FACULTY OF ENGINEERING AND ARCHITECTURE

Adapting an open-source API registry to support publication of IoT sensor data

Robbe De Vilder
Student number: 01507157

Supervisors: Dr. Pieter Colpaert, Prof. dr. ir. Ruben Verborgh
Counsellors: Julian Andres Rojas Melendez, Brecht Van de Vyvere

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

GHENT UNIVERSITY

This work would not have been possible without

Brecht Van de Vyvere which as counselor provided invaluable insights and guidance during regular meetings,
Dr. Pieter Colpaert which as supervisor helped shaping the idea and provided evaluation of the progress,
Jef Willems for the development of the original version of ApiApi and help with its development set up,

Adapting an open-source API registry to support publication of IoT sensor data

Robbe De Vilder

Supervisor(s): Brecht Van de Vyvere, Dr. Pieter Colpaert, Prof. Dr. Ir. Ruben Verborgh

Abstract **With the rise of IoT sensor data being published over the Internet comes the diversity in data models used to do so. This diversity in models limits the reusability and interportability of the published data. To address this problem the ApiApi framework is being developed. This tool aggregates endpoints by combining and mapping basic JSON data sources and publishing the result in the form of a homogeneous data dump. A graphical user interface is included which provides the user with an accessible solution for managing different data collection. While a step in the right direction ApiApi still misses some key features such as linked data support and a fragmentation strategy. This work proposes an adaptation to this framework to address those issues. RML and YARRRML are proposed as solution for creating mappings to semantic models with additional SHACL support for shape validation. The framework proposes a dynamically generated form which produces simple RML mappings from users input and a SHACL defined data model. Sampling is used to compute deltas in the source data and publish these updates as part of an event stream. Each event stream can be fragmented further using a spatial query provided using the slippy tile notation. Since each event stream fragment is theoretically immutable, they can be cached for long periods of time.**

Keywords **Event Stream, RML, YARRRML, SHACL, linked data fragments**

I. Introduction

With the rise of Internet connected devices and the resulting Internet of things data gets published ever rising quantities [1]. From transportation services to smart trash cans more and more devices get connect to the world wide web and contribute their measurements. This increase in data holds a treasure of information, but is often limited in usefulness since it is published by many different actors with heterogeneous data models which limits the reusability of the data. To address this ApiApi [2] is being develop. This framework provides an easy to use method to quickly deploy an aggregated endpoint which publishes data from different sources as one uniform data dump. All data is mapped to a single data model before publishing. While this is a step in the right direction the tool still misses key aspects such as linked data support and a fragmentation strategy. The proposed framework will adapt the existing ApiApi implementation and address these limitations. Mapping to semantic models will be made possible by using RML [3] and YARRRML [4] while an event stream will be used to create a cacheable and fragmented endpoint. Additionally SHACL [5] will be used to to load existing data models in the system and to validate the mappings. This paper will first give an introduction of the work flow of the updated system after which mapping and event stream generation will be explored in more detail.

II. Related work

ApiApi [2] is an endpoint aggregation framework develop by Digipolis. ApiApi combines and maps data from JSON endpoints and creates a single homogeneous data dump. ApiApi has a graphical user interface which can be used for each step in the process. A visual mapper tool allows the user to create simple JSON to JSON mappings which transform the data from all endpoint into a single data model.

RDF Mapping Language (**RML**) [3] is a semantic data mapping language designed for creating mappings between one of several non semantic data sources and RDF. These mappings are themselves expressed as RDF graphs.

YARRRML [4] is a semantic data mapping language which is serialized as YAML. Since YAML is designed to be human readable, YARRRML is an easy to use and transparent mapping language. As the name suggests YARRRML is related to RML since it is designed by the same working group as a way of increasing the ease of creating mappings.

Shapes Constraint Language (SHACL) [5] is a language for validating RDF graphs. A SHACL file exists of any number of SHACL shapes, which describes constraints on an RDF node. Each SHACL shape contains either an RDF id or type and each RDF node with the id or type will get validated against the shape. SHACL supports a wide range of constraints such as counts, types, lengths and so on.

Linked data fragments are the grouped name for methods of fragmenting linked data. Several fragmenting methods exists with the most trivial being a data dump and SPARQL endpoint. However, each method implies certain trade-offs, such as high server load or high duplication. Linked Data Fragment allows to compare fragmentation strategies and encourages the search for intermediary fragmentation strategies. Such solutions are for example Triple Pattern Fragments [6] or event streams.

Event Streams are a type of linked data fragment. Each fragment holds a number of updated data points. Fragments are linked to each other using the tree ontology. Event stream fragments are ordered by age, so the newest fragment contains the most recent updates and so on. Since event streams publish updates about the data instead of data dumps there is no endless replication of the same unchanged data. Additionally since older fragments are not edited when changes occur fragments can be cached for a long period of time.

R. De Vilder is with the Department of Information Technology, Ghent University (UGent), Gent, Belgium. E-mail: Robbe.DeVilder@UGent.be .

III. Implementation

A. An introduction to the framework

The proposed framework is a solution for combining, mapping and republishing data. All of this can be configured using a graphical user interface to make it accessible and easy to use. The framework uses a few key concepts. A Collection is the top level component, which is an combination of any amount of APIs and a single Model. A Collection has a single endpoint which combines the data from all APIs in it. Each API should individually comply with the constraints as specified by the Model. An API can contain any number of endpoints, and should contain each endpoint which is required to create a valid mapping. To illustrate this take for example a Model which contains a SHACL shape which requires a location and measurement to be present. If this information is distrusted in two different endpoints, none of these endpoints alone would contain enough information to create a valid mapping. For this reason both of them are added to the same API.

B. Configuring a Collection

The user starts with creating a Model. This Model can hold a single SHACL document containing any number often SHACL shapes. Then, the user can create a Collection which contains this Model. The Collection is initially empty and can be populated by creating any number of APIS. Creating an API is done by first defining all endpoints. For each endpoint meta information can be provided. The amount of meta information provided will define the functionality of the endpoint. Three different functional tiers are available with the latter always including the functionality of the former. The three tiers are a mapped data dump, event stream and spatial fragmented event stream. After this a mapping can be created using one of three options. Either an RML or YARRRML definition can be provided as text, or the RML by form option can be used to create an RML mapping by providing reference paths. The form is generated based of the SHACL shape selected and currently only supports simple shapes. These are shapes which are not nested and have a dept of 1. This component is further detailed in subsection D. This process can be repeated to populate the Collection with APIS. Finally, the user should start the sampling process in the Collections overview. From this point on an event stream will be available on the endpoint

C. Using the endpoint

When creating a Collection its endpoints are made available. The number of available endpoints depend on the information provided when creating the APIs. The endpoint uses the standard REST principle where a HTTP GET request is sent to retrieve its information. Each endpoint uses the same base URL template being {baseURL}\api\{CollectionID}. A first functionality is receiving a data dump of all mapped data of all endpoints. This can be done by sending a GET request to the base endpoint. An event stream of all data in the Collection is available on the endpoint \stream. This will return the first fragment of the event stream which contains links to the other fragments. Spatial fragmentation can be achieved by adding a suffix to the stream endpoint with the slippy tile notation of the location. Slippy tiles use three parameters being zoom, x and y. The surface of the earth is devided in a number of tiles based on the zoom level and the tile desired is defined using the x and y values. An event stream which holds only entries of a tile can be retrieved using \stream\{zoom}\{x}\{y}. Similar to the base event stream, this returns the most recent fragment. This fragment contains links to both its temporal and spatial neighbours. Additionally the endpoints supports a method to query a fragment by time. This is done by sending a request to \stream\{unixtime} or \stream\{zoom}\{x}\{y}\{unixtime} respectively. The unix time has to be provided in milliseconds. If multiple Records are generated at the same time which makes them span multiple pages, the first page is retrieved.

D. Creating an RML mapping using the SHACL generated form

Besides inputting RML or YARRRML as text, the framework supports the creation of simple RML mappings using a form. This eases the mapping since the RML is mainly abstracted behind the form and the user only has to provide the paths to the properties in his JSON data. Unlike similar solutions [9] the user only has to provide the subject of the mapping, since the objects are provided by the SHACL shape in the Model. Before mapping the user can select one shape of the SHACL file after which input fields are generated. For each field the user can select to input a path or a constant. In the former, the user provides the path in JSON endpoint which points to the property. In the latter, a constant value is passed on which will be the same for each data point. This is implemented as Vue.js component which is an adaptation of a SHACL form component [7].

E. Sampling and creating the event stream endpoint

In order to keep track of changes in the data which are required to publish an event stream, the framework samples the endpoints every sample interval. The duration of such interval is configurable by the user. For each interval requests are sent to all the endpoints and the data is compared with the previous run by comparing its hash. Comparison is done per data point. All changed points are kept and saved, unchanged data points are not saved again. Changed data points are saved as Records. These Records are then used to generate the event stream. For each request a data base query is constructed which fetches the correct Records. Since event stream fragments are constant, they are cached after first creation. This results in cache hits and nearly instant retrieval of fragments from the second request on. This is true for all fragments besides the most recent, since this fragment can still change by appending new updates. Variations of the default stream endpoint request such as a slippy tile stream or time query are implemented as simple adaptations of the same data base query with additional constraints. These fragments can also be efficiently cached at the users side for the same reason mentioned above.

IV. Conclusion

This work proposes a framework which transforms any number of simple JSON endpoints into an event stream. During each step linked data technologies are used.

First, RML or YARRRML can be used to map JSON data to RDF. Next, SHACL is used to validate the RDF output with an RDF vocabulary. Last, the Event Stream fragmentation method is used to publish the data. The strength of the proposed framework lies in the ease to which these technologies can be applied to the source data. The proposed

framework is a single tool which handles all the steps from source data insertion to publication. Furthermore all steps are done using a graphical user interface and are documented in application. The design of a custom RML via form method further increase the accessibility. The modularity of the system allows its application in many use cases, from simple mapping to the creation of spatial fragmented event streams. In future work the framework could be expanded or improved in several ways. The implementation of other input data formats such as XML or CSV would widen its use cases further. In addition, the form used to generate RML rules only supports simple shapes. Therefore, adding support for nested objects would broaden its potential usage.

REFERENCES

[1] McKinsey & Company, *Growing opportunities in the Internet of Things*,https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things

[2] Digipolis, *ApiApi*, https://github.com/lab9k/apiapi

[3] Anastasia Dimou and Miel Vander Sande, *RDF Mapping Language (RML)*, https://rml.io/specs/rml/

[4] Ben De Meester and Pieter Heybaert and Anastasia Dimou,*YARRRML*, https://rml.io/yarrrml/spec/

[5] Holger Knublauch and Dimitris Kontostas, *Shapes Constraint Language (SHACL),* https://www.w3.org/TR/shacl/

[6] Ruben Verborgh and Miel Vander Sande and Olaf Hartig and Joachim Van Herwegen and Laurens De Vocht and Ben De Meester and Geral Haesendonck and Pieter Colpaert, *Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web*, https://linkeddatafragments.org/publications/jws2016.pdf

[7] Babibubebon, *vue-shacl-form* ,https://github.com/Babibubebon/vue-shacl-form

[8] *RML Editor*,https://rml.io/tools/rmleditor/

Inhoudsopgave

Lijst van figuren

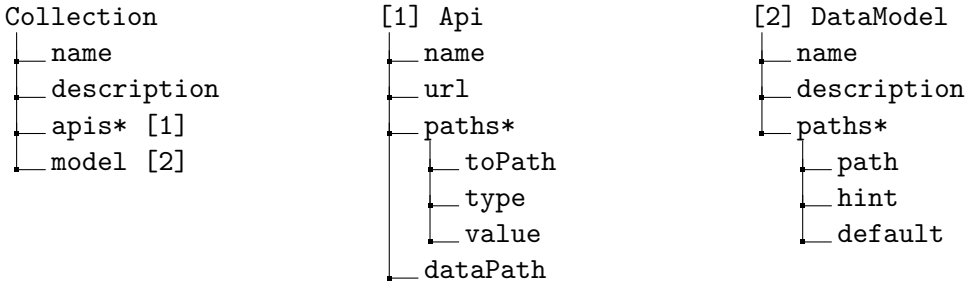
Lijst van tabellen

Lijst van listings

1

Introduction

With the ever expanding number of Internet connected devices comes the ever increasing amount of data being published. More and more common devices get revised to include sensors which produce information about its state. Besides a vast difference in devices on this Internet of Things, a vast difference in standards used to publish data exists. While efforts have been and are being made to increase standardization the reality is still that a large amount of devices produce data in their own proprietary formats. While the information kept within this data is often the same, the difference in this standard makes it hard to use in an interoperable fashion. To address this issue the ApiApi framework is proposed. ApiApi combines and maps several endpoints with different standards and creates a single endpoint with a single standard. While ApiApi is already a step in the right direction the lack of linked data support and the simplicity of the endpoint it provides still prevents it from being a truely Internet of Things ready framework. This work will propose and implement several changes to the ApiApi framework which increases the quality of the data and the endpoint with the Internet of Things in mind. These changes will transform the endpoint from a simple JSON data dump to a publisher of linked data fragments. Besides the endpoint the mapping interface will be reworked to allow mappings to semantic models. This introduction will first provide an overview of the current state of ApiApi after which a short overview of the meaning of linked data will be given. Finally a set of user stories will be proposed which encapsulate the goals of this work.



Figuur 1.1: ApiApi MongoDB models. Models have been simplified for illustration. An * represents an Array.

1.1 ApiApi Current

ApiApi [1] is a Api aggregation framework develop by Digipolis[1] for the city of Ghent. The goal of ApiApi is twofold. First it combines data from different data sources into a single endpoint. Secondly it transforms this data to fit a predefined data model. This way, the data is uniform is place (a single endpoint) as uniform in form (a single data model). ApiApi presents this functionality in a user friendly Vue.js graphical user interface.

This section will cover the current state of the ApiApi in more detail. The section starts with an introduction and general overview of ApiApi, followed by a specific look at the two main components: the Api Mapper and the model creator.

1.1.1 Overview

The logical ApiApi structure is built upon three concepts: a Collection, an Api and a Model. A **Model** represents the data model the data coming from an Api should be mapped to. Models are created in the Model Creator. An **Api** represents a single endpoint and its meta data which allows it to be mapped to a Model. A **Collection** represents a set of Apis with a common Model. When adding an Api to a Collection, a mapping should be created which transforms the data coming from the endpoint into a form compliant with the Collections Model. Each Collection has a single endpoint which combines the data from all the Apis each mapped to the Model. As a whole, ApiApi thus creates a single endpoints which holds data from all the where Apis each data point is presented as an instance of the Collections Model.

The ApiApi landing screen is an overview of the Collections and Models currently in the database. Collections and Models can be deleted by pressing the *Delete* button on the corresponding card. To create a new Collection or Model, the + or *+ Add New Collection* (Model) button

[1] https://www.digipolis.be/

should be pressed. This then redirects to either the Collection creation or Model creation page. Collection creation is straightforward and only requires a name and a Model, optionally a description can be given. The Model can be selected from one of the current Models present in the database displayed on the landing screen. The Model creation process is detailed in section 1.1.3. Back on the landing page the *Details* button on each Model or Collection card will redirect to a page with more information about the entity. Model details are not implemented in this version. Collection Details will redirect to the Collection overview page, which displays the Api's currently in the collection and the Model linked to the Collection. From here Apis can be deleted from the Collection by pressing the *Delete* button. This removes also removes the Api entity from the database. An Api can be added to the current Collection by pressing the + or + *Add New Api* button. Adding a new Api will redirect to the Api Mapper where a new Api entry can be created together with a mapping to the Model. This is covered in section 1.1.2. Finally the header contains a button which redirects to the server settings. Currently only a flush cache setting is present which will delete the local Redis cache.

With Collection creation ApiApi enables a single REST endpoint which is baseURL\api\CollectionId. Sending a GET request to this endpoint will retrieve a data dump of all mapped data from each Api in the Collection. Depending on the state of the cache the endpoints are either contacted to retrieve their data or the cache data is used. All data is mapped not matter the state of the cache since only source data is stored.

As for the technical implementation ApiApi uses the Nuxt.js Vue.js framework. For storage mongoDB is used for persistent data and a Redis store for the endpoint data cache. MongoDB is interacted with using Mongoose which is a Node framework which handles the database queries. For quick styling and component construction the Vuetify UI Library is used. All programming is done in Javascript and compiled using Node.js. A short overview of these technologies is presented below.

Vue.js[2] is an open-source Node.js framework for dynamic web application development. For GUI development Vue.js uses a Component structure, where each Component can be logically seen as a reusable piece of GUI with the possibility to have its own logic. Components are then combined in higher level Components up to the page level. This principle makes Vue.js a well organized and scalable framework to work with.

The **Nuxt.js**[3] is a an open-source framework built upon Vue.js which eases development by organizing the project. It uses best practice development conventions to create file structure and makes customization easy by using a nuxt-config file. Besides this it increase Vue.js load speeds by implementing server side rending.

[2]https://vuejs.org/
[3]https://nuxtjs.org/

MongoDB[4] is a document-based database which uses JSON documents as data structure. It probably needs little introduction since MongoDB is the most popular of the document-based database systems and used in wide range of applications. MongoDB documents are stored with an unique *_id* property which can be either generated or supplied. Besides this, the user is free to create any JSON like object using complex JSON structures and arrays. MongoDB supports 15 data types including Strings, Numbers, Dates and Timestamps.

To interact with the Mongo Data Base ApiApi uses **Mongoose**[5]. Mongoose is a wrapper for Node.js that handles some database interactions such as modeling and queries. ApiApi has a model for each entity as described above: Collections, Apis and Models.

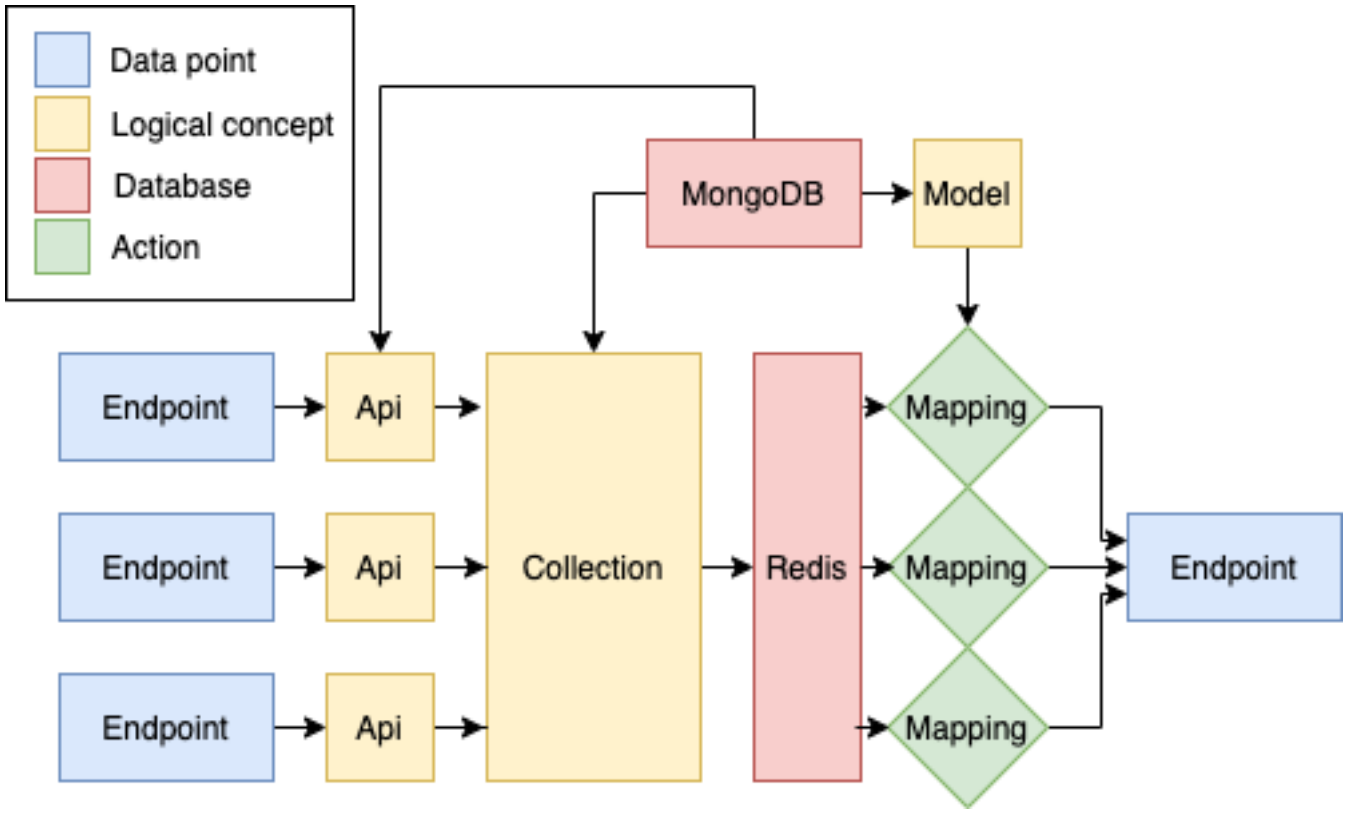
Finally **Redis**[6] is used as cache database for the endpoint data. Redis is an open source in memory store, which allows for fast retrieval of entries. ApiApi uses Redis to store the endpoints response the avoid having to query the endpoint every endpoint for every request. The Redis records can be set to expire after some time, after which the record will be deleted and the endpoint requeried upon the next request regardless. Setting this expire time requires balancing between the importance of up to date data (expire time set to 0 requeries for every request) and bandwidth efficiency. Note that Redis stores the original data, so mapping has to happen after every request.

1.1.2 The Api Mapper

Api creation is done using ApiApis mapper tool. ApiApi only supports mapping from JSON endpoints. These endpoints should be constructed in such a way that each data point correspond with a single JSON object in a JSON array. This array can be present as root object or under a property of the root object. Data points can be complex, in which case properties are accessed as *a.b*. Data points can contain arrays, in which case these properties are accessed as *a[i]*. Since ApiApi uses the first data entry of the endpoint as template, support for different objects is not present. For the same reason, support for different array sizes is not present. Each data point should have each relevant property of the first and the first should contain all the relevant properties for the mapping.

Accesing the mapper is done by creating a new Api in the Collection detail screen. By pressing the *+* or *+ Add new Api* button a redirect is created to the Api creation screen. The first three entries that should be provided are the name, URL and authentication method. As authentication method ApiApi supports either open endpoints, closed with api key or the use of custom headers. When selecting one of the two latter a menu drops down where either the api key or the custom

[4] https://www.mongodb.com/
[5] https://mongoosejs.com/
[6] https://redis.io/



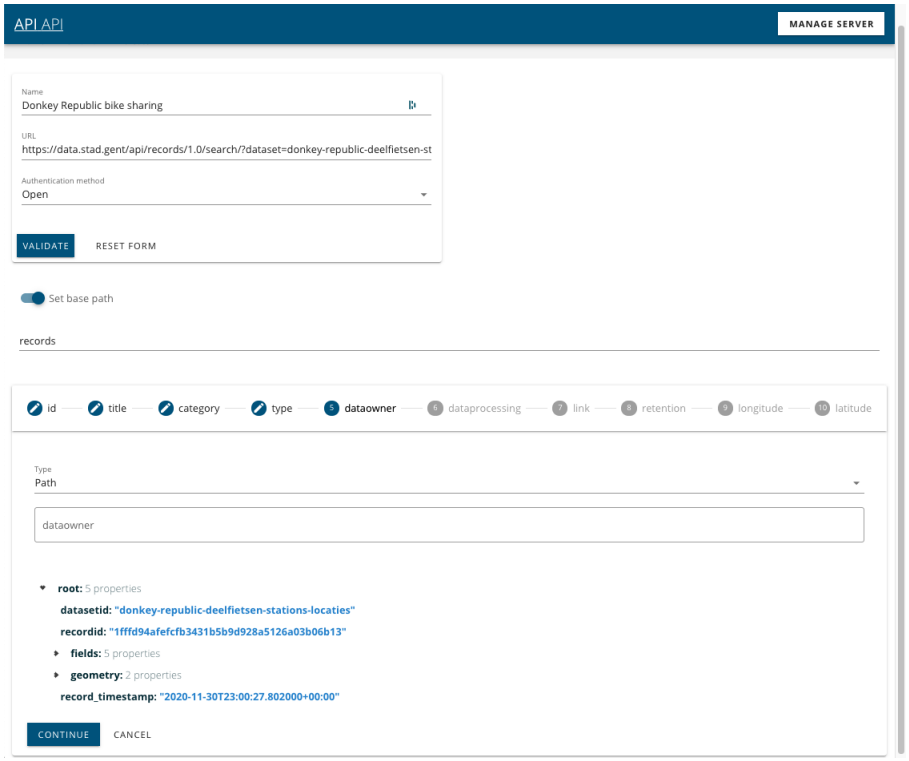
Figuur 1.2: The structure of ApiApi

headers can be inserted. There is support for any number of custom headers. This basic api information header is concluded by pressing validate, which contacts the endpoint on its given URL using given authentication method and tries to retrieve the data. On succes the data is locally stored and the user can continue. On failure continuation is blocked. There is no indication of failure except the not loading of the next component. Also note that success is defined freely as *the endpoint responded* but does not check the response. So any HTTP code, as long as the response is JSON parsable, passes. Including 404's and others.

After successful validation the mapper is loaded. This is the graphical tool used to create a relation between the endpoints data scheme and the data model. The mappers header contains the possibility to set a base path different from the root. This base path will be resolved and the resulting property will be set as the root for the mapping. This root should contain the data points as a JSON list. Using the mapper itself is straightforward because of the graphical interface. The header displays the different properties of the Model which should be mapped to. Pressing a property opens a drop down where the mapping information can be inserted. Two types of mappings are possible being constant or path. Constant mappings require the constant value which will be identical for each data point. Path mappings create a relation between the the original data and the mapped version. When selecting path another drop down appears with the first JSON data point of the given endpoint displayed. The user is then able to click a property which fills in the path with the JSON path of the clicked property. Either pressing continue or using the header advances the mapping until all of the Models properties are accounted for. When reaching the last property the continue button changes into complete which will allow the confirm button to be pressed which finalizes the Api creation. Confirming the Api creation results in a pop up being shown with the entered mapping. The pop up does not give context so its use is limited for revision. The pop up can be either submitted or rejected. The former creating an Api MongoDB entry and redirecting to the Collection detail page. The latter returns to the Api creation page where the information can be adjusted. Api submission can fail if not all of the Models properties are mapped. This can be done by simply continuing without inserting a value in the mapper. ApiApi will attempt creation of the Api but since each mapping property is set to *required* in the mongoDB model a validation error will be thrown and the submission will be rejected. There is no indication of this, and a failed mapping can only be concluded after observing that the added Api is not present under the Collection Apis. Such failed submission requires a restart of the Api creation.

1.1.3 The Model creator

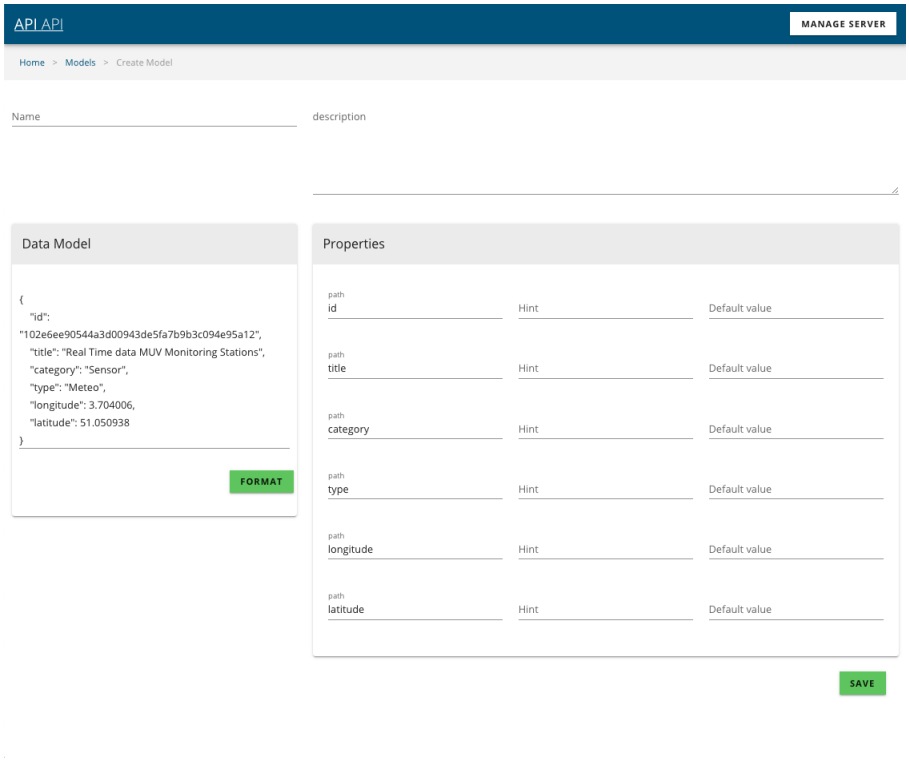
The Model creator is responsible for constructing a data model where Apis should be mapped to. The user is able to do this by filling out a form. At the top the required model name and optional model description can be provided. In the *Data Model* text area the model is created



Figuur 1.3: The Api creation view

by example. This is done by pasting a single model data point into the text box. Since the implementation is based on extracting properties only a JSON example is possible. The JSON properties are thus extracted and displayed in the form area labeled *Properties*. Complex JSON objects can be used and are displayed as *a.b* and *b[0]*. This form can be used to annotate the properties further with a Hint (used to give additional information about the property) and a Default value. Type checking or additional restrictions are not supported.

When saving the Model, a single DataModel database entry is constructed with the information supplied. This construction is straightforward since the DataModel model only has 4 properties which are all supplied. Note that while type checking or additional restrictions are not supported there is a single implicit restriction in that every property of the given DataModel should exactly hold 1 entry when creating a mapping. This is due to data base validation. The MongoDB model is constructed as such so that every property is marked as a required field. This means that MongoDB will not accept entries with empty fields and the mapping will be rejected.



Figuur 1.4: The model creation view

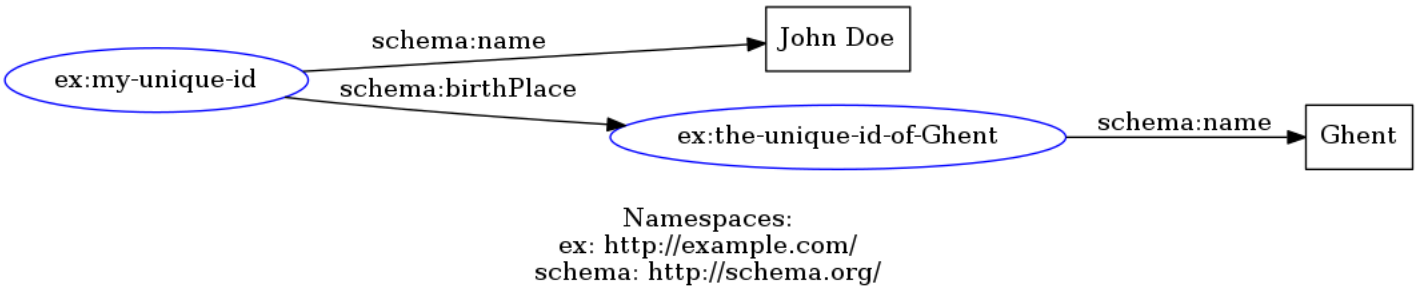
1.2 Linked Data

Linked Data is a method for publishing structured data on the Internet. Linked refers to the fact that that the data is interlinked by the use of common vocabularies. Instead using custom terms linked data uses these vocabularies to describe entities. The common use of these terms thus allow for transparent interpretation of linked data from different data sources most importantly by machines which lack the ability to interpret as humans do. In linked data each term which is not a value is expressed as a URI, for example the name of a person could be described using the schema vocabulary[7] as http://schema.org/name.

Because of these advantages publishing by using Linked Data is being pushed by organizations and governments. An example of this is the Open Standaarden voor Linkende Organisaties [2] (Open Standards for Linked Organizations, **OSLO**) initiative by the Flemish government which encourages and provides tools for organizations to publish its data as Linked Data. The OSLO initiative includes vocabularies and application profiles which define terms and describe entities respectively.

RDF [3] [4] is a standard for describing such interchangeable data developed by the w3c foun-

[7]https://www.schema.org/



Figuur 1.5: A visual example of the RDF structure

dation. RDF uses the concept of subjects, predicates and objects. A combination of a subject, predicate and object is called a triple. Triples can be used to describe relations between entities, or describe the values of entities. This creates a graph which interlinks the different entities trough these triple relations. An example of RDF triples in the case of the author could be his name relation described as <my-unique-id> <http://schema.org/name> ”John Doe”, with an unique id URL which describes my entity as person uniquely, such as generated my nationality combined with social security number. In this example my id is the subject, name is the predicate and the actual value is the object. RDF objects can also be URIs which refer to other entities, for example <my-unique-id> <https://schema.org/birthPlace> <the-unique-id-of-Ghent>. <the-unique-id-of-Ghent> can then also engage in a triple such as <the-unique-id-of-Ghent> <https://schema.org/name> ”Ghent”as a subject. RDF can be serialized in many different forms. The serialization used in the example are **N-Quads** which serialize the subject predicate object directly. Other serialization forms such as **Turtle** or **JSON LD** hold this same information in a different structure.

1.3 User Stories

This section will provide an exploration into the goals of this work. With both an overview of the current state of the ApiApi framework and linked data concepts, the question remains: how to combine these? How to use the opportunities linked data provides with in the current project. This section will construct several user stories which incorporate different aspects of the frameworks use cycle. This work will then explore existing linked data scoped technologies to implement these in a state of the art fashion.

As a preface to the user stories, some context will be given about the actors. First of all there is the *Consumer*. This actor consumes data from the ApiApi endpoint. He interacts with the endpoint and receives a response. Secondly there is the *Administrator*. This actor is the owner of the ApiApi and has access to its interface. He creates entities such as collections and models.

He adds apis to these collections and is responsible for the mapping of those.

When talking about the inclusion of linked data into ApiApi, a first obvious user story is the support of such. ApiApi would hardly incorporate linked data best practices if it does not produce linked data itself. For this, the user story *As a consumer, i can receive a semantically annotated response* is created. To facilitate the consumption of semantically annotated data, there should be a possibility for creation. As described the the exploration of the current state of ApiApi, there is currently no support for mapping to linked data. Thus addition of such a feature is required. This is covered in user story *as an administrator, i can map an endpoint to a semantic data model*. To facilitate a mapping more easily, a method of feedback should be provided to the user. This feedback should give insight in the compliance of his mapping to the data model at hand. For this *As an administrator, i can receive feedback about the compliance of my mapping according to the restrictions imposed on the data model* is created. For mapping form an endpoint to a semantic data model, the possibility to create such model should exist. ApiApi already has this possibility, but this work will implement the support for semantic models. In addition to the support it would be beneficial to be able to re use existing data models such as the OSLO application profiles. For this reason *As an administrator, i load existing data models into the platform* is created.

With the ability to create and consume semantically rich data a minimal working concept for the framework is established. There are however several additional optimizations that could be made to the framework to incorporate best practices or enrich the data further. A first is the ability to receive fragmented responses. Optimal fragments are cacheabale by the server and client to reduce recomputing and requerying. Additionally ideally the size of the fragments should be controllable such that size optimizations can be made. For these reasons two user stories are created *As a user, i can receive a cacheble response*, *As an administrator, i can publish linked data as cacheable fragments* and *As an administrator, i can control the size of a fragment*. Fragmentation rises the question which dimension to fragment on. Since the system is mainly designed for IoT sensors two dimensions are most relevant being place and time. Place for allowing the retrieval of data from a certain relevant location for example a single city. Time for allowing the retrieval of data from a certain time frame which gives access to sensor readings from the past and their evolution over time. These two dimensions are added as requirements for the user story as *Temporal and geo spatial fragmentation should be supported*. Extracting historic information from simple JSON endpoints is not straight forward since most of these are simple live data dumps with only the current sensor readings. The current framework simply forwarded this data to its own endpoint without any sense of history. For this reason an additional user story is created which states *As a consumer, i can receive historical information about the data.*

All of these user stories can roughly be translated into a single research question which is *How to transform ApiApi into a linked data system which publishes cacheable fragments of a spatial and temporal nature?*.

Tabel 1.1: User stories

Number	User Story	Requirements
1.1.1	As a consumer, i can receive a cacheable response	
1.1.2	As a consumer, i can receive a semantically annotated response	
1.1.3	As a consumer, i can receive historical information about the data	1.2.6
1.1.4	As an administrator, i load existing data models into the platform	
1.1.5	As an administrator, i can provide restrictions for those data models	1.2.8
1.1.6	As an administrator, i can map an endpoint to a semantic data model	1.2.1,1.2.2,1.2.3
1.1.7	As an administrator, i can receive feedback about the compliance of my mapping according to the restrictions imposed on the data model	1.2.9
1.1.8	As an administrator, i can control the size of a fragment	1.2.4
1.1.9	As an administrator, i can publish linked data as cacheable fragments	1.2.5 1.2.7, 1.2.10, 1.2.11

Tabel 1.2: Requirements

Number	Requirement
1.2.1	As input, the mapper should at least support JSON endpoints
1.2.2	As output, the mapper must support JSON-LD
1.2.3	The mapper should have a GUI, which allows for a mapping from and to simple data models
1.2.4	The number of data points in a response should be customizable
1.2.5	Temporal and geo spatial fragmentation should be supported.
1.2.6	Historical information should not be live, but arbitrary close to live data.
1.2.7	Fragments should hold contain all the information for traversal to other fragment. All fragments should be reachable from any of those fragments in a limited number of steps.
1.2.8	At least SHACL should be supported to impose restrictions.
1.2.9	Feedback should be visible before submitting the mapping so that the user is able to make adjustments if necessary
1.2.10	Fragments should be at least be available as JSONLD
1.2.11	Fragments should be cacheable for longer periods of time

2

Related Work

2.1 Mapping Languages

2.1.1 R2RML and variants

R2RML [5] (RDB to RDF Mapping Language) is a language designed for creating mapping between relational databases and RDF. These mappings are them selfs expressed as RDF graphs. After the release of this specification, several extensions have been proposed that adapt the R2RML specification to serve different needs. A comparison of these extension is provided in this paper by De Meester e.a. [6]. Since this thesis uses JSON endpoints as data source, only extensions that allow this are useful for this cause. The comparative work [6] lists RML and YARRRML as possible R2RML extensions with JSON support.

2.1.2 RML

RML (RDF Mapping Language)[7] [8] is an R2RML extension that adds support for more input data sources. The RML specification lists CSV, TSV, XML and JSON as supported input formats. Besides this, RML follows a similar syntax as R2RML.

This work will limit itself to a short summary of the RML language and an example. For a full description of the RML language the specification can be consulted [7]. RML is written in Turtle and a single RML file can exists out of any number of **TripleMaps**. A TripleMap corresponds to a node in RDF. Similar to how each node should have an id, each TripleMap should contain a **SubjectMap**. This map defines the how to subject of the node is constructed. The subject can also be set to blankNode. A TripleMap may contain any number of **predicateObjectMaps**. These maps describe the construction of the properties of the node. Each predicateObjectMap will together with the SubjectMap generate a single RDF triple. This principle is illustrated in figure 2.1. A predicateObjectMap contains a **predicate** and **objectMap**. An objectMap is either a constant which will be the same for each data point or a reference to the source file. Besides a subjectMap and any number of predicateObjectMaps a TriplesMap should also contain a **logicalSource**. This object contains all the information about the source of the data. In the case of a JSON source an iterator which will be used to iterate over the data points. The RML mapper[1][2] supports multiple output formats. In this example and in ApiApi the format is set to JSON-LD. For more complicated mappings RML supports the creation of links between TripleMaps. Instead of either a reference or constant the objectMap can contain an **parentTriplesMap** which has another TriplesMap as value. This is illustrated by figure

2.1.3 YARRRML

YARRRML [9] is a specification describing mapping rules expressed as YAML. Unlike RML, YARRRML is designed to be easily human readable hence the choice for the YAML serialization. YARRRML in itself is no extension of R2RML, but was created out of the desire to make RML rules more accessible. Resolving YARRRML rules is currently only possible by first transforming the ruleset[3] to RML and using the RML mapper.[1,2]. Due to the similarity in functionality and structure between the two mapping languages this work will not cover YARRRML extensively. The main difference beside the serialization are the naming conventions. A **subject** has a similar function to a subjectMap, **po** to predicateObjectMap, **object** to objectMap and **source** to logicalSource. The same example as given in the RML section is provided here as YARRRML in figure 2.2. Note the similarities between the two languages and how the YARRRML mapping is considerable shorter and easier to read, which is exactly its design goal.

[1] Java: https://github.com/RMLio/rmlmapper-java
[2] Javascript wrapper: https://github.com/RMLio/rmlmapper-java-wrapper-js
[3] https://github.com/rmlio/yarrrml-parser

```
[
  {
    "id": 45643,
    "name": "John Doe"
  },
  {
    "id": 45534,
    "name": "Jane Doe"
  },
  ...
]
```

```
[
  {
    "@id":
        "https://short.url/45643",
    "schema:name": "John Doe"
  },
  {
    "@id":
        "https://short.url/45534",
    "schema:name": "Jane Doe"
  },
  ...
]
```

```
rr:subjectMap [
  rr:template "https://short.url/{id}"
];
rr:predicateObjectMap [
  rr:predicate schema:name;
  rr:objectMap [
    rml:reference "name"
  ]
].
```

Figuur 2.1: Illustrative example of a RML mapping, header is omitted

```
    subject: https://short.url/$(recordid)
    po:
      - predicate: shema:name
        object: $(name)
```

Figuur 2.2: Illustrative example of a YARRRML mapping, header is omitted

2.2 SHACL

SHACL[4] (SHApes Constraint Language) is a language created for validating RDF graphs. SHACL rules are serialized as Turtle, and in themselves are an RDF graph. SHACL files can hold several graphs, named shapes graphs, which can be used to validate data graphs over a set of conditions. Data graphs and shape graphs can be linked by either id or class, meaning each graph with a certain id or class will be validated against that shapes graph. SHACL shapes can hold a wide range of constraints such as counts, types, lengths and so on. For this reason SHACL can also be used to describe data models such as is done in the OSLO project [2].

A full specification of the SHACL language can be found on the W3C website4. An interesting sideshow to get quick started can be found on Slideshare[5].

2.2.1 SHACL form generation

Since SHACL describes a shape where data should be in, a logical follow up question is how to generate data that is in such shape. One of such ways is creating a GUI for users to input information. Currently there is an unofficial draft for an extension of the SHACL vocabulary with the DASH name space which adds information to the SHACL shape for GUI generation information[6]. Independently several libraries have been created which generates a GUI form from a SHACL shape[7][8]. The *Babibubebon/vue-shacl-form* provides the form as a Vue.js component, which fits perfectly in the current ApiApi technology stack.

The *Babibubebon/vue-shacl-form* provides a Vue.js components which generates a form based of a SHACL shape, and a data point from the user input in this form. The principle is simple, the user can select one shape from a SHACL form and each property of the shape will be translated to one input field in the form. Similar to the DASH principle, it uses certain GUI components for different SHACL constraints, although unlike DASH it it not provided in the SHACL shape as annotation but hard coded for certain SHACL constraints. An example of this is using a GUI date picker for a date constraint, number input for a number constraint and so on. The component support nested SHACL shapes, but only if the nesting is explicit in the shape and not a reference. For example a complex GeoLocation property which is defined elsewhere in the file is not supported. On the other hand, if this GeoLocation property is defined as a nested property of the original SHACL shape it is. These nested structures are visualized as a nesting of the form.

[4] https://www.w3.org/TR/shacl/
[5] https://www.slideshare.net/jelabra/shacl-by-example
[6] http://datashapes.org/forms.html
[7] https://github.com/Babibubebon/vue-shacl-form
[8] https://github.com/CSIRO-enviro-informatics/shacl-form

2.3 Linked Data fragments

Linked Data fragments[9] [10] (LD fragments) are as the name suggest methods for fragmenting linked data. The most basic LD fragment is a **data dump**, which returns all data as a single fragment. The opposite of this spectrum are **SPARQL endpoints**, which return the custom result of a SPARQL query as a single fragment. These two extremes forms of LD fragments both have their share of disadvantages. Data dump suffer from a high client cost for processing and a high bandwidth use. SPARQL endpoint suffer from a high server cost and a low availability. The main research field of LD fragment is exploring the middle ground between these two extremes and to find more optimized methods of publishing data to the web. Two intermediate LD fragments are LD Documents and Triple Pattern Fragments. **LD documents** return all related information of an entity. This is every triple where the entity is either subject or object. **Triple Pattern Fragments** [11] uses intelligent clients [12] to efficiently retrieve the required fragments from the server to resolve users query. This query is defined using a triple pattern hence the name. These requested fragments are cacheable since the client only request simple reusable queries and does the othet processing itself, hence *intelligent* clients. A third option in the middle are Event Streams. This method of LD fragments publication will be explored in the next section. **Event Streams** are a type of LD fragments which focus on reducing the replication problem which presents itself with a data dump of live data while keeping the server side complexity low. To be up to date with the latest data and detect changes to the dataset, the user application would have to retrieve and inspect the full dataset each time. This is a cumbersome process that wastes bandwidth and client resources for each application. Event Streams solves this by not publishing a data dump, but the changes in data. This vastly reduces the required data and eliminates the need for the client to do a full dataset comparison each time.

An Event Stream fragment hold information about the changes, meta data and links to traverse the data. Event Streams fragments use the tree ontology for traversal. The tree ontology has in its vocabulary entries such as *tree:lessThanRelation* and *tree:moreOrEqualThanRelation* which can be conditioned on specific properties and refer to the next fragment. A simple example of such is pagination in the time dimension, with the newest records being present in the first page. Using a *tree:lessOrEqualThenRelation* on the generated time property with as value the oldest time of the current fragment and a link to the next page. A second example is using a combination of *tree:greaterThanRelation* and *tree:lessThanRelation* to create geo spatial fragmentation, with as properties the latitude and longitude of the samples. An example of this is presented in figure 2.3.

[9] https://linkeddatafragments.org/

Listing (2.1) A temporal tree ontology relation

```
{
    "@type":
        "tree:GreaterOrEqualThanRelation",
    "tree:node":
        "https://urlToFragment",
    "sh:path": {
        "@list": [
            "prov:generatedAtTime"
        ]
    },
    "tree:value":
        "2020-11-28T13:14:56.848Z",
    "tree:remainingItems": 107
}
```

Listing (2.2) A geospatial tree ontology relation

```
{
    "@type": "tree:LessThanRelation",
    "tree:node":
        "https://urlToFragment",
    "sh:path": {
        "@list": [
            "ngsic:location",
            "ngsic:value",
            "ngsic:coordinates",
            "rdf:rest",
            "rdf:first"
        ]
    },
    "tree:value": 51.01375465718819,
    "tree:remainingItems": 9
}
```

Figuur 2.3: Two tree ontology relations which link to other fragments.

3

Implementation

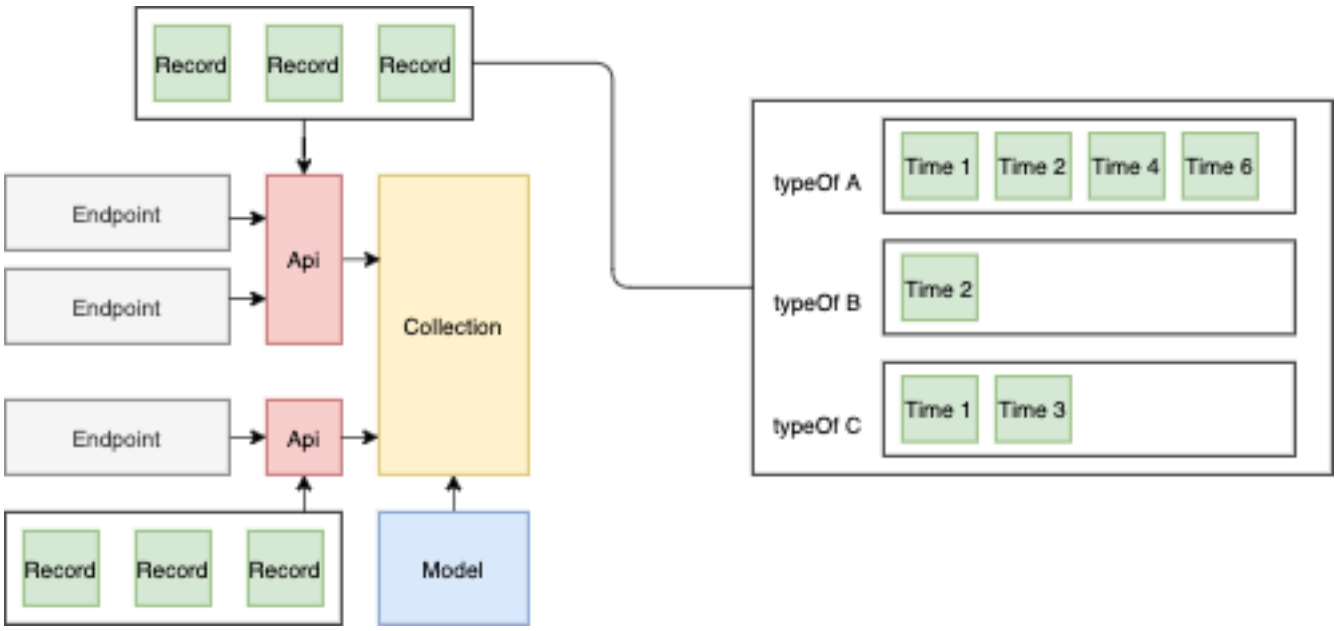
This work will modify ApiApi in order to enhance its functionality with the features as outlined in the introduction using the technology explored in related work. Most notably the Api creation tool will be revised to support semantic models and the endpoint will be transformed to an event stream. To do achieve this first a revision of ApiApis internal structure should be made, since no concept of history is present in the current implementation. The first part of this chapter introduces the changes to the ApiApi structure, after which an more in depth overview of the implementation of the two most notable components is provided: The Api creation tool and the endpoint.

The base version of ApiApi provided a logical structure existing out of Apis, Collections and Models. To recapitulate each collection would exist out of any number of Apis and a single Model. For each Api a mapping would then be created which transforms the data to conform with the model. Each Api consisted out of a single endpoint and a mapping. Data could be requested on Collection level, which resulted in a fetch to the endpoint of each Api in the Collection and the corresponding mapping to create a single fragment which is a live data dump. Caching can be done per Api, and if presented with a request the cache would act as a buffer for the endpoint fetch if the data was fresh enough. In this model, data is always live (or close to live using the cache). No concept of version is present. For this reason a new logical entity is proposed, a Record. A **Record** is used to represent data of an Api at a certain time. Records are kept per data point, each Record can thus be seen a being a version or a snapshot of the state of the data

point at set time. The Record stands in relation with the data point as being a *typeOf*. When data changes over time, a new Record is created instead of editing the older version. Having a set of Records of the same type thus gives an overview of the evolution of the data point in the time. Records are logically part of an Api, and each Api holds a set of Records originating from its endpoints. Finally each Record can be tagged with a location. This location can be used when solving a geo spatial query.

A second alteration to the logical structure has to be made based on the observation that information about the same logical data points can be distributed between different physical endpoints. An example of this are two endpoints one of which holds the location of all sensors while the second holds its readings. If the corresponding Model required both of these features this data would be unusable, since no Api can be created which holds both location and reading. Creating an individual Api for each endpoint would not work, since each Api is required to comply to the Model in itself. Apis are individual entities which do not relate to each other in one way or another, so two Apis with only half the required data would be created. It should be noted that while this data is present in different locations, it conceptually contains information about a single data point, being the sensor which has a location and a reading. Two allow such structures to be used, the implementation of an Api is changed to contain any number of endpoints instead of one. Each endpoint added to an Api holds a part of the information of the logical data point and all information combined suffices to comply to the Model. This should be differentiated from differnt Apis itself, since these do not hold complementary information about the same data point but information about different data points. To illustrate this with an example a first Api could contain two endpoints, one which holds location and one which holds readings from sensors from company A, while the second Api contains a single endpoint which holds both location and readings of all sensors of company B.

With the revised internal structure covered, the next two section will cover the creation of an event stream and the creation of a semantic mapping. Creation of an event stream can be split will be covered based on two distinct questions, *How to generate the Records* and *How to generate the event stream*. Creation of a semantic mapping will be subdivided in *Creating a semantic mapping* and *Defining a model*. *Creating a semantic mapping* is further subdivided based on the method used. Implemented methods are *RML via Text*, *YARRRML via Text* and *RML via Form*.



Figuur 3.1: The updated logical structure of ApiApi

3.1 Creating an event stream

3.1.1 How to generate the Records

ApiApi is designed to transform simple JSON endpoints. These endpoints mostly take the form of a data dump with the live information of each data point without notion of history. A method should thus be provided to keep track of changes of the Api with only the current live data as resource. ApiApi does this by sampling the Api and creating a version its response. This way the response data can be evaluated and changes can be detected. This is where the Records are used. Each Record is the state of the a single data point of the Api at a set time, in this case the sample time. Each sample interval a Record of each data point can be generated. Evaluating these Records thus gives an overview of the changes of the data point. ApiApi does not generate all possible Records however but only keeps Records of changed data points. This minimizes the data stored while keeping all information since the data set can be reconstructed at any time by reverse applying the Records to the current data set.

A problem with sampling however is that it is a *lossy* method of change detection. If the sample rate is larger than the update rate of the sensors observation, two consecutive sensor updates can appear where the first is being mascaraed by the last. ApiApi tries to minimize this problem by allowing the sample rate to be configurable, so it can be adapted to the applications needs.

Generating Records is not straight forward however and needs some additional information

from the user to correctly uniquely identify data points and tag its location. For this reason an additional information is requested from the user when creating an Api. This information includes the path to the unique id of the data point and the path to its location. The unique id is required for detecting changes, since without unique identifier a data point could not be distinguished from other and not compared to its previous versions. This id doubles as the typeOf value, since the Record represents a version of the the data point uniquely identified by the id at a set time. The location path is used to extract the coordinates of the data point to location tag the Record with. Note that neither of these are required, however not supplying certain paths can limit the features ApiApi can provide. In this case, not supplying an id path is the minimum requirement for generating the event stream. Supplying a location path is not, but if not provided geo spatial fragmentation will not be available on the stream.

Sampling the data is done by creating an asynchronous request for each Api (and by extend for every endpoint in the Api) every interval time. Each data point from the response is hashed and compared to its previous version based on the data points id. Unchanged hashes and the corresponding unchanged data points are ignored, for changed data points a Record is created. This Record is then further populated using a location tag if provided. Sampling is not done by default but can be controlled for each Collection. On the Collection overview screen the user can select to start or stop the sampling, change sample and fragmentation settings or to clear the hash cache. In the latter case each data point will be seen as new since it is not present in the hash cache anymore.

3.1.2 How to generate the event stream

The event stream is generated from the Records generated by sampling. Depending on the request, a data base query is constructed which selects the corresponding Records and maps these using the supplied RML. The mapped data is then encapsulated into an event stream fragment and provided to the consumer. Each event stream fragment can also hold two contexts, one for the fragment as a whole which provided in the root object, and one for each individual data point in the fragment. The former is by default populated with context for the fragments meta data, such as the tree ontology which links the fragments. The user is able to add context to both by defining additional context elements when creating the Model. Each event stream request to the Collection which uses this Model will then be accompanied by the extended context. Each data point can be configured to compact according to its context by default, which greatly increases the readability and decreases the size of the request. Besides context the event stream encapsulation holds information about links to other fragments as is the custom of an event stream. The base event stream holds links to its temporal neighbors. If when requesting a geo spatial fragment additional links are provided to neighboring spatial fragments.

Several endpoints provide an entry to the event stream. A full list of supported stream endpoints is listed in figure **??**

Each event stream fragment is generated at request time. The required Records are collected using a database query and mapped using the RML provided. After generating a fragment the fragment is cached so that consecutive request of the same fragment will result in a cache hit. Each fragment also comes with a cache time parameter in the GET response. This parameters contains the time the consumer should cache the fragment. Cache time can be configured when starting the sampling process. Cache time of the most recent fragment is set to the sample time since this fragment can still change, and the a change will occur at most once every sample interval.

Tabel 3.1: Event stream entry point

Type of request	GET
Appendix to base	\stream
Description	Default entry point to the event stream, returns the most recent fragment

Tabel 3.2: Single event stream fragment

Type of request	GET
Appendix to base	\fragmentID\stream
Description	Returns the specific fragment using its fragmentID

Tabel 3.3: Event stream entry point using spatial fragmentation

Type of request	GET
Appendix to base	\stream\zoom\x\y
Description	Default entry point of the geo spatial restricted event stream. This stream only contains data points which are located in the tile provided. Returns the most recent fragment

Tabel 3.4: Single spatial event stream fragment

Type of request	GET
Appendix to base	\fragmentID\stream\zoom\x\y
Description	Returns the specific fragment using its fragmentID

Tabel 3.5: Retrieving the event stream fragment which includes the time stamp

Type of request	GET
Appendix to base	\stream\UNIXtimestamp
Description	Returns the fragment which temporally contains this timestamp. If multiple fragments contain this timestamp the first one is returned.

Tabel 3.6: Retrieving the spatial event stream fragment which includes the time stamp

Type of request	GET
Appendix to base	\stream\zoom\x\y \UNIXtimestamp
Description	Returns the fragment which temporally contains this timestamp. If multiple fragments contain this timestamp the first one is returned.

3.2 Creating a mapping to a semantic model

This work proposed an adaptation to the current mapping and model creation workflow to allow mapping to semantic data models and validation of such mappings. To create mappings, the RML semantic mapping language will be used as base. Two extensions of this will be implemented as well being YARRRML and the ability to create an RML mapping by using a GUI form. For validation SHACL is proposed as solution which allows for validation of the mapped shape. SHACL will server a dual purpose, since besides validation of the shape it will be used for generating the RML form.

3.2.1 Creating the mapping

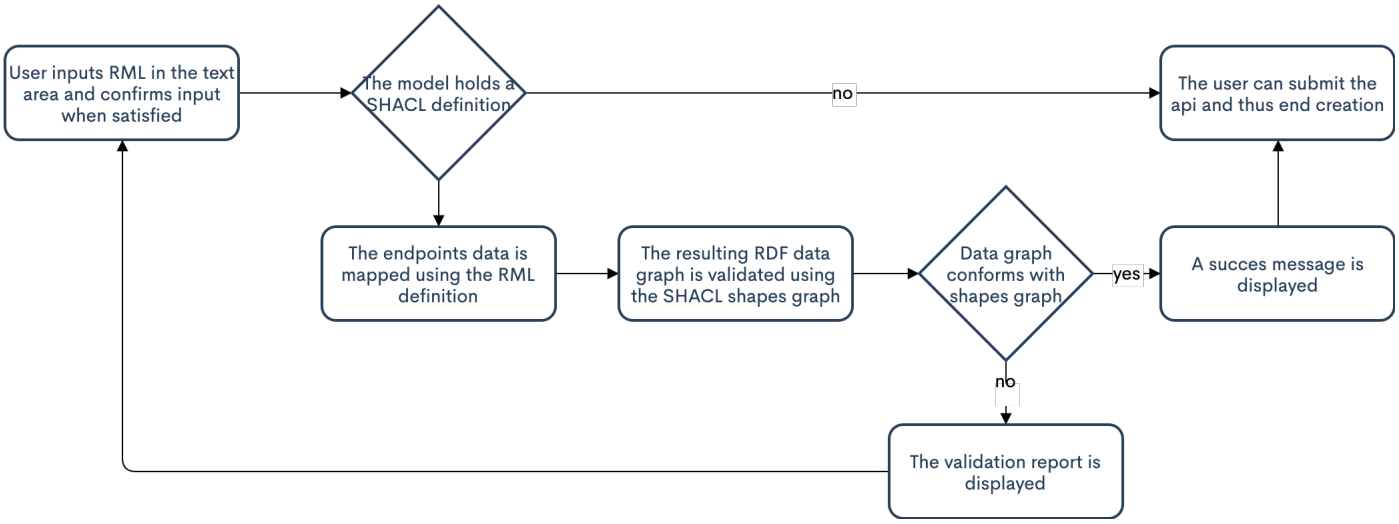
Three methods are proposed to create a mapping, being RML via text, YARRRML via text and RML via form. The first two are straightforward and require the user to create an RML or YARRRML mapping of his Api and provide it in plain text when creating an Api. This mapping can be made in ApiApi itself where a text box is provided, but more specialized tools such as the RML editor [1] can be used as well. The same holds true for YARRRML mappings. To ease the use of creating such mapping and increase its accessibility a third option is provided in a form which generates RML. The form provides the user with a set of inputs which he can use to transform his original data model to the model defined by the SHACL definition. This work flow differs from the the RML Editor since the user does not have to provide the property which he wants to map to himself. These properties are extracted from the SHACL shape. The only input required is the path which defines where this property can be found in the original data. This subsection will provide a more in depth description of the work flow and implementation of all three methods.

When creating an Api, one of the three methods can be selected after providing the general Api information in the creation header. One selecting an option a the corresponding Vue.js component is loaded. Not all options are available at all times. The availability depends on the structure of the Api. Currently YARRRML via text and the RML form are not supported when an Api has multiple endpoints.

3.2.2 RML via text

When RML via text is selected the RML editor component is loaded. The component itself is simply a textArea with a complete button. The user can insert his RML definition in the

[1] https://app.rml.io/rmleditor/



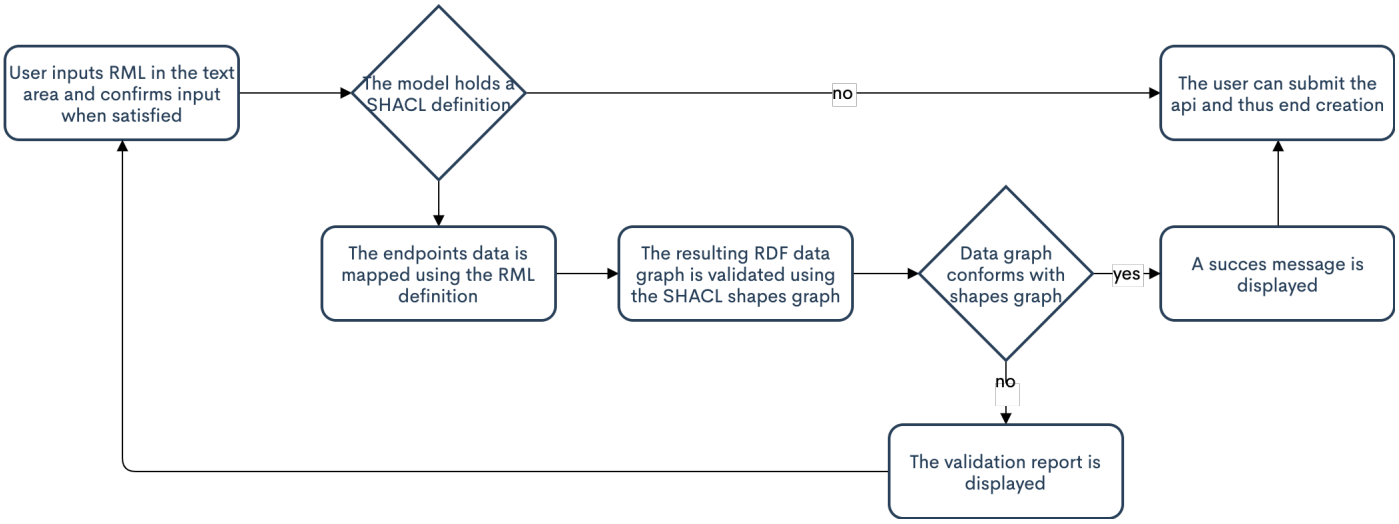
Figuur 3.2: RML via text validation work flow

text box and select complete when satisfied. Upon completion, the RML definition is set as local variable which will be included in the creation of the Api database entry. If the current data model holds a SHACL definition, the RML is first sent to the back end for validation before it can be submitted. This is done by first mapping the users endpoint data using the RML definition and then validating the result using the models SHACL constraints. For RML mapping the RMLio/rmlmapper-java-wrapper-js[2] is used. SHACL validation is done using TopQuadrant/shacl-js[3]. When the generated data graph complies with the shapes graph a success message is relayed to the user and the submit button is enabled for him to submit his endpoint. When the data graph does not comply, the corresponding validation report is relayed so the user can view the error and adjust his RML definition accordingly. After adjustment, the user can reconfirm his RML definition. When the data graph does not complies, the submit button is disabled.

3.2.3 YARRML via text

When YARRRML via text is selected the YARRRML editor component is loaded. The component itself is simply a textArea with a complete button. The user can insert his YARRRML definition in the text box and select complete when satisfied. Upon completion, the YARRRML definition is set as local variable which will be included in the creation of the Api database entry. If the current data model holds a SHACL definition, the YARRRML is first sent to the back end for validation before it can be submitted. Since there is no YARRRML mapper, the

[2]https://github.com/RMLio/rmlmapper-java-wrapper-js
[3]https://github.com/TopQuadrant/shacl-jsrmlmapper-java-wrapper-js



Figuur 3.3: YARRRML via text validation work flow

YARRRML is first converted to RML before mapping using the rmlio/yarrrml-parser [4]. After generating the corresponding RML, the endpoints data is mapped and validated using the models SHACL shapes graph. As with the RML editor, RMLio/rmlmapper-java-wrapper-js[2] and TopQuadrant/shacl-js[3] are used. If the data graph complies with the shapes graph, a success messages is relayed to the user and the submit button is unlocked. The user can then submit the Api. If the data graph does not comply, the validation report is relayed such that the user can use this information to update his mapping definition.

3.2.4 RML via form

When selecting RML via form the RML Mapper component is loaded. Unlike the previous described components, the RML Mapper is a more complex and layered component. The RML Mapper is structured is visualized in figure 3.5. This section starts with a simple introduction of the component from the users perspective. Secondly the implementation of the two major steps is explained. Finally, additional information about the components structure and their relation is provided.

The RML mapper is a form like component where three different parts can be identified: the header, the endpoint data view and the mapper view. **The header** is used to input general information about the endpoint. File format, Target and Iterator are required fields. Under File Format, the user selects the file format of his endpoint. In the current implementation only JSON endpoints are supported. Under Target the user selects the shapes graph which is the

[4]https://github.com/rmlio/yarrrml-parser

target for his mapping using a drop down select box. These shapes graphs are retrieved from the SHACL file which is defined when creating the data model. In the iterator field the user uses the JSON path[5] notation to define the iteration to be executed on his endpoint data. In the **data view** the user can view the data coming from his endpoint. This eases the mapping since the user can see the original data which should be mapped. In **mapper view** the user can define the actual mapping. This view uses the selected shape graph to generate a vue.js form which can be seen under the Form tab. Input in these fields are translated to equivalent RML triples which can be constructed in real time under the Data tab. Finally the SHACL shape graph can be consulted in the Shapes tab.

The working of the RML Mapper involves two major steps. The generating of a Vue.js form based on a SHACL definition, and the generation of RML rules based on the users input.

3.2.4.1 Generating a Vue.js form from a SHACL shape graph

Generating the form is done using a an adapted version of the Babibubebon/vue-shacl-form[6]. Generating the form from SHACL has mainly remained the same. One important change however is the addition of a slider which identifies if the property is a constant or not. If the property is a constant a value is expected while if not a path.

3.2.4.2 Generating RML from user input

While form generation was a conceptually simple matter of porting from one framework to another, generating a mapping is a more thought of process. First of it should be noted that generating RML is not the goal per se. Since the generated RML is not intractable and thus in essence transparent for the user, any kind of data structure for mapping would suffice. The goal of creating a GUI is in the first place geared towards user accessibility. The choice for RML is made rather out of a technical standpoint. Technically, the choice makes sense since an RML mapper is already in place because of the RML via text implementation. So this service can be reused. Additionally, the original Babibubebon/vue-shacl-form[6] component generates a data point expressed in RDF and Turtle, which is similar for RML. Because of this, the RDF functionality can be reused in parts. An additional advantage of choosing RML is uniformity with the rest of the mapper tool. So a knowledge of RML would suffice to understand the underlying principle of the entire mapping tool. In essence, the RML via form method serve a similar goal to YARRRML in making an RML mapping more accessible. Because of this, and while not in-

[5]information about the JSON path syntax can be found here: https://restfulapi.net/json-jsonpath/
[6]https://github.com/Babibubebon/vue-shacl-form

tractable, the user can see the generated RML mapping in the data tab before submitting. This way he can see the result of his visual mapping and get accustomed to the RML mapping concept. Which in term will lead to the ability to use the more complex and direct RML via text tool.

Before evaluating the implementation, it should be noted that the RML Mapper does only support a part of the features of RML. The idea of the tool is to allow for simple mappings of common data points to happen in a quick and simple way. However, the tools usability could be already significantly improved by the addition of relatively simple features. A suggestion of such can be found under future work. Additionally, for a more complete experience of the RML mapping functionality, the RML by text or YARRRML by text methods can be used.

Conceptually, using the form for data generation and data mapping are similar. When generating data, the actual content is provided. While mapping, a pointer to where the actual content can be found is provided. Because of this similarity, using a data generation form for data mapping is a logical solution. Since RML is used to specify mappings the question can be narrowed down to: How to link form field input to a or several RML triples. As starting point a comparison is made between a simple data point produced by the SHACL form and an RML mapping which would produce such data point. Several similarities can be observed. When excluding the prefix and rml:logicalSource triple, an one on one comparison can be made between the lines of the data point and the remaining RML triples. This is demonstrated in figure 3.4. From this comparison can be noted that an RML triple can be constructed from the form in a similar manner to the construction of a data point. But instead of entering the actual data, the user should enter the path to the property in his data. The mapper must then generate a number of RML triples instead of a simple data triple. Using this observation, the forms implementation can be modified to use the users input to construct a RML triple. This rests an analysis of rml:logicalSource, the triple we left out in the comparison. This triple cannot be matched with data in the generated data point. For this, additional fields have to be added. One for the iterator, and one for the file type.

3.2.4.3 Component structure

A brief overview of the Vue.js component structure will be given. The mapper inherits the base component structure of the Babibubebon/vue-shacl-form[6] component, which it is based on.

The **ShaclMapper** component is the top component which ties the RML generation together. It provides meta information needed for the mapping and the initialization of a ShaclForm component. First of, it provides inputs for file type, target class and iterator. As discussed above, this information can not be found in a Shacl definition so these fields are not generated in the

Listing (3.1) Subject of the data point

```
<https://www.donkey.bike/5743>
```

Listing (3.2) RML Subject mapping

```
rr:subjectMap [
  rr:template
      "https://www.donkey.bike/{recordid}"
];
```

Listing (3.3) Type predicate object of the data point

```
  a ngsi:BikeHireDockingStation ;
```

Listing (3.4) Type RML mapping

```
rr:predicateObjectMap [
  rr:predicate rdf:type;
  rr:objectMap [
    rr:constant
        ngsic:BikeHireDockingStation
  ]
];
```

Listing (3.5) ngsic:name predicate object of the datapoint

```
ngsic:name "Station
    Antwerpen-centraal" .
```

Listing (3.6) ngsic:name RML mapping

```
rr:predicateObjectMap [
  rr:predicate ngsic:name;
  rr:objectMap [
    rml:reference "fields.name"
  ]
].
```

Figuur 3.4: Comparison between the Babibubebon/vue-shacl-form[6] generated data point and an RML graph which maps to such data point.

ShaclForm component. However, they are require for the generation of an RML mapping. Additionally, the ShaclMapper provides gives the user the option to view the input data, which increases the ease of creating a mapping. The user is also able to browse between three tabs. One holds the ShaclForm component, one the Shacl Definition used for generation of previous mentioned and the final tab holds the generated RML mapping. This way the user can view the RML his input generates.

From the **ShaclForm** component on the component bases are taken from the Babibubebon/vue-shacl-form[6] and modified to suit RML mapping as described above. Because of this, the same component structure is kept. The ShaclForm is responsible for creating the final RML shape, for this the ShaclMapper provides it with the user input for the iterator, file type and target class. From this information it generates quads and combines these with the quads received from the user input in the FormGroup.

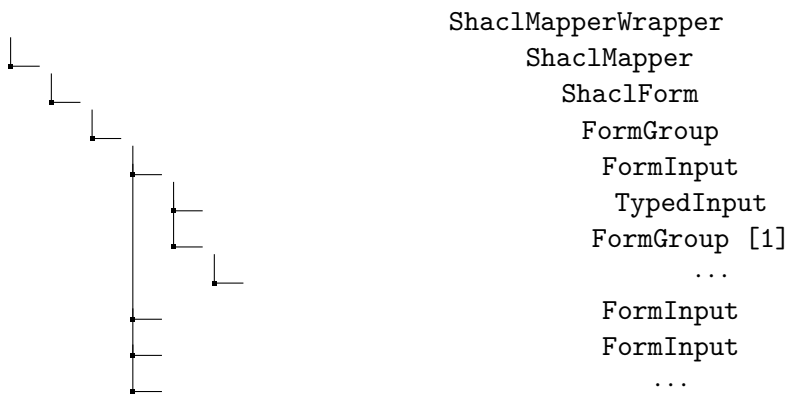
The **FormGroup** component is the wrapper used to generate the form. It iterates over Shacl fields and generates a FormInput for each. Besides this, it holds little logic.

A **FormInput** corresponds with a single property of a Shacl shape. This also holds the single property controls, such as the constant switch. The original component also used this component to recursively generate another FormGroup for nested Shacl definitions. In the ShaclMapper this functionality if not currently available. But this component can be adjusted to again support such functionality.

TypedInput is the final component in the structure. It corresponds with the input field for the Shacl property. The current implementation only support text fields, but additional types can be introduced to support data selectors, sliders etc. The Shacl data type can then be used as switch to identify which input field is best suited.

3.2.4.4 Comparison of mapping methods

As conclusion to this section about mappings methods an overview of the different implementation is provided. To illustrate the differences between the three proposed methods and the original method a comparison is constructed based on five characteristics of the mapping process. Note that is comparison is not meant as evaluation but rather as illustration to provide a structured overview of available methods and how they relate to the original work flow. The first characteristic that will be compared is the the method with which the user can input the mapping. Two options this work proposed are via text or via a form. A second characteristic is how the mapping is defined. The impact to the user of this definition depends on the first



Figuur 3.5: The component structure of the RML Mapper [1] Not ported functionality

characteristic. This is because when using a form there is less need to interact with the definition since it is generated as result of the forms input and thus is transparent to the user. When using text however the definition is saved as is and the user should use the conventions of the definition to construct his mapping. The third and forth characteristics are similar to the first two but for the model instead of the mapping. The final characteristic is used to compare the method with which a mapping is verified. These characteristics are rephrased as question and listed in table 3.7 and compared in table 3.8.

Tabel 3.7: Questions

Number	Question
3.7.1	How to input a mapping?
3.7.2	How to define a mapping?
3.7.3	How to input a model?
3.7.4	How to define a model?
3.7.5	How to validate a mapping?

Tabel 3.8: Comparison of different mapping methods [1] Not intractable since generated by the form [2] Automatically generated based on the model

±	Q 3.7.1	Q 3.7.2	Q 3.7.4	Q 3.7.3	Q 3.7.5
ApiApi by form	Custom[1]	Form[2]	Custom[1]	Form or example	DB validation
RML by text	RML	Text	SHACL	Text	SHACL
YARRRML by text	YARRRML	Text	SHACL	Text	SHACL
RML by form	RML[1]	Form[2]	SHACL	Text	SHACL

3.2.5 Creating a model

Creating a model is done by providing a SHACL document which holds information about the shape of the graph and constraints on it. One SHACL file can contain multiple shapes. The SHACL shape serves a dual purpose. First it is used for validating the mapping the users inputs by validating the mapped data. Secondly it is used to generate the RML form which can be used to generate an RML mapping as described above. Note that although the SHACL shape is used to generate the RML form, using the RML form as mapping method does not imply compliance to the model by default. SHACL can be provided when creating a Model. On the Model creation page a text box is present where a SHACL shape can be inserted. When submitting the Model this shape will be saved together with the other Model information.

4

Evaluation and conclusion

Evaluation of the proposed system will be done by first providing an overview of the implemented functionality after which will be discussed how this functionality provides the tools necessary to successfully implement the user stories. ApiApi in its current form is a framework for combining, mapping and publishing linked data fragments from simple JSON data sources. This accomplished by combining RML, SHACL and event stream technology. These are combined in a Vue.js interface which uses MongoDB and Redis for data storage. ApiApi has several optional features such as a graphical RML form and the ability to use geo spatial fragmentation on top of event streams. However not all features have to be used at all time. This evaluation will start with an overview of the features and the requirements to enable those. It is advised at least read the introduction to Chapter 3 Implementation as it covers the fundamental concepts of ApiApi.

Mapping data in ApiApi is done by using one of three mapping options. These options are *RML by text*, *YARRRML by text* or *RML by form*. In the first two the user has to provide an RML or YARRRML mapping in textual form while the last option allows the user to create an RML mapper based on form input. This form is generated using a SHACL definition in the Model. The textual input is alway available, however the RML form has requirements that have to be met before it can be used. These are that the Api should only have a single endpoint, and that the SHACL shape which generates the form is simple meaning it has no complex or multilayer properties.

Data validation is done using a SHACL definition that the user can optionally provide. When creating a mapping, this mapping is then evaluated on its compliance to the shape an rejected if it does not comply. Adding a SHACL shape is optional, but necessary if the RML by form input method is preferred for mapping.

Data publication has three modes. Each mode combines all data from every endpoint in a Collection and maps the data according to the provided mappings. The first mode stops there and provides a data dump of the mapped data. For this mode to work there are no requirements. The second publication mode publishes a single event stream of the data. Deltas are computed using a sampling technique and changes in the source data are published as part of event stream fragments. For this mode to work each data entity should have a unique id since this id identifies the entity and is used to detect changes. This id should further be in the same location for each data point in a single endpoint since the user should supply a path to the id while mapping. In the last mode an event stream is published but fragmented further using a spatial tag, in the case of ApiApi this is a slippy tile notation at the end of the URL. For this mode to work each data point should additionally have a location on a constant path. If spatial fragmentation is desired the user should include a location path in the Api creation. Note that not each endpoint has to have a location, but each data point. So if a single endpoint holds all locations for each data point additional endpoints do not need a location path. One location per record id suffices.

An overview of these features and their requirements are provided in table **??**. This table can be interpreted from both left to right as right to left. Reading left to right lists the requirements that need to be met for a certain feature to be available. For example if an event stream with SHACL validation with RML via text input is desired any amount of endpoints with any shape can be used as long a they have an id path. Reading from right to left shows the features available when a certain amount of data is available. For example if the user has several endpoint, no SHACL shape and does not supply an id path, he will be able to create only a mapping without validation using the RML or YARRRML via text methods. The remainder of this chapter will cover the user stories

(a) An overview of the features and needed requirements. v.t. = via text, EV = event stream, SF = spatial fragmentation

Features			Requirements		
F1	F2	F3	R1	R2	R3
Mapping	No Validation	RML v.t. / YARRRMLv.t.	Any amount	No shape	No data
Mapping	SHACL Validation	RML v.t. / YARRRMLv.t.	Any amount	Any shape	No data
Mapping	SHACL Validation	All	1	Simple shape	No data
Event Stream	No Validation	RML v.t. / YARRRMLv.t.	Any amount	No shape	Id path
Event Stream	SHACL Validation	RML v.t. / YARRRMLv.t.	Any amount	Any shape	Id path
Event Stream	SHACL Validation	All	1	Simple shape	Id path
EV + SF	No Validation	RML v.t. / YARRRMLv.t.	Any amount	No shape	Id + location path
EV + SF	SHACL Validation	RML v.t. / YARRRMLv.t.	Any amount	Any shape	Id + location path
EV + SF	SHACL Validation	All	1	Simple shape	Id + location path

(b) Features

	Features
F1	Endpoint functionality
F2	Data validation
F3	Mapping tools available

(c) Requirements

	Requirements
R1	Maximum amount of endpoints per Api
R2	Complexity of SHACL shape
R3	Provided fragmentation data when creating the Api

Figuur 4.1: Test

4.1 User story evaluation

The evaluation of the proposed system will done by evaluation the user stories proposed in introduction. Evaluation of each user story will be done by describing the method the actor has to perform to accomplish the story, together with the possible limitations and other comments. Requirements attached to the user stories will be evaluated in similar fashion together with the corresponding story. Since this evaluation includes a description of the work flow necessary to achieve each user story, it is also usable as use documentation for the ApiApi system.

4.1.1 Retrieving data from ApiApi as a consumer

This section will cover the retrieval of data from ApiApi as a consumer. This includes three distinct user stories being *As a consumer, i can receive a cacheable response*, *As a consumer, i can receive a semantically annotated response*, *As a consumer, i can receive historical information about the data*. First the different methods of retrieving data from ApiApi as a Consumer is covered, after which is explained how these methods can be used to fulfill the proposed user stories.

ApiApi provides a REST endpoint for each collection. The base for each request is the same, being *{baseURL}/api/data/{collectionID}*. The most basic request the consumer can send is a GET request to the *base* which will return all data from all Apis in the Collection mapped using the mapping specified when creating an Api.

Tabel 4.1: Retrieving all live data from all Apis in a Collection

Type of request	GET
Appendix to base	none
Description	Retrieve the live data of all Apis in the Collection, mapped to the Collections data model.

A second type of request the consumer can use is a request for the event stream of a collection. The event stream publishes updates of the data rather than the data set itself. Accessing the event stream can be done using the *base \stream* endpoint which will return the first fragment of the event stream. Fragments are paginated from new to old. Other fragments can be reached from the entry point first fragment by traversing fragments using the links provided in the meta data of each fragment. Each fragment holds a link to their two temporal neighbors.

Tabel 4.2: The entry point of a Collections event stream

Type of request	GET
Appendix to base	\stream
Description	Gets the first fragment of the Collections event stream which serves as its entry point. This fragment holds the information to traverse to other fragments in the temporal dimension.

ApiApi also supports goe spatial fragmentation using the slippy tile notation. By sending a GET request to the *base* *\stream\{zoom}\{x}\{y}* endpoint. A slippy tile is a geo spatial square defined using three parameters, one which define the zoom level and two to identify its location. The response will include the first fragment of the Collections event stream but with entries constrained to ones inside the slippy tile provided. As before fragments are paginated from new to old and each fragment contains meta data links to their temporal neighbors. Besides temporal neighbors however these kinds of fragments also hold links to their geo spatial neighbors, being the four neighboring tiles. Additionally ApiApi also provides the ability to request a single page using a time query. A full list of event stream queries is provided in figure **??**.

Tabel 4.3: The entry point of a Collections event stream

Type of request	GET
Appendix to base	\stream\zoom\x\y
Description	Gets the first fragment of the Collections event stream with results constrained to the tile as specified in the URL. This fragment serves as the entry point for this stream. This fragment holds the information to traverse to other fragments in both temporal as spatial dimension.

All responses contain a *cache-time* parameter which informs the user of the optimal cache time for the fragment. The cache time of a fragment is configurable by the administrator when creating a sampling process. An exception of this is the cache time of the most recent fragment of the event stream which is set to the sample rate.

Historical information can be retrieved using the event stream. The stream can be traversed to retrieve older updates. Retrieving historical data is limited by two factors. As first, since ApiApi uses sampling to generate the updates it is possible that not all changes are detected. This happens of concurrent changes happen faster than the sample rate. To this end it is possible that the event stream does not represent every change in the data set. Secondly the access to older fragments is limited by the expiration date set for records in the database. The System Administrator can configure records to expire and not be accessible anymore after a certain

amount of time. Of course the expiration date can be set to never expire in which case every record will be available for ever.

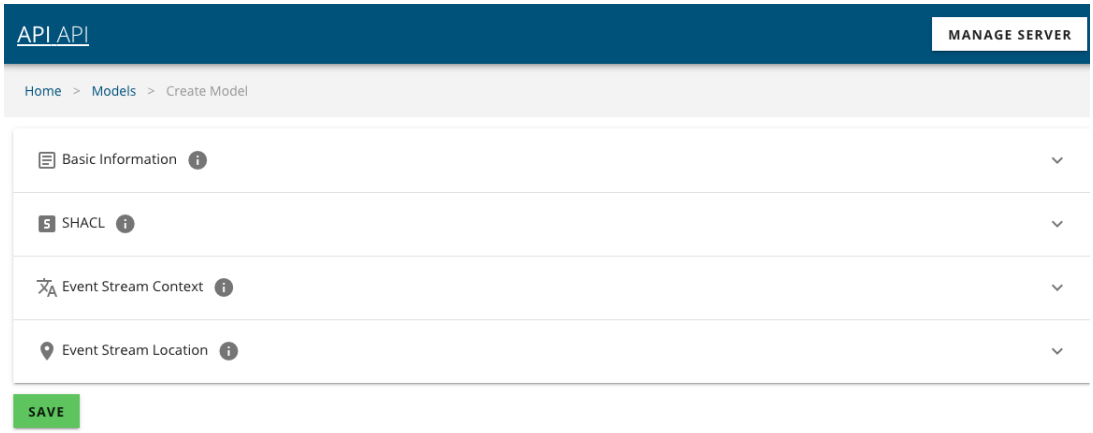
Three user stories where involved in the retrieval of data. A first required the consumer to be able to retrieve a semantically annotated response. This is completed by the response being serialized as JSON LD and the ability to include a context. A second user story is the ability to receive historical information about the data. This is done by providing an event stream which the user can traverse to recreate the history of the data as mentioned above. Finally the response should be cacheable. This feature is also achieved by using an event stream since changes in data are published as new fragment rather than amending older fragments. Because of this older fragments can be cache for longer periods of time. Only the most recent fragment is not cacheable since this fragment can change every time new updates are detected.

4.1.2 Using existing data models

This section will cover the imputation of existing data models into ApiApi. This concerns two user stories being *As a administrator, i load existing data models into the platform* and *As a administrator, i can provide restrictions for those data models.* To input existing data models SHACL is used. SHACL both provides the form of the model as can contain restrictions on this form. Creating a new model can be done by pressing the + button on the landing screen and going to the model creation screen. In this screen it is now possible for the Administrator to insert a SHACL definition which will be associated with the model. This SHACL definition will then be used to validate the mapped endpoint to and generate the mapping form which will be explained later. Since SHACL both holds a shape and can contain constraints on such it serves both user stories at once. SHACL covers a wide range of different shapes and constraints, but of course there are limitations to the extend that each constraint or shape that can not be modeled using SHACL will not be able to be verified using ApiApi. Another limitation is that there is only support for SHACL, so a shape or constraints as any other format are not supported. Several other constraint or modeling solutions exist such as SHEX, JSON schema validator etc. Due to the code design however any of the latter could be implemented fairly easy if desired.

4.1.3 Creating and mapping an Api

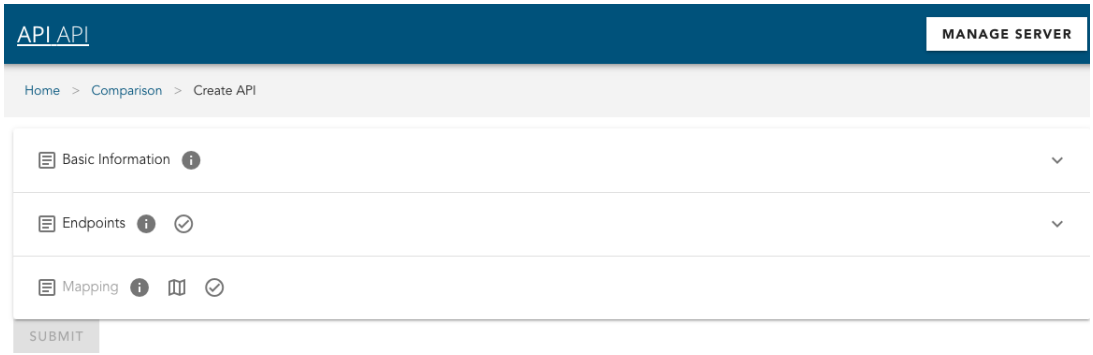
This section covers the creation and mapping of an endpoint to an Api. This concerns user stories *As a administrator, i can map an endpoint to a semantic data model* and *As a administrator, i can receive feedback about the compliance of my mapping according to the restrictions imposed on the data model.* Creating an Api exists out of two steps. First general information about the endpoint and fragmentation information should be provided. Secondly the mapping to the



Figuur 4.2: The model creation page

Collections data model should be made. General information starts with a name and the URL of the endpoint. ApiApi only supports JSON endpoints, and only endpoints which have its data points as a list. The name can be arbitrary since it is only used to differentiate between Apis. After this a base path can be configured which is necessary if the data points are not present in the root of the JSON object. For example a JSON object with a property *data* which is a JSON list of all data points should have a base path of $.data. The next tree text fields are used for the event stream and the fragmentation. The recordId field should point to the id which identifies the object uniquely. This is for example the id of the bike hire station or the sensor which produces data. Versions of this id will be created if the value changes. The two following paths should point to the latitude and longitude values of the record. These paths are resolved and used to tag each record with a location which allows for fragmentation in geo spatial tiles.

After completing the general and fragmentation info the user can select one of a few options for creating a mapping. The first option is RML via text, which allows the user to insert a RML mapping as text. A second option is YARRRML via text, which allows the user to insert a YARRRML mapping as text. Finally the user can chose RML via form which will generate a form based on the SHACL shape of the model for the user to fill in. From the input of the form an RML mapping will be created which represents the mapping. All three options require a verification before the mapping can be submitted. During this verification process the data



Figuur 4.3: The model creation page

will be mapped using the RML or YARRRML document provided and the mapped data will be compared to the SHACL shape. If there are no violations the user is able to submit his mapping. Otherwise the violations are displayed to the user and submitting is halted until they are resolved.

Submitting trough RML or YARRRML is straightforward, the user should use the RML of YARRRML spec respectively to create a compliant mapping and can simply paste the result in the text box. However since the RML form is designed for ApiApi specific some extra explanation is in place. The RML form has two functions. First it generates a GUI form from a SHACL shape, second it generates an RML mapping from the users input in the form. Before mapping a shape should be selected from the SHACL document, since a single document can contain several shapes. Once selected, a form is generated. Each input field of the form corresponds to a property as defined in the SHACL shape. Besides the properties generated form the SHACL shape, two fields are always present being the id template and the data path. The id template is used to construct the id for each data point. Using a template allows for the id which is commonly a number to be transformed into a URI. The data path corresponds to the path where the data points can be found.

Using RML, YARRRML text or the visual RML form mapping to a semantic model is possi-

ble. Using RML or YARRRML text complexed and layered semantic models can be generated from a simple JSON endpoints. However using this method has its limitations. The first two straightforward limitations of this approach are that mapping which are not supplied as either RML or YARRRML are not supported and should be translated to one of these. While Api-Api supports most of the RML specifications, it has the limitation which is the restriction to JSON endpoints. The RML specification has the ability to use several endpoints of different types in a single mapping which is not supported by ApiApi. When using the RML via form more limitation apply. Besides the restriction mentioned when using RML text, the form only supports simple SHACL shapes. This means shapes which have a flat depth without references to different shapes. For example a SHACL shape of a Sensor which has a different shape as child which is Location is not supported. Note that these shapes are supported when using RML or YARRRML text, but there is no GUI support.

After submitting a mapping, being it by way of text or form, the mapping is evaluated on its compliance with the SHACL shape. This is done by using the generated RML to map the data from the endpoint and evaluate the mapped data. If the mapped data complies the user is able to finish the Api creation with the current mapping. If not the user receives a report which states the infractions to the SHACL shape. The user is not able to submit the Api since the mapping provided results in invalid data. After adapting the mapping the user is able to resubmit and if compliant create the Api.

After a description of the functionality an evaluation of the user stories can be made. The first user stories requires the system administrator to be able to map to a semantic model. As mentioned above this can be done using either RML via text, YARRRML via text or the RML via form. The system was required to have support for JSON input and JSON-LD output which are both supported in ApiApi and part of the RML specification. As third requirement there should be a GUI tool which supports simple models which is provided by the RML form. The form is not able to use any SHACL shape as input, but as the requirement states is able to map simple models. In the case of ApiApi simple is thus seen as flat models with no references. The second user story states that he system administrator should be able to receive feedback about his mapping in a timely fashion to adapt his model in time. This is done using the SHACL feedback loop where the data is mapped and compared before the user is able to submit.

4.1.4 Publishing as linked data in fragments

Two user stories involve the publishing of linked data as fragments, these are *As a administrator, i can control the size of a fragment* and *As a administrator, i can publish linked data as cacheable fragments.* This overlaps partly with the user stories involving the consumer, since the data published by the system administrator will be received by the consumer. However in this section

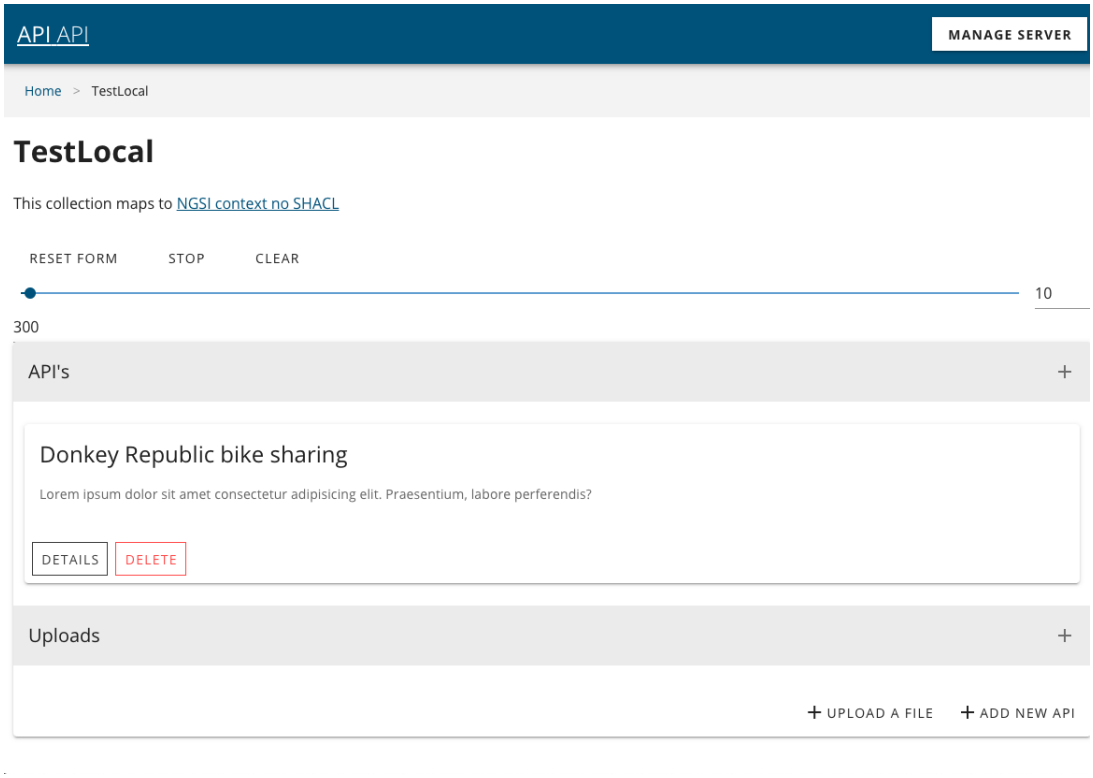
an approach from the administrators point of view is given while the previous evaluation was done from the point of view of the consumer.

Publishing data is done per Collection, and an endpoint is created automatically when a Collection is created. After adding Apis to a Collection request to this endpoint will then be populated with mapped live responses without any action from the system administrator. However to generate the Event Stream action is required. Caching the deltas is not done automatically and has to be started manually. This can be done in the Collection overview by creating a new sampling process. When creating this process the user is able to select the sampling interval, set the cache-time response parameter and set the local cache behavior. Since the hash cache is empty initially all records will be seen as new, however from the next sample on only the changed will be saved.

The user story states that the administrator must be able to publish cacheable linked data fragments. This includes three requirements being that the fragments should be linked, older fragments archivable (cacheable for a long period of time) and available as JSON LD. These statements can be evaluated one by one. First the user story states that it should be linked data and fragments which is trivial. These fragments are serialized as JSON LD which validates the requirement. Event Stream uses the tree ontology to create links between fragments. Temporal neighbors are reachable by link, and using this iteratable process every fragment is reachable from the entry point which is the latest fragment. Besides these links ApiApi also provides spatial fragmentation which when used includes a link to the spatial neighboring event stream defined as the neighboring slippy tile of the same size. This makes is possible to reach all information from any arbitrary fragment. Finally the requirement states that older fragment should be archivable. This is done by keeping the URI of a fragment constant over time. Since changes in the data generate a new record in the stream instead of changing the existing record, each record in the stream is thus in principle constant and does not change. These two characteristics make the fragment thus perfectly cacheble for longer periods of time. The only exception of this is the most recent fragment. While this fragment is not full, new records can be added which change the fragment. Therefor the most recent fragment should have a small cache time. The second user story states that the system administrator should be able to control the size of a fragment by adjusting the number of data point in a page. This is possible by changing the number of entries in a page when creating a Collection.

4.1.5 Summary of user stories

This section provides a single table which summarizes the user stories, requirements and how they are implemented in the proposed framework. Each user story is referred to using its number from table 1.1. Each requirement is refered to using its number from table 1.2.



Figuur 4.4: The Collection overview page

Tabel 4.4: A summary of the user stories and their evaluation. User stories are presented as using a grey cell color.

US/R	comment
1.1.1	see 1.2.11
1.1.2	All responses support JSON-LD
1.1.3	Event Stream keeps track of changes in time
1.2.6	Sampling interval is configurable
1.1.4	SHACL shapes can be loaded when creating a Model
1.1.5	SHACL shapes can be loaded when creating a Model
1.2.8	SHACL is supported
1.1.6	YARRRML and RML is supported for mapping to semantic models
1.2.1	RML and YARRRML support JSON input
1.2.2	RML and YARRRML support JSON-LD output
1.2.3	The RML form provides a user interface
1.1.7	A SHACL compliance report is provided when submitting a mapping
1.2.9	Feedback is provided before submitting
1.1.8	Fragments hold a configurable number of data points
1.2.4	Fragments hold a configurable number of data points
1.1.9	Event Stream is a sort of Linked Data fragment which is highly cacheable due to its temporal in variety
1.2.5	Temporal fragmentation is standard part of an Event Stream, geo spatial fragmentation is provided using the slippy tile convention
1.2.7	Event Streams use the tree ontology to traverse to other fragments.

4.2 Endpoint performance evaluation

This section will perform several experiments on the framework to simulate real use cases and compare its performance. First an experiment is performed which compares mapping in the proposed framework to the base framework. A second experiment is conducted which compares the impact of differences in the source data structure on the event stream fragment generation time. Finally an experiment is conducted which illustrates the impact of server side caching on response times. Most experiments are conducted multiple times to rule out any accidental irregularities, in that case the average is reported.

Depending on the experiment any combination of either the Donkey Republic location endpoint[1], Donkey Republic availability endpoint[2] or Blue bike location endpoint[3] are used. The Donkey Republic endpoints contain information about the location and availability of bike hire stations around the city of Ghent. At the time of writing both contain 572 entries. The Bluebike endpoint contains locations of bike hire docking stations around the city of Antwerp and contains 65 entries. The frameworks are ran locally on the same machine for each experiment. Postman is used to determine the total response time while intermediate times are reported using in code checkpoints. All experiments transform the source data to the NGSI bike hire docking station data model[4]. When possible the linked data variant of the model is used. All mappings in the proposed framework are defined using an RML mapping.

During comparisons three different sizes of fragment will be used. A fist type of fragment holds 5 data points and is around 2.5 KB in size. A second holds 50 data points and is around 23 KB in size and the last type of fragment holds 100 data points which is around 44 KB in size. Since 50 KB[5] is quoted as optimal page size the 100 data point fragment will be used for testing when possible.

Some experiments report the time of intermediate subprocesses. These are data retrieval, mapping, transformation. **Data retrieval** is the time it takes to contact the source endpoint and fetch the data. **Mapping** is the time it takes to transform the source data to the required shape using the mapping supplied. Finally **text** includes any additional data transformation which is done besides mapping. This is the compacting of the JSONLD according to its context, encapsulating the mapped data in an event stream fragment and computing different meta parameters for the event stream links. Finally **rest** reports the time which is not allocated to any of these

[1] https://data.stad.gent/explore/dataset/donkey-republic-deelfietsen-stations-locaties/

[2] https://data.stad.gent/explore/dataset/donkey-republic-beschikbaarheid-deelfietsen-per-station/information/

[3] https://portaal-stadantwerpen.opendata.arcgis.com/datasets/bluebike/

[4] https://fiware-datamodels.readthedocs.io/en/latest/Transportation/Bike/BikeHireDockingStation/-doc/spec/index.html

[5] https://pietercolpaert.be/research/2018/12/30/automating-reuse.html

Figuur 4.5: Live endpoint response time comparison between the original and proposed framework

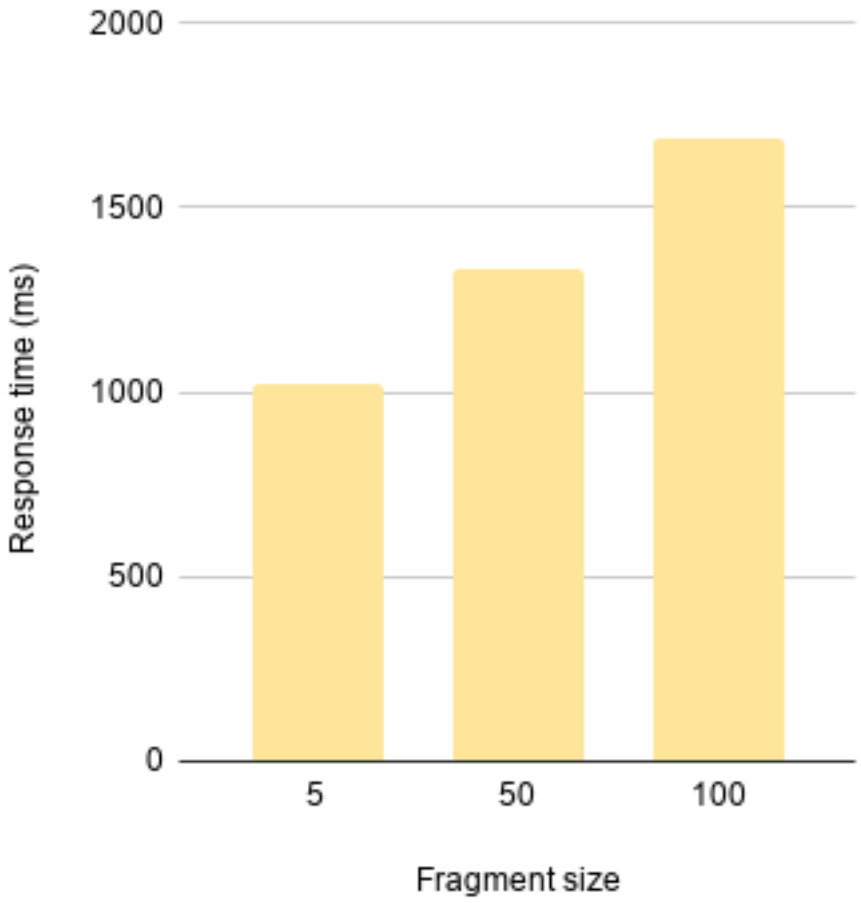
processes. Under the subprocess the percentage of response time it takes is displayed.

This first experiment compares the response times when creating a mapping between the proposed and original framework. The Donkey Republic location endpoint is used as data supplier. This endpoints contains 572 entries. Since the original framework does not support mappings to semantic models the regular bike hire docking station model is used. Both frameworks are contacted on their base endpoint which generates a mapped data dump. Response times and intermediate checkpoints are recorded and displayed in figure 4.5. This experiment concludes that the proposed framework is significantly slower compared to the base framework when mapping large data sets. The result clearly show this is due to a vast increase in the time needed to create a mapping.

The following experiment compares three different fragments sizes and their response times. Page sizes are 5, 50 and 100 data points with a approximate size of 2.5, 23 and 44 KB respectively. The Donkey Republic location data set is used as source.

There is clear difference in response time between the three sizes of fragment. This can directly be attributed to a longer mapping time. Results show that while larger pages have a longer generation time, the time per data point is clearly lower.

The next experiment simulates the creation of pages of an event stream. Each page is generated from a Collection with varying characteristics. The first Collection holds a single Api with the Donkey Republic location endpoint as data source. All data is gathered in a single sample interval. This Collection thus simulates an endpoint without changes. The second Collection has a similar configuration which is a single Api with the Donkey Republic location endpoint. However changes are simulated on the endpoint which results in records with three unique time stamps. For the purpose of simulation the changes are made in such a way that they can be included in a single page. Since change is equal to initial addition the endpoint is first populated using 33 records, after which 33 more are added after which the rest. The oldest page of the event stream thus holds records from three different time stamps. This page will be used. The



Figuur 4.6: Comparison in response time of different fragment sizes

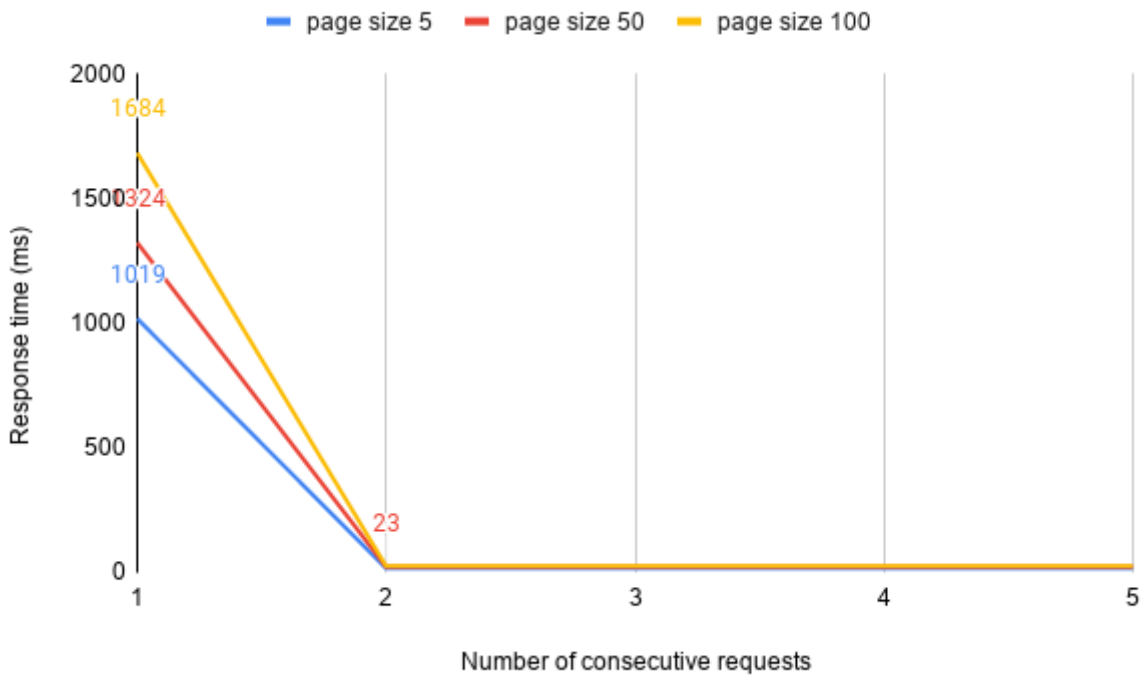
Figuur 4.7: Response times when generating pages from different types of Collections

last Collection is created from two Apis and three endpoints. An Api is created from the two Donkey Republic endpoints and one from the Bluebike endpoint. Data is kept constant. This page which will be examined has 72 data points from the Donkey Republic endpoint and 28 Blue bike data points.

This experiment is performed since page generation is handled differently is these three cases. When generating a page from Records of the same Api and same sample time a single RML mapping process is started which maps this data. This is simulated in the first example. However when a page has Records from multiple timestamps RML mapping is split to prevent prevent false duplicates from being generated. This results in an asynchronous RML mapping process being created for each time stamp in the page. Lastly when a page contains Records from different Apis, an asynchronous RML mapping process is created for each Api. Since each Api has its own mapping data from different Apis can not be mapped using a single process. Note however that this is not the case for each endpoint, a single mapping process is created for both Donkey Republic endpoints since they are contained in the same Api. The results of this experiment are displayed in figure 4.7.

The results show that limited diversity in sample times or data source have little impact on the page generation time. The minor increase in page creation time when handling multiple Apis could potentially be explained by the fact that the amount of incoming data points in the Donkey Republic case is now multiplied by 2. Each mapped data point now contains two original data points, one from the location endpoint and one from the availability endpoint. For this reason the RML process has to process 144 data points.

This experiment simulates the caching behavior of the framework. Multiple consecutive retrieval requests are made of the same page. The framework caches each page after generation, so from the



Figuur 4.8: Response times before and after cache hits.

second request a cache hit occurs and the fragment is returned from cache instead of generated. Figure 4.8 shows the achieved results. The results of this simulation clearly show the major time save the server side caching is able to accomplish. After a first cache miss and page generation a cache hit occurs for consecutive requests which is around 60 times faster. Average response time for the smallest page size lay around 17 ms, while the average response time for the largest page size lay around 27ms.

From these experiments can be concluded that while the proposed framework is slower in direct mapping, the cacheablility of the generated fragments allow for a very efficient server side cache implementation. Since each fragment is almost instantly provided after the first cache mis, the average time of retrieving a data fragment plummets after the first request. Additionally there is the possibility of precomputed the pages which removes the initial response delay entirely. Besides this efficient server side cache behavior, the client can also cache the fragment efficiently which eliminates the need for frequent full data set downloads to detect changes.

4.3 Conclusion

This work proposes a framework which transforms any number of simple JSON endpoints into an event stream. During each step linked data technologies are used. Mapping is done by using RML or YARRRML, constraints validation and model shaping by SHACL and publishing by event stream. The strength of the proposed framework lies in the ease to which these technologies can be applied to the source data. The proposed framework is a single tool which handles all the steps from source data insertion to publication. Furthermore all steps are done using a graphical user interface and are documented in application. The design of a custom RML via form method, although currently limited in functionality, further increase the accessibility. The modularity of the system allows its application in many use cases, from simple mapping to the creation of spatial fragmented event streams. The implementation of event streams as publishing method makes fragments highly cacheable both at server as client side since updates to the data are published as new fragments.

As for the performance the experiments show that while mapping is significantly slower than the base framework the cacheablility of the event stream fragments allow for close to instant responses after the first cache mis. Furthermore fragments can be precomputed to eliminate the first cache miss entirely. This is true for all fragments but the most recent which is still subject to change. However since the size of this fragment is at most equal to the page size the maximum response time is contained. Additionally the experiments show that a limited variety of origin and sample time have a minor impact on generation time of realistic sized fragments. For further improvement all experiments show mapping as by far the most time intensive task. However since there is a single RML mapper available at the moment there is limited possibility for improvement when using RML.

5

Future Work

This chapter will cover several aspects of the proposed framework which could be improved or expanded.

A first improvement to the RML by form method could be the support for more complex SHACL shapes. The limitation of only having support for shapes of depth one does limit its practical use. A first improvement could already be the support for commonly used complex structures such as Geojson objects. This could be hard coded into the mapper and input for such objects could be created. Another more scalable approach would be the support for any complex shape by recursion. The tool could potentially search the SHACL document (or any data source) for a shape which corresponds to the complex property and recursively creates a subform which instead of a simple text input now is a form in itself. The vue js shacl form already supported complex SHACL shapes, but only if the shape itself is complex without references. This functionality was neither ported nor extended when adapting the form for use within ApiApi however, but its existence in the original library could make porting it the the RML form easier. Additionally adapting the form to support multiple endpoints would further increase its usability.

Secondly the framework could be adapted to allow for addition or proposition of endpoints by users other than the Administrator. This would allow the framework to be used by multiple people without having to give every one full privileges. This could be useful in for example

community driven projects where contributers each add their own sensor to a larger collection. Two methods to achieve this are proposed. A first is the implementation of a user authentication system where users could log in and be granted privileges. Such method would allow users who simply want to contribute their sensor to the project to have minimal privileges and so on. An alternative is the creation of a client side version of the framework without the back end which focuses on mapping endpoints. This would allow users which are not administrators to create mappings and submit them to a server instead of saving them locally. The contributer would then just run a shell for the mapper tool and instead of creating locally send a creation request to an full ApiApi server. The server administrator would then be able to add the endpoint to his Collection or reject if the mapping or endpoint do not comply. Since ApiApi is written in Vue.js, implementing both of these options rather straight forward. The first requires some authentication middle ware while the last requires the mapper component to be extracted from the project and made into a stand alone application.

Bibliografie

[1] Digipolis, "ApiApi." [Online]. Available: https://github.com/lab9k/apiapi

[2] F. Goverment, "OSLO." [Online]. Available: https://overheid.vlaanderen.be/producten-diensten/oslo

[3] W3C, "RDF Primer," 2014. [Online]. Available: https://www.w3.org/TR/rdf-primer/

[4] ——, "RDF overview page." [Online]. Available: https://www.w3.org/RDF/

[5] D. Souripriya, S. Seema, C. Richard, and N. U. of Ireland, "R2RML: RDB to RDF Mapping Language," 2012. [Online]. Available: https://www.w3.org/TR/r2rml/

[6] B. De Meester, P. Heyvaert, R. Verborgh, and A. Dimou, "Mapping languages: analysis of comparative characteristics," in *Proceedings of First Knowledge Graph Building Workshop*, Jun. 2019. [Online]. Available: https://openreview.net/forum?id=HklWL4erv4

[7] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Van de Walle, "RML: a generic language for integrated RDF mappings of heterogeneous data," in *Proceedings of the 7th Workshop on Linked Data on the Web*, ser. CEUR Workshop Proceedings, C. Bizer, T. Heath, S. Auer, and T. Berners-Lee, Eds., vol. 1184, Apr. 2014. [Online]. Available: http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf

[8] B. D. Meester, P. Heyvaert, and T. Delva, "RDF Mapping Language (RML)," 2020. [Online]. Available: https://rml.io/specs/rml/

[9] ——, "Yarrrml," 2020. [Online]. Available: https://rml.io/yarrrml/spec/

[10] R. Verborgh, M. Vander Sande, P. Colpaert, S. Coppens, E. Mannens, and R. Van de Walle, "Web-scale querying through Linked Data Fragments," in *Proceedings of the 7th Workshop on Linked Data on the Web*, ser. CEUR Workshop Proceedings, vol. 1184, Apr. 2014. [Online]. Available: http://ceur-ws.org/Vol-1184/ldow2014_paper_04.pdf

[11] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert, "Triple Pattern Fragments: a low-cost knowledge graph

interface for the Web," *Journal of Web Semantics*, vol. 37–38, pp. 184–206, Mar. 2016. [Online]. Available: http://linkeddatafragments.org/publications/jws2016.pdf

[12] R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh, "Comunica: a modular sparql query engine for the web," in *Proceedings of the 17th International Semantic Web Conference*, Oct. 2018. [Online]. Available: https://comunica.github.io/Article-ISWC2018-Resource/

Appendix II: Setting up the framework

Requirements

ApiApi is developed and tested on Linux and Mac systems.
Node.js[1] is required. ApiApi is developed using version 14.13.1.
MongoDB[2] is required, ApiApi is developed using the community edition[3] and version 4.4.0
Redis[4] is required, ApiApi is developed using version 6.0.8

Startup

```
git clone https://github.com/rgdvilde/apiapi.git
cd apiapi
npm install
npm run dev
```

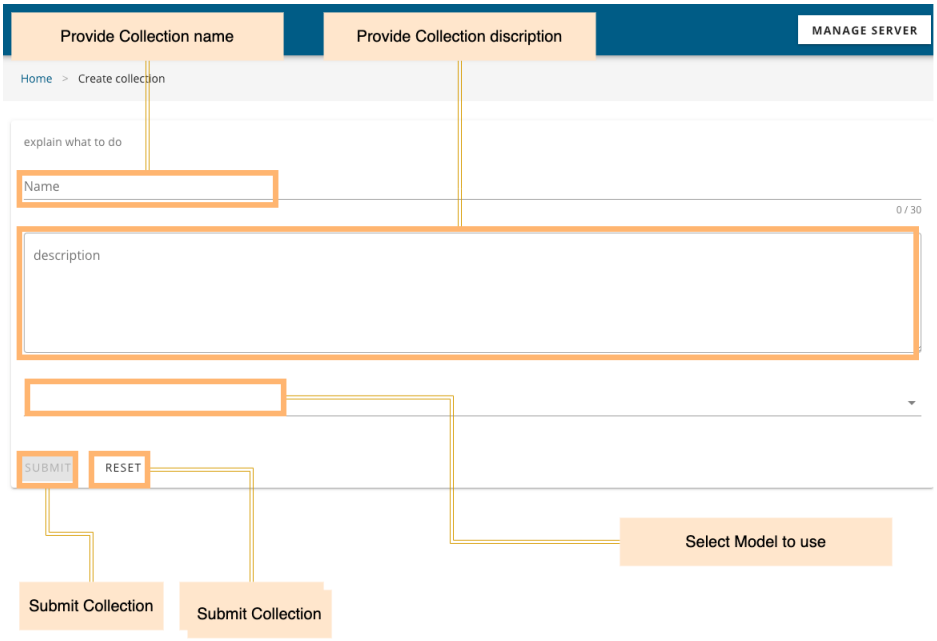
Listing 5.1: Cloning and starting ApiApi

[1] https://nodejs.org/en/download/
[2] https://www.mongodb.com/
[3] https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/
[4] https://redis.io/download

Appendix III: Visual user guide

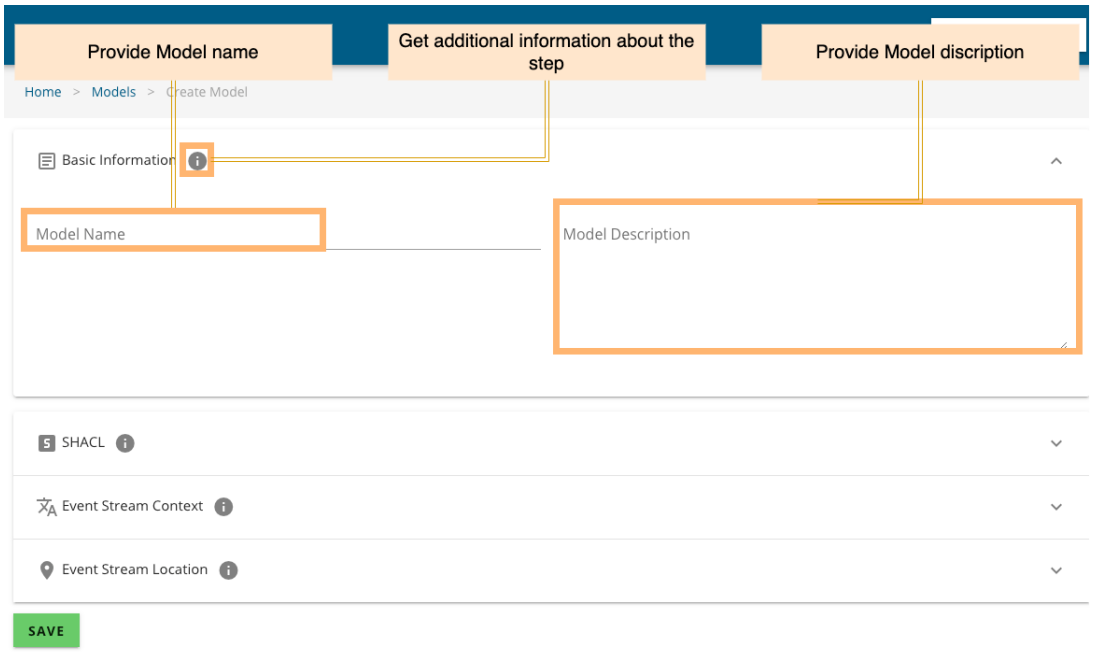
This Appendix provides several annotated screenshots of the proposed framework.

5.1 Creating a Collection

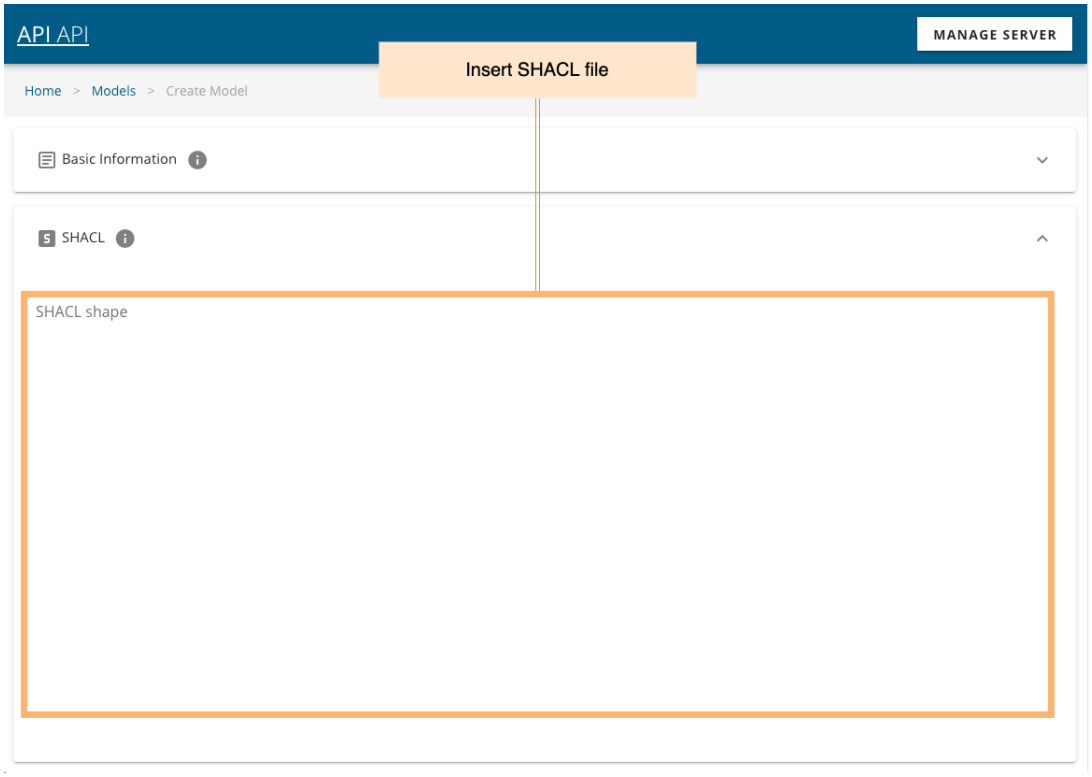


Figuur 5.1: The Api creation page with the Mapping tab unfolded

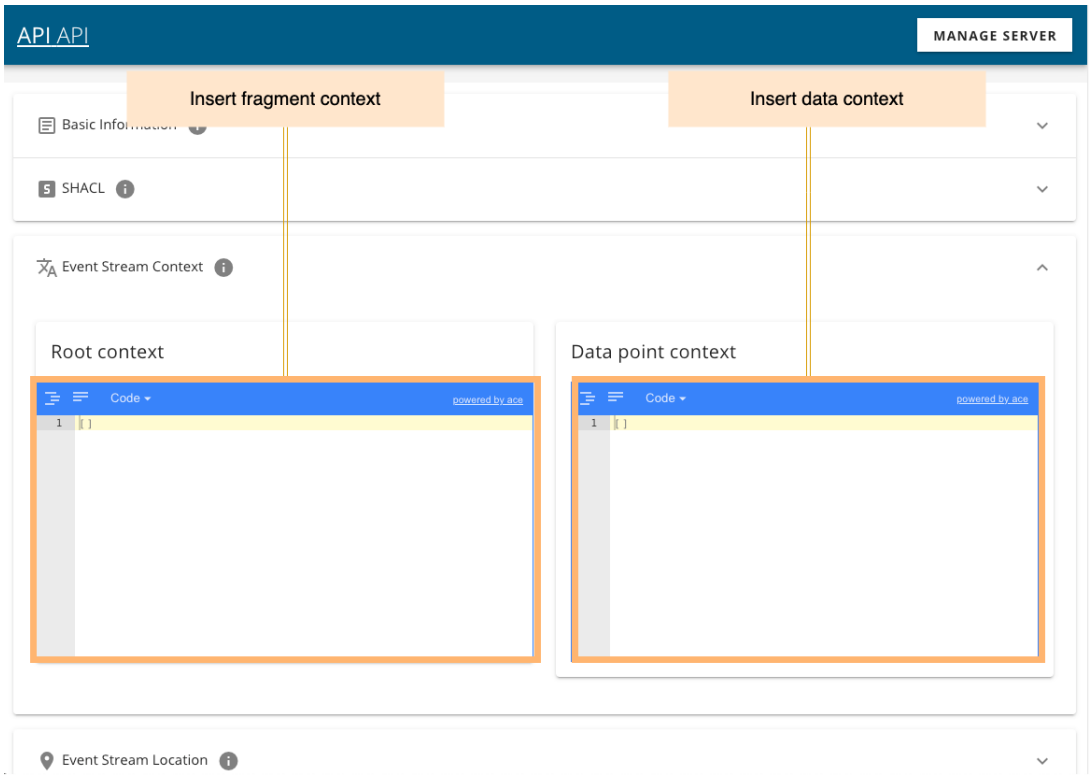
5.2 Creating a Model



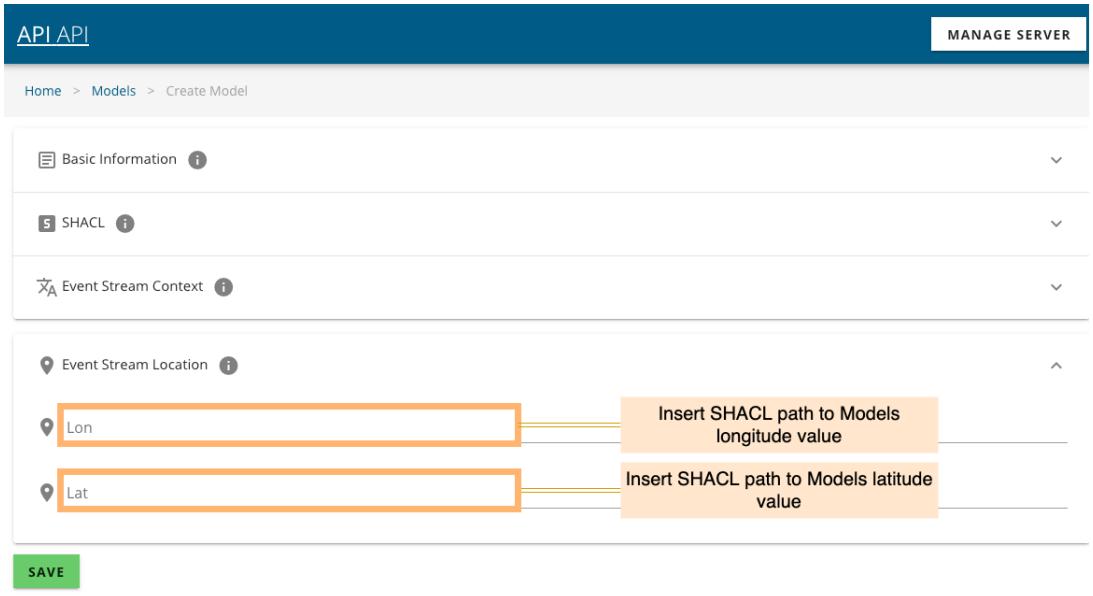
Figuur 5.2: The Api creation page with the Mapping tab unfolded



Figuur 5.3: The Api creation page with the Mapping tab unfolded

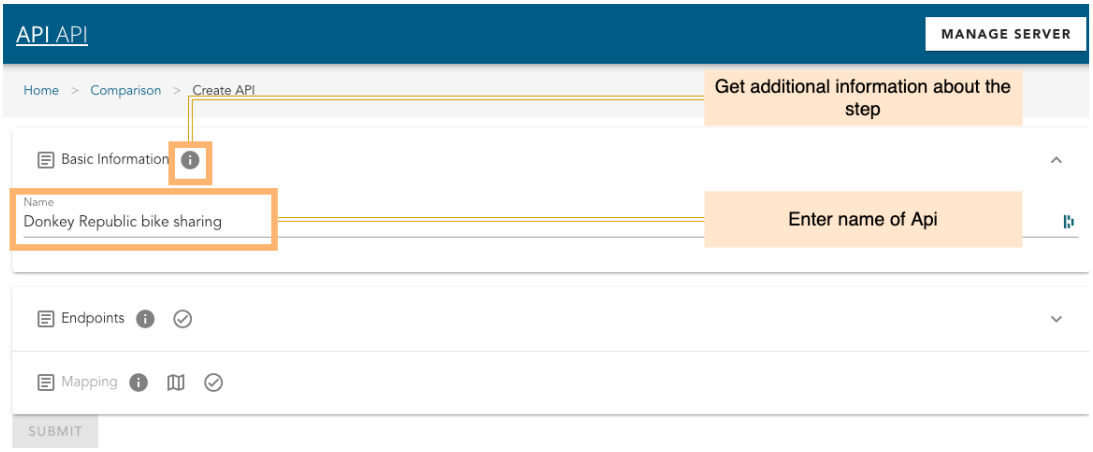


Figuur 5.4: The Api creation page with the Mapping tab unfolded

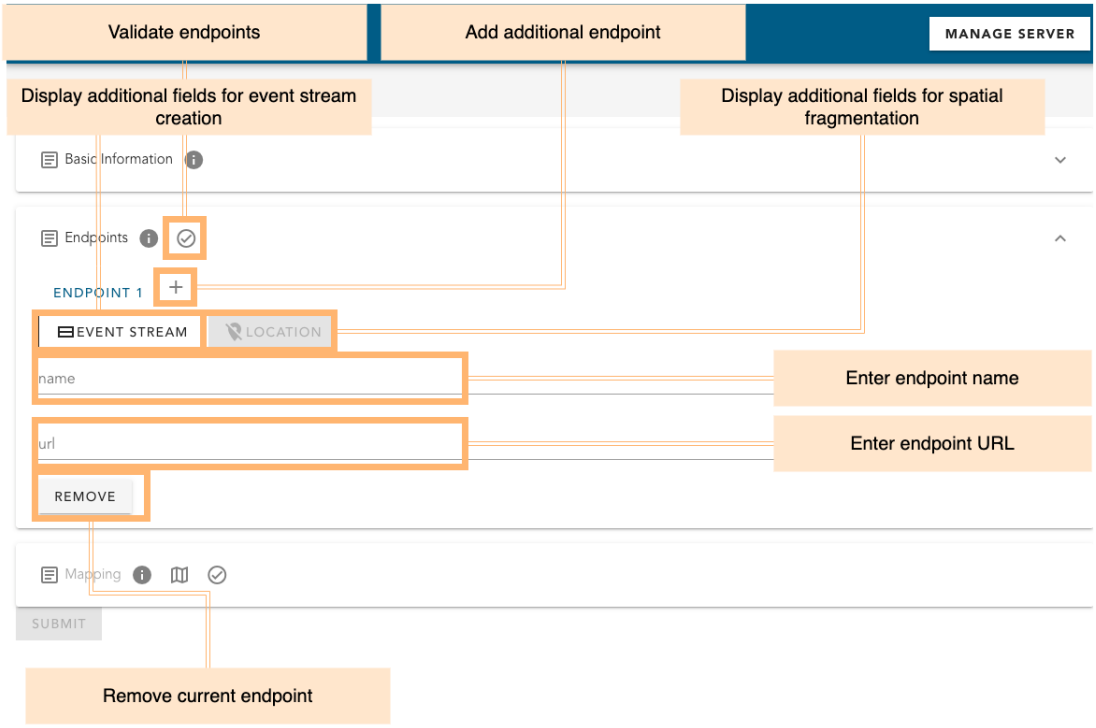


Figuur 5.5: The Api creation page with the Mapping tab unfolded

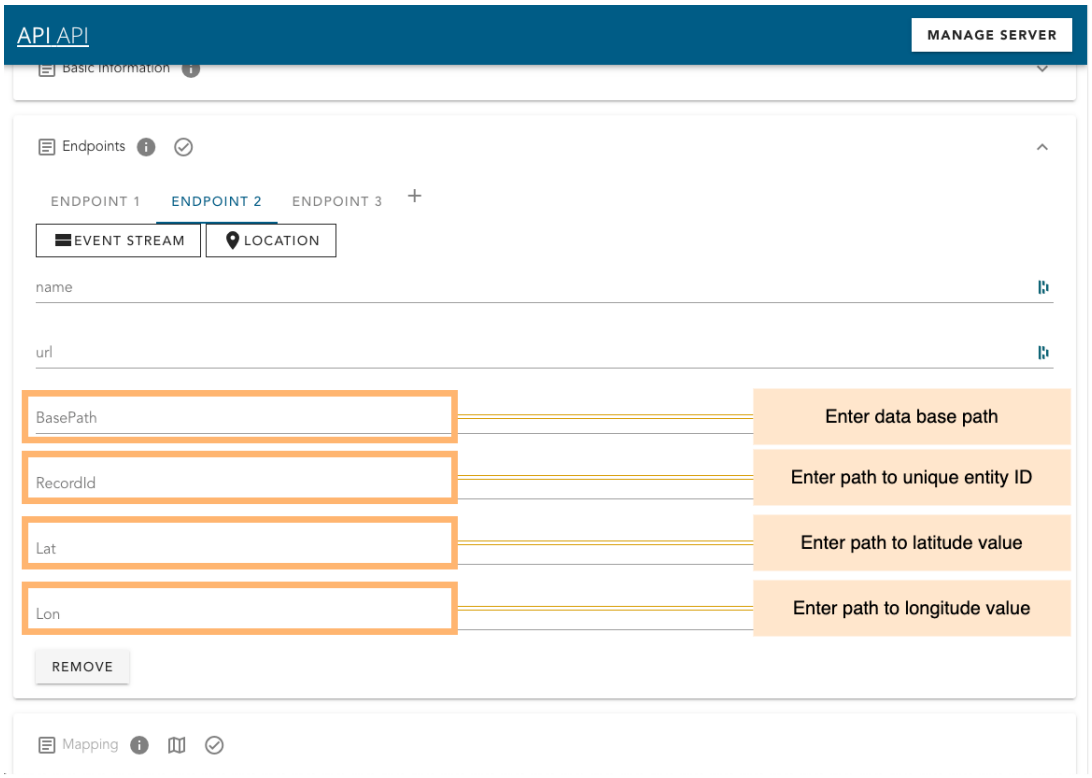
5.3 Creating an API



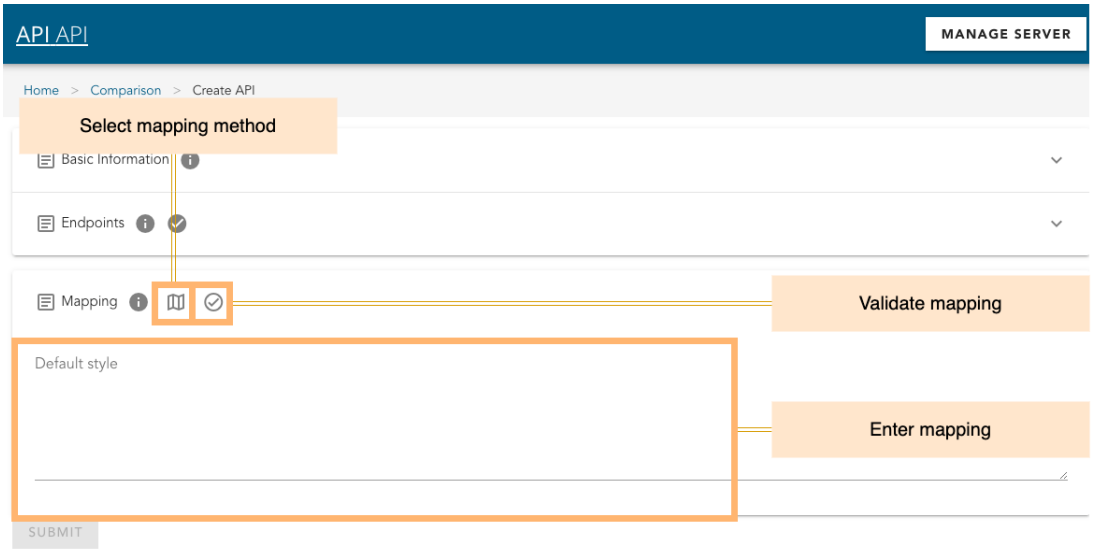
Figuur 5.6: The Api creation page with the Basic Information tab unfolded



Figuur 5.7: The Api creation page with the Endpoints tab unfolded, frame 1



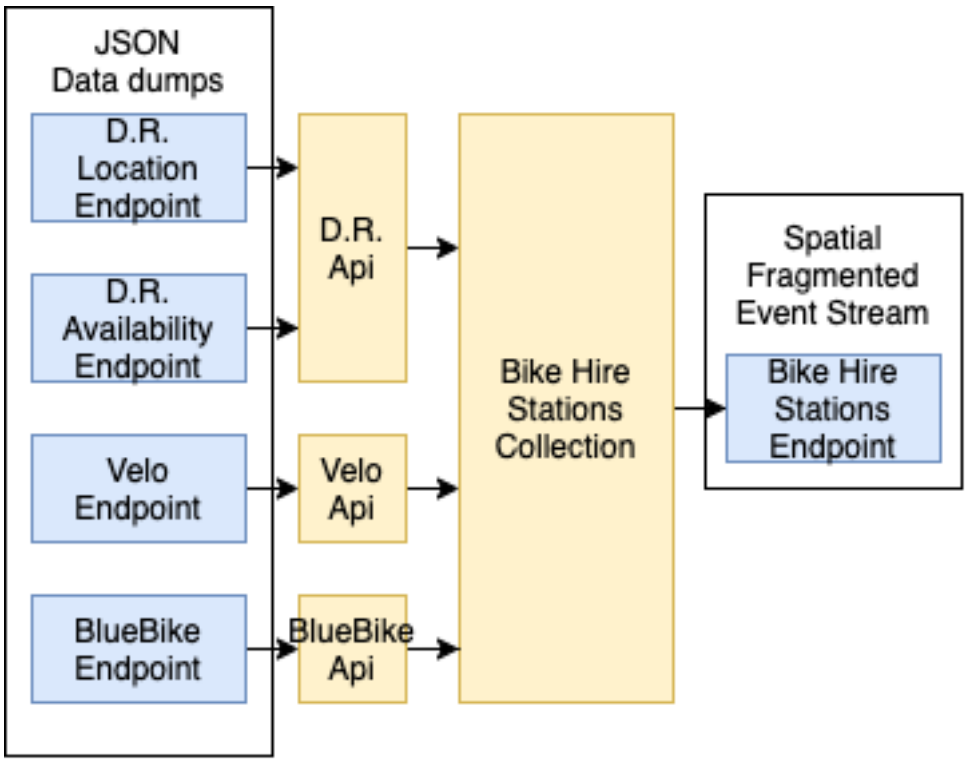
Figuur 5.8: The Api creation page with the Endpoints tab unfolded, frame 2



Figuur 5.9: The Api creation page with the Mapping tab unfolded

Appendix I: Demonstration set up and code

This Appendix covers the set up, SHACL shape and RML mappings used for creating a demonstration of the proposed framework. This demonstration will create a single spatial fragmented event stream from four bike hire station endpoints. Two of these endpoints belong to Donkey Republic bike hire stations, one to BlueBike bike hire stations and the last one to the Velo bike hire stations. These endpoint will mapped to the NGSI bike hire docking station data model. Compliance with this model will be enforced by multiple SHACL shapes. The set up is visualized in figure 5.10. Recreating the demonstration can be done by first creating a Model with the SHACL file provided in code block 5.2. A Collection should then be created using this Model. Three APIS are added to this Collection being one for the Donkey Republic endpoints, one for the Velo endpoint and one for the BlueBike endpoint. For each of these APIS both event stream as location information should be provided. Mapping is done using RML via text. The three RML mappings are provided in code blocks 5.3, 5.4 and 5.5.



Figuur 5.10: Demonstration set up

Listing 5.2: Partial SHACL model for the NGSI Bike Hire Docking Station model

```
@prefix dash: <http://datashapes.org/dash#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix schema: <http://schema.org/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ngsi: <https://uri.fiware.org/ns/data-models#> .
@prefix sdm:
    <https://smart-data-models.github.io/data-models/terms.jsonld#/definitions/> .
@prefix ngsic: <https://uri.etsi.org/ngsi-ld/> .
@prefix ex: <http://example.com/> .
@prefix rl: <http://example.org/rules/>.
@prefix geojson: <https://purl.org/geojson/vocab#> .

ngsi:BikeHireDockingStationShape
    a sh:NodeShape ;
    sh:targetClass ngsi:BikeHireDockingStation ;
    sh:property [
        sh:path sdm:availableBikeNumber ;
        sh:class ngsic:Property ;
```

```
        sh:minCount 1 ;
        sh:description "The number of bikes available in the bike hire docking station
            to be hired by users.";
        sh:nodeKind sh:BlankNode ;
        sh:property [
            sh:path ngsic:hasValue ;
            sh:minCount 1 ;
        ] ;
    ] ;
    sh:property [
        sh:path ngsic:name ;
        sh:class ngsic:Property ;
        sh:minCount 1 ;
        sh:description "Name given to the docking station.";
        sh:nodeKind sh:BlankNode ;
        sh:property [
            sh:path ngsic:hasValue ;
            sh:minCount 1 ;
        ] ;
    ] ;
    sh:property [
        sh:path sdm:totalSlotNumber ;
        sh:class ngsic:Property ;
        sh:description "The total number of slots offered by this bike docking
            station.";
        sh:nodeKind sh:BlankNode ;
        sh:property [
            sh:path ngsic:hasValue ;
            sh:minCount 1 ;
        ] ;
    ] ;
    sh:property [
        sh:path ngsic:location ;
        sh:class ngsic:GeoProperty ;
        sh:minCount 1 ;
        sh:description "Geolocation of the station represented by a GeoJSON
            (Multi)Polygon or Point.";
        sh:nodeKind sh:BlankNode ;
        sh:property [
            sh:path ngsic:hasValue ;
            sh:minCount 1 ;
            sh:nodeKind sh:BlankNode ;
            sh:class geojson:Point ;
            sh:property [
```

```
            sh:path ngsic:coordinates ;
            sh:minCount 1 ;
        ] ;
    ] ;
] ;
sh:property [
    sh:path schema:address ;
    sh:class ngsic:Property ;
    sh:description "Registered docking station site civic address.";
    sh:nodeKind sh:BlankNode ;
    sh:property [
        sh:path ngsic:hasValue ;
        sh:minCount 1 ;
    ] ;
] ;
sh:property [
    sh:path sdm:freeSlotNumber ;
    sh:class ngsic:Property ;
    sh:description "The number of slots available for returning and parking bikes.
        It must lower or equal than totalSlotNumber.";
    sh:nodeKind sh:BlankNode ;
    sh:property [
        sh:path ngsic:hasValue ;
        sh:minCount 1 ;
    ] ;
] .
```

Listing 5.3: Demonstration RML for mapping the Velo endpoint to the NGSI Bike Hire Docking Station model

```
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rl: <http://example.org/rules/>.
@prefix ngsic: <https://uri.etsi.org/ngsi-ld/> .
@prefix geojson: <https://purl.org/geojson/vocab#> .
@prefix ngsi: <https://uri.fiware.org/ns/data-models#> .
@prefix sdm:
    <https://smart-data-models.github.io/data-models/terms.jsonld#/definitions/> .
@prefix schema: <https://schema.org/> .

rl:BikeHireDockingStation a rr:TriplesMap;
  rml:logicalSource [
```

```
  rml:source "Endpoint 1";
  rml:referenceFormulation ql:JSONPath;
  rml:iterator "*"
];
rr:subjectMap [
  rr:template "https://www.velo-antwerpen.be/{id}"
];
rr:predicateObjectMap [
  rr:predicate rdf:type;
  rr:objectMap [
    rr:constant ngsi:BikeHireDockingStation
  ]
];
rr:predicateObjectMap [
  rr:predicate ngsic:name;
  rr:objectMap [
    rr:parentTriplesMap rl:name;
    rr:joinCondition [
      rr:child "id";
      rr:parent "id"
    ]
  ]
];
rr:predicateObjectMap [
  rr:predicate ngsic:location;
  rr:objectMap [
    rr:parentTriplesMap rl:GeoJson;
    rr:joinCondition [
      rr:child "id";
      rr:parent "id"
    ]
  ]
];
rr:predicateObjectMap [
  rr:predicate schema:address;
  rr:objectMap [
    rr:parentTriplesMap rl:address;
    rr:joinCondition [
      rr:child "id";
      rr:parent "id"
    ]
  ]
];
rr:predicateObjectMap [
```

```
    rr:predicate sdm:availableBikeNumber;
    rr:objectMap [
      rr:parentTriplesMap rl:availableBikeNumber;
      rr:joinCondition [
        rr:child "id";
        rr:parent "id"
      ]
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate schema:areaServed;
    rr:objectMap [
      rr:parentTriplesMap rl:areaServed;
      rr:joinCondition [
        rr:child "id";
        rr:parent "id"
      ]
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate sdm:freeSlotNumber;
    rr:objectMap [
      rr:parentTriplesMap rl:freeSlotNumber;
      rr:joinCondition [
        rr:child "id";
        rr:parent "id"
      ]
    ]
  ].

rl:GeoJson a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "*"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:GeoProperty
    ]
```

```
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rr:parentTriplesMap rl:Point;
      rr:joinCondition [
        rr:child "id";
        rr:parent "id"
      ]
    ]
  ].

rl:Point a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "*"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant geojson:Point
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:coordinates;
    rr:objectMap [
      rml:reference "location.value.coordinates[0]"
    ];
    rr:objectMap [
      rml:reference "location.value.coordinates[1]"
    ]
  ].

rl:address a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "*"
  ];
  rr:subjectMap [
```

```
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:Property
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rr:parentTriplesMap rl:addressvalue;
      rr:joinCondition [
        rr:child "id";
        rr:parent "id"
      ]
    ]
  ].

rl:addressvalue a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "*"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant schema:PostalAddress;
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate schema:streetAddress;
    rr:objectMap [
      rml:reference "address.value.streetAddress"
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate schema:addressCountry;
    rr:objectMap [
      rml:reference "address.value.addressCountry"
```

```
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate schema:postalCode;
    rr:objectMap [
      rml:reference "address.value.postalCode"
    ]
  ].

rl:name a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "*"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:Property
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rml:reference "name.value"
    ]
  ].

rl:availableBikeNumber a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "*"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:Property
```

```
  ]
];
rr:predicateObjectMap [
  rr:predicate ngsic:hasValue;
  rr:objectMap [
    rml:reference "availableBikeNumber.value"
  ]
].

rl:freeSlotNumber a rr:TriplesMap;
rml:logicalSource [
  rml:source "Endpoint 1";
  rml:referenceFormulation ql:JSONPath;
  rml:iterator "*"
];
rr:subjectMap [
  rr:termType rr:BlankNode
];
rr:predicateObjectMap [
  rr:predicate rdf:type;
  rr:objectMap [
    rr:constant ngsic:Property
  ]
];
rr:predicateObjectMap [
  rr:predicate ngsic:hasValue;
  rr:objectMap [
    rml:reference "freeSlotNumber.value"
  ]
].

rl:areaServed a rr:TriplesMap;
rml:logicalSource [
  rml:source "Endpoint 1";
  rml:referenceFormulation ql:JSONPath;
  rml:iterator "*"
];
rr:subjectMap [
  rr:termType rr:BlankNode
];
rr:predicateObjectMap [
  rr:predicate rdf:type;
  rr:objectMap [
    rr:constant ngsic:Property
```

```
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rml:reference "areaServed.value"
    ]
  ].
```

Listing 5.4: Demonstration RML for mapping the BlueBike endpoint to the NGSI Bike Hire Docking Station model

```
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rl: <http://example.org/rules/>.
@prefix ngsic: <https://uri.etsi.org/ngsi-ld/> .
@prefix geojson: <https://purl.org/geojson/vocab#> .
@prefix ngsi: <https://uri.fiware.org/ns/data-models#> .
@prefix sdm:
    <https://smart-data-models.github.io/data-models/terms.jsonld#/definitions/> .
@prefix schema: <https://schema.org/> .

rl:BikeHireDockingStation a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.features[*]"
  ];
  rr:subjectMap [
    rr:template "https://www.velo-antwerpen.be/{properties.@id}"
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsi:BikeHireDockingStation
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:name;
    rr:objectMap [
      rr:parentTriplesMap rl:name;
      rr:joinCondition [
```

```
      rr:child "properties.@id";
      rr:parent "properties.@id"
    ]
  ]
];
rr:predicateObjectMap [
  rr:predicate ngsic:location;
  rr:objectMap [
    rr:parentTriplesMap rl:GeoJson;
    rr:joinCondition [
      rr:child "properties.@id";
      rr:parent "properties.@id"
    ]
  ]
];
rr:predicateObjectMap [
  rr:predicate sdm:availableBikeNumber;
  rr:objectMap [
    rr:parentTriplesMap rl:availableBikeNumber;
    rr:joinCondition [
      rr:child "properties.@id";
      rr:parent "properties.@id"
    ]
  ]
];
rr:predicateObjectMap [
  rr:predicate sdm:totalSlotNumber;
  rr:objectMap [
    rr:parentTriplesMap rl:totalSlotNumber;
    rr:joinCondition [
      rr:child "properties.@id";
      rr:parent "properties.@id"
    ]
  ]
];
rr:predicateObjectMap [
  rr:predicate sdm:freeSlotNumber;
  rr:objectMap [
    rr:parentTriplesMap rl:freeSlotNumber;
    rr:joinCondition [
      rr:child "properties.@id";
      rr:parent "properties.@id"
    ]
  ]
```

```
  ].

rl:GeoJson a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.features[*]"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:GeoProperty
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rr:parentTriplesMap rl:Point;
      rr:joinCondition [
        rr:child "properties.@id";
        rr:parent "properties.@id"
      ]
    ]
  ].

rl:Point a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.features[*]"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant geojson:Point
    ]
  ];
  rr:predicateObjectMap [
```

```
    rr:predicate ngsic:coordinates;
    rr:objectMap [
      rml:reference "properties.latitude"
    ];
    rr:objectMap [
      rml:reference "properties.longitude"
    ]
  ].

rl:availableBikeNumber a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.features[*]"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:Property
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rml:reference "properties.bikes_available"
    ]
  ].

rl:name a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.features[*]"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:Property
```

```
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rml:reference "properties.name"
    ]
  ].

rl:freeSlotNumber a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.features[*]"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:Property
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rml:reference "properties.docks_available"
    ]
  ].

rl:totalSlotNumber a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.features[*]"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:Property
```

```
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rml:reference "properties.capacity"
    ]
  ].
```

Listing 5.5: Demonstration RML for mapping the DonkeyRepublic endpoint to the NGSI Bike Hire Docking Station model

```
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rl: <http://example.org/rules/>.
@prefix ngsic: <https://uri.etsi.org/ngsi-ld/> .
@prefix geojson: <https://purl.org/geojson/vocab#> .
@prefix ngsi: <https://uri.fiware.org/ns/data-models#> .
@prefix sdm:
    <https://smart-data-models.github.io/data-models/terms.jsonld#/definitions/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

rl:BikeHireDockingStation a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.records[*]"
  ];
  rr:subjectMap [
    rr:template "https://www.donkey.bike/{fields.station_id}"
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsi:BikeHireDockingStation
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:name;
    rr:objectMap [
      rr:parentTriplesMap rl:name;
      rr:joinCondition [
```

```
      rr:child "fields.station_id";
      rr:parent "fields.station_id"
    ]
  ]
 ];
 rr:predicateObjectMap [
  rr:predicate ngsic:location;
  rr:objectMap [
   rr:parentTriplesMap rl:GeoJson;
   rr:joinCondition [
    rr:child "fields.lon";
    rr:parent "lon"
   ];
   rr:joinCondition [
    rr:child "fields.lat";
    rr:parent "lat"
   ]
  ]
 ].

rl:name a rr:TriplesMap;
 rml:logicalSource [
  rml:source "Endpoint 1";
  rml:referenceFormulation ql:JSONPath;
  rml:iterator "$.records[*]"
 ];
 rr:subjectMap [
  rr:termType rr:BlankNode
 ];
 rr:predicateObjectMap [
  rr:predicate rdf:type;
  rr:objectMap [
   rr:constant ngsic:Property
  ]
 ];
 rr:predicateObjectMap [
  rr:predicate ngsic:hasValue;
  rr:objectMap [
   rml:reference "fields.name"
  ]
 ].

rl:GeoJson a rr:TriplesMap;
 rml:logicalSource [
```

```
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.records[*].fields"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:GeoProperty
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rr:parentTriplesMap rl:Point;
      rr:joinCondition [
        rr:child "lon";
        rr:parent "lon"
      ];
      rr:joinCondition [
        rr:child "lat";
        rr:parent "lat"
      ]
    ]
  ].

rl:Point a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 1";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.records[*].fields"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant geojson:Point
    ]
  ];
  rr:predicateObjectMap [
```

```
    rr:predicate ngsic:coordinates;
    rr:objectMap [
      rml:reference "lat"
    ];
    rr:objectMap [
      rml:reference "lon"
    ]
  ].

rl:BikeHireDockingStation2 a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 2";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.records[*]"
  ];
  rr:subjectMap [
    rr:template "https://www.donkey.bike/{fields.station_id}"
  ];
  rr:predicateObjectMap [
    rr:predicate sdm:freeSlotNumber;
    rr:objectMap [
      rr:parentTriplesMap rl:freeSlotNumber;
      rr:joinCondition [
        rr:child "fields.station_id";
        rr:parent "fields.station_id"
      ]
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate sdm:availableBikeNumber;
    rr:objectMap [
      rr:parentTriplesMap rl:availableBikeNumber;
      rr:joinCondition [
        rr:child "fields.station_id";
        rr:parent "fields.station_id"
      ]
    ]
  ].

rl:availableBikeNumber a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 2";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.records[*]"
```

```
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:Property
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rml:reference "fields.num_bikes_available"
    ]
  ].

rl:freeSlotNumber a rr:TriplesMap;
  rml:logicalSource [
    rml:source "Endpoint 2";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.records[*]"
  ];
  rr:subjectMap [
    rr:termType rr:BlankNode
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:Property
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:hasValue;
    rr:objectMap [
      rml:reference "fields.num_docks_available"
    ]
  ].
```