
vim solrconfig.xml
```

Insert into `<config>` root element following line

```
<schemaFactory class="ClassicIndexSchemaFactory"/>
```

delete the following two elements

```
<initParams path="/update/**">
  <lst name="defaults">
    <str name="update.chain">add-unknown-fields-to-the-schema</str>
  </lst>
</initParams>

<processor class="solr.AddSchemaFieldsUpdateProcessorFactory">
  <str name="defaultFieldType">strings</str>
  <lst name="typeMapping">
    <str name="valueClass">java.lang.Boolean</str>
    <str name="fieldType">booleans</str>
  </lst>
  <lst name="typeMapping">
    <str name="valueClass">java.util.Date</str>
    <str name="fieldType">tdates</str>
  </lst>
  <lst name="typeMapping">
    <str name="valueClass">java.lang.Long</str>
    <str name="valueClass">java.lang.Integer</str>
    <str name="fieldType">tlongs</str>
  </lst>
  <lst name="typeMapping">
    <str name="valueClass">java.lang.Number</str>
    <str name="fieldType">tdoubles</str>
  </lst>
</processor>
```

```
then remove the file managed-schema
rm managed-schema
```

```
link the CKAN schema.xml from the CKAN source folder, e.g.
ln -s /ckan/default/src/ckan/ckan/config/solr/schema.xml schema.xml
```

```
Then (re)start solr by
solr-6.5.0/bin/init.d/solr restart
```

In CKAN's configuration `production.ini` set the Solr URL to `localhost:8983/solr/ckan`.

A.2 Kafka

We follow the Apache Kafka quickstart guide on <https://kafka.apache.org/quickstart> to set up the Kafka broker.

If the environment has low amount of RAM available, you can decrease the JVM heap size for the Zookeeper and Kafka. Both read the environment variable `KAFKA_HEAP_OPTS`.

```
tar -xzf kafka_2.12-2.2.0.tgz
cd kafka_2.12-2.2.0

export KAFKA_HEAP_OPTS="-Xmx128M -Xms64M"

nohup bin/zookeeper-server-start.sh config/zookeeper.properties > zklog &
nohup bin/kafka-server-start.sh config/server.properties > kafkalog &
```

We further did changes to the `server.properties` in order to save disk space on the VM.

```
log.retention.minutes=15
enable.auto.commit=false
offsets.retention.minutes=1

log.segment.bytes=1048576
log.cleaner.dedupe.buffer.size=31457280
```

A.3 Creating the web streams

For streaming and connecting the output to the brokers we require the following components:

- triplewave

- triplewave-jrml
- mqtt broker
- kafka connectors

We use the Node.js module `pm2` to manage all of our Node.js processes. A process can be started with `pm2 start <name>`.

A.3.1 TripleWave and JRML

First clone the `triplewave` and the `triplewave-jrml` repository. Then

1. go to the `triplewave` folder and `npm install` and `npm link`
2. go to the `triplewave-jrml` folder and `npm install` and `npm link`
3. go back to the `triplewave` folder and `npm link jrml` to make `jrml` available to `triplewave`

Then go to the `TripleWaveConfig` folder and run `triplewave` from there `triplewave` or `pm2 start triplewave`.

`Triplewave` uses by default the `tw_config.js` to get its configuration. Any other configuration can be specified by running `triplewave` with `-c <path_to_config>`

A.3.2 MQTT Broker

We use the Node.js module `aedes` as our MQTT broker.

Go to the `mqtt` folder and run `npm install`. Then simply run `node aedes.js` or `pm2 start aedes.js`.

A.3.3 Kafka Connectors

The Kafka connectors are located in the `kafka` folder. Again run `npm install` and run each `.js` file within that folder directly with `node` or `pm2`.

List of Figures

4.1	Overview of all components	20
4.2	Overview of TripleWave (source: Bernhaut (2018))	21
4.3	Overview of the message brokers	22
5.1	The stream descriptor for a stream of temperature sensor data	27
5.2	An RDFStream containing two different endpoints	27
5.3	A StreamEndpoint	28
5.4	TripleWave registering the stream	29
6.1	CKAN architecture (source: https://docs.ckan.org/en/2.8/contributing/architecture.html)	38
6.2	Plugin components	39
6.3	The stream description as shown on the catalog	42
6.4	Preview of the stream on the catalog	44
6.5	The catalog homepage serving as an entrypoint for discovering web streams	46

List of Tables

5.1	Catalog settings	32
6.1	RDF stream description to CKAN mapping	40
7.1	Zurich fiber optic network performance data	51
7.2	Rail traffic information	51
7.3	Vienna GTFS	52
7.4	Air quality and meteorological data in Upper Austria	52
7.5	Public bike locations in the city of Bonn	53
7.6	Locations and availability of charging stations	53
7.7	GFTS real-time Rhein-Sieg Transport Network	53
7.8	Selected datasets: some mappings from ¹ Muntwyler (2017) and ² Bernhaut (2018)	54
7.9	Links to the original data source	55
7.10	The mapping of the RTI dataset: prefix <i>srtri</i> : http://streamreasoning.org/rtri/	56
7.11	The mapping of the AQZH dataset: prefix <i>sraqzh</i> : http://streamreasoning.org/aqzh/ prefix <i>air</i> : http://qweb.cs.aau.dk/airbase/ , prefix <i>wgs84_pos</i> : http://www.w3.org/2003/01/geo/wgs84_pos#	57
7.12	The mapping of the WSZH dataset: prefix <i>srwszh</i> : http://streamreasoning.org/srwszh/	58
7.13	The mapping of the charging stations dataset: prefix <i>slacs</i> : http://streamreasoning.org/lacs/ , prefix <i>schema</i> : http://schema.org/	58