Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Lijst van listings

1 Introduction

Owner collects and maps JSON to **JSON** and a **data dump** is provided
Anyone can suggest and map JSON to **JSON LD** and **cacheable endpoint** is provided

1.1 ApiApi Current

ApiApi is a Api aggregation framework develop by Digipolis for the city of Ghent. The goal of ApiApi is twofold. First off 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

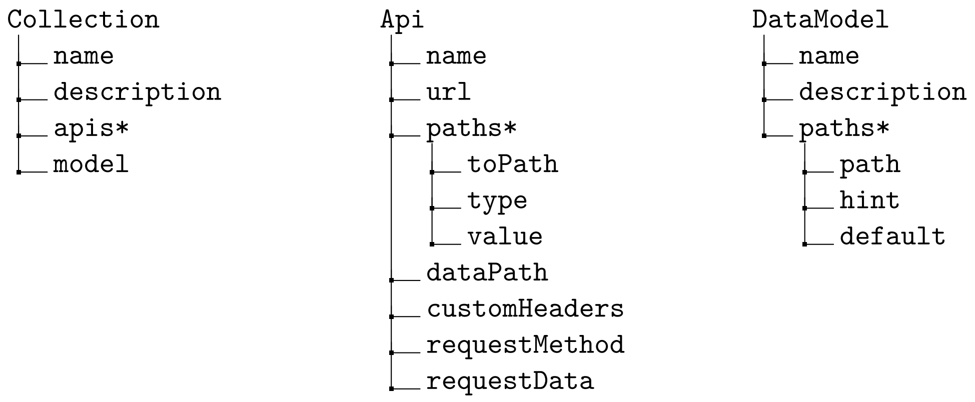
The logical ApiApi structure is built upon three concepts: a collection, an api and a model. An **Api** represents a single endpoint which serves data. A **Model** represents the data model the data coming from an Api should be mapped to. Models are created in the Model Creator . An **Api** represents a single endpoint. A **Collection** represents a set of Apis, each which have to be mapped to the same Model. When adding an Api to a Collection, a mapping should be created which transforms the data coming from the endpoint into the Collections Model. Each Collection has a single endpoint which combines the data from all the Apis in the Collection each mapped to the Collections Model. As a whole, ApiApi thus creates a single endpoints which holds data from all the Collection Apis each data point represented as an instance of the Collections Model.

add ref

screenshots a
overview

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 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 and displayed on the landing screen. The *Details* button 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 (under the collection name). From here Apis can be deleted from the Collection (which removes the Api entirely) by pressing the *Delete* button. 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 Collections Model. Finally there is a redirect the server settings by pressing the *Manage Server* button on the top right on every screen. Currently only a flush cache setting is present which will delete the local Api cache.

Technically, ApiApi uses the Nuxt.js Vue.js framework with Babel compiling. For storage a mongoDB database is used for persistent data and Redis store for the endpoint data cache. The MongoDB database is interacted with using Mongoose which is a Node framework which handles the interaction. For quick styling and component construction it uses the Vuetify UI Library. All programming is done in Javascript and ran using Node.js. While this technical implementation do not really impact the conceptual best practice research, it should be noted that it does limit the implementation scope to compatible technologies. Fortunately most technologies have implementations in many programming languages including Javascript and Node. A short overview of these technologies is presented below.



Figuur 1.1: ApiApi MongoDB models. An * represents an Array.

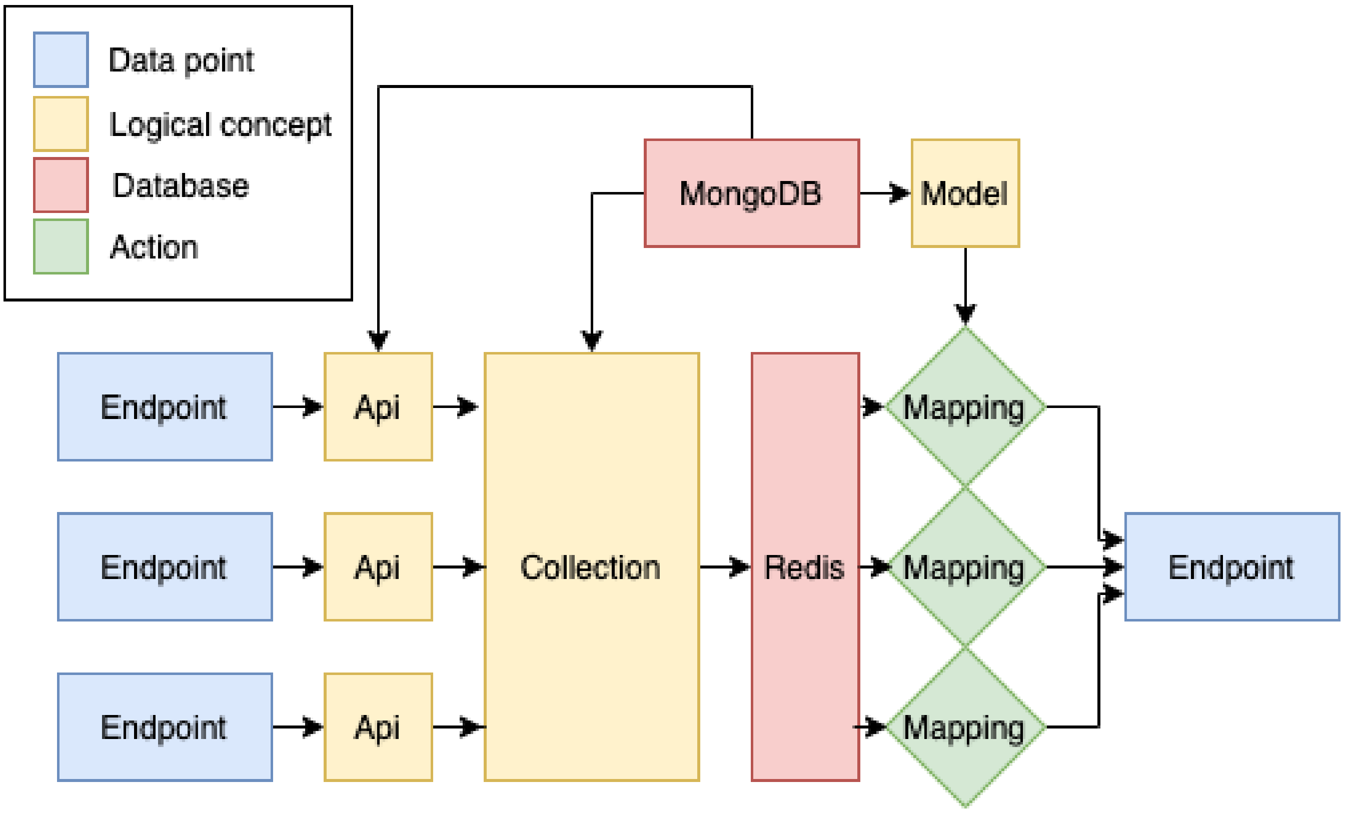
Vue.js is an open-source Node.js framework for dynamic web application development. For GUI development Vue.js uses a Component structure, where each Component can be logically seen as a reusable piece of GUI with the possibility to have its own logic. Components are then combined in higher level Components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

The **Nuxt.js** 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.

MongoDB 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**. Mongoose is a wrapper for Node.js that handles some database interactions such as modeling in a easy to use fashion. ApiApi has a model for each entity as described above: Collections, Apis and Models.

Finally **Redis** is used as cache database for the endpoint data. Redis is an open source in database structure 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



Figuur 1.2: The structure of ApiApi

happen after every request.

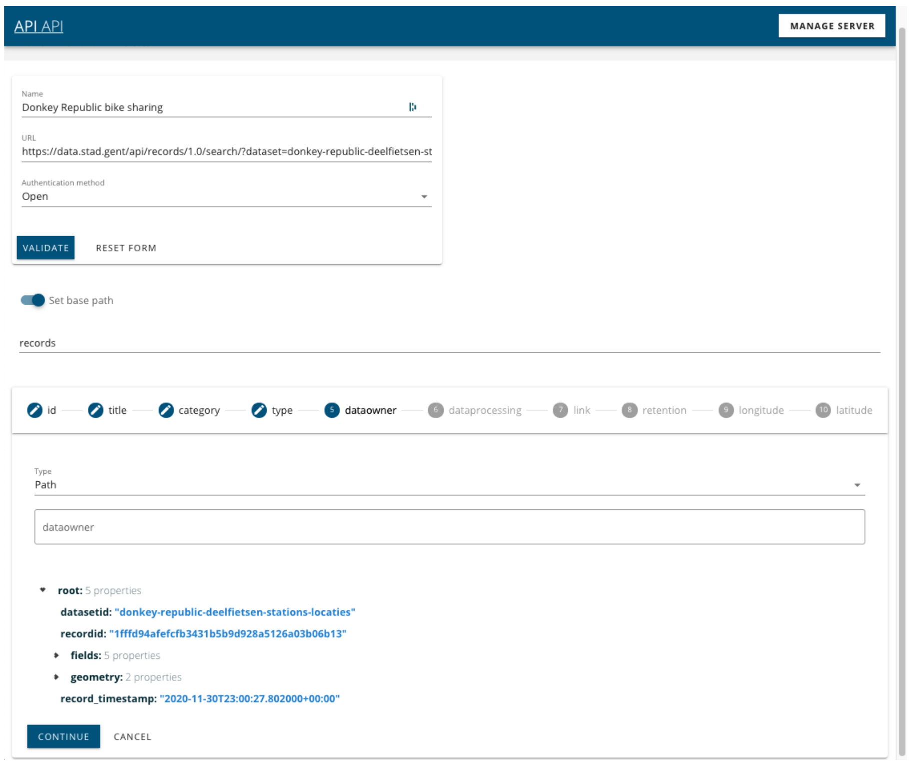
write about
point

1.1.2 The Api Mapper

Api creation is done using ApiApis mapper tool. This deserves special attention since it is one of the main features and most complex component. 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. The partly free to chose since the cache stores endpoint data per name, and duplicate names for different endpoints will result in erroneously cache hits. As authentication method ApiApi supports either open endpoints, closed with api key or the use of custom headers. When selecting either but open a menu drops down where either the api key or the custom headers can be inserted. There is support for any number of custom headers. This basic api information header is concluding by pressing validate, which contacts the endpoint on its given URL and 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 required data model. The mappers header contains the possibility to set a base path different from the root. This base path will then 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 data in the original data point 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. In the pop up the creation can either be 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.



Name
Donkey Republic bike sharing

URL
https://data.stad.gent/api/records/1.0/search/?dataset=donkey-republic-deelfietsen-st

Authentication method
Open

Set base path

records

Type
Path

dataowner

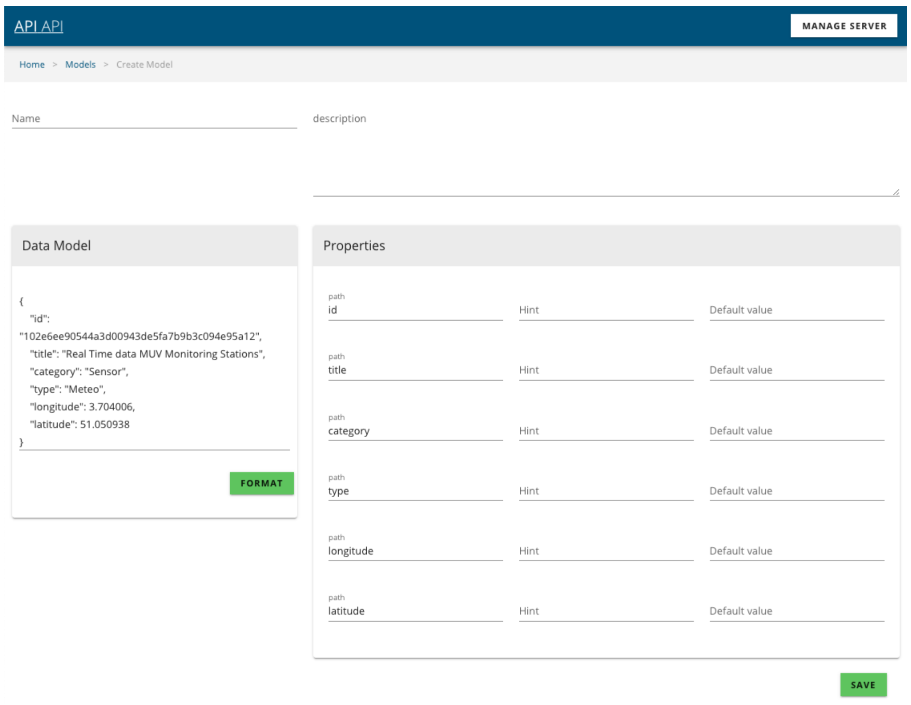
```
root: 5 properties
  datasetid: "donkey-republic-deelfietsen-stations-locaties"
  recordid: "1fffd94afefcfb3431b5b9d928a5126a03b06b13"
  fields: 5 properties
  geometry: 2 properties
  record_timestamp: "2020-11-30T23:00:27.802000+00:00"
```

Figuur 1.3: The top view of the mapper which defines the endpoint to use

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 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 as stated type checking and additional restrictions are not supported. However 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



```
{
   "id":
"102e6ee90544a3d00943de5fa7b9b3c094e95a12",
   "title": "Real Time data MUV Monitoring Stations",
   "category": "Sensor",
   "type": "Meteo",
   "longitude": 3.704006,
   "latitude": 51.050938
}
```

path	Hint	Default value
id		
title		
category		
type		
longitude		
latitude		

Figuur 1.4: The model creation page

a required field. This means that MongoDB will not accept entries with empty fields and the mapping will be rejected. This way a single implicit validation rule is created: each property should have exactly one value.

1.2 Linked Data

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. This user story needs little explanation, the benefits of linked data are described above . 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 a system administrator, i can semantically map my endpoint to an approved 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 a system administrator, i can receive feedback about the compliance of my mapping* is created. For mapping form an endpoint to a approved data model, the possibility to create such endpoint should exist. ApiApi already has this possibility, but this work will look at a way to generalize this tool and make use of linked data specifications.

add ref

add ref

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. These fragments should be cacheable so that the can be reused. Size of fragments should be controllable such that size optimizations can be made. *As a user, i can receive a cacheble response* and *As a system administrator, i can control the size of a fragment.* Finally the possibility of enriching the data by a time component will be looked at. This dimension is omitted in the current design where only the latest status of each data point is given. However, for historical data serves many purposes. For this reason: *As a user, i can receive a historical response* is added.

Tabel 1.1: User stories

Number	User Story	Requirements
1.1.1	As a user, i can receive a cacheble response	
1.1.2	As a user, i can receive a semantically annotated response	
1.1.3	As a user, i can receive a historical response	
1.1.4	As a system administrator, i can create allowed data models	
1.1.5	As a system administrator, i can provide restrictions for those data models	
1.1.6	As a system administrator, i can semantically map an endpoint to a data model	
1.1.7	As a system administrator, i can receive feedback about the compliance of my mapping	
1.1.8	As a system administrator, i can control the size of a fragment	

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

research questions · · ·

Which linked data technologies are currently state of the art to implement these user stories into ApiApi?

2 Related Work

2.1 Mapping Languages

2.1.1 R2RML and variants

R2RML (RDB to RDF Mapping Language)[1] 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 can be found in [1]. Since this thesis uses JSON endpoints as data source, only extensions that allow this are useful for this cause. The comparison found in [1] lists RML and YARRRML as possible R2RML extensions with JSON support.

2.1.2 RML

RML (RDF Mapping Language)[2] 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.

[1]https://www.w3.org/TR/r2rml

[2]https://rml.io/specs/rml/

Besides this, RML follows the same syntax as R2RML.

A full specification of RML can be found on on its website2. This work will limit itself by describing a proof of concept mapping which generates ngsi Bike Hire Docking Station RDF from Donkey Republic JSON data. The mapping can be found in Appendix I6.3. This proof of concept maps the id, type, name and location of Donkey Republic records to their ngsi counterpart. An RML documents starts by defining prefixes. Beneath this the triples are modeled. rr:TriplesMap describes triples with the same subject. A rml:logicalSource triple defines the source (rml:source), data type (rml:referenceFormulation). In the case of a JSON file, an iterator (rml:iterator) should be present. This iterator defines the root of the mapping. In this case, the property records holds a JSON list with the datapoints, so $.records[*] provided. This subject is defined as an rr:subjectMap. In this case, a rr:template is used to create an URI from the recordid from each JSON list entry. After this the different predicate-object relations of this subject are modeled. This is done using a rr:predicateObjectMap. As mentioned, this proof of concept maps type, name and location. The mapping of type and name are similar. The corresponding predicate (rr:predicate) and object (rr:object) are defined. These differ in that type is the same for each entry and thus is a constant (rr:constant), in this case a ngsi:bikeHireDockingStation. The name however is variable over the entries, which is represented by a reference (rml:reference). In the case of a json file, this reference is simply the json path of the value starting from the defined root. The mapping of ngsi:location requires more care since it is not a simple but complex object. That is, each location is a GeoProperty, and each GeoProperty holds the type of geometric structure. In this case that is a Point. This differs from previous predicateObjectMaps since it depth is two. To map such complex structure, R2RML provides the option to create a parent-child relation between the the base object (the ngsi:bikeHireDockingStation) and its child (the ngsic:GeoProperty). A new rr:TriplesMap should be created for the child, linked to the parent. To link two tripleMaps, a rr:parentTripleMap is used. Optionally a rr:joinCondition can be provided, which only links the triples if it holds true. In this case the triples are correlated by location which is represented by the lan and lon fields. Note that the parent and child do not share the same root, so the reference to these fields is different. In a similar fashion the ngsic:GeoProperty is linked to its child, a goejson:Point.

ARRRML
g

2.1.3 YARRRML

YARRRML[3] is a serialization of 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

[3]https://rml.io/yarrrml/spec/

accessible. Hence, currently the only way to use YARRRML for generation of RDF data is by inserting a YARRRML to RML translation step into the pipeline and using an RML generator. No direct YARRRML generation has been implemented.

A full specification of YARRRML can be found on the RML website3. As with RML, this work will limit itself to describing a proof of concept mapping which maps a Donkey Republic endpoint to the ngsi Bike Hire Docking Station data model.

```
prefixes:
  donkey: "https://www.donkey.bike/"
  ngsi: "https://uri.fiware.org/ns/data-models#"
  ngsic: "https://uri.etsi.org/ngsi-ld/"


mappings:
  person:
    sources:
      - ['data.json~jsonpath', '$.records[*]']
    s: donkey:$(recordid)
    po:
      - [a, ngsi:BikeHireDockingStation]
      - p: ngsic:location
        o:
        - value: $(fields.lat)
        - value: $(fields.lon)
      - [ngsic:name, $(fields.name), xsd:string]
```

Listing 2.1: An example YARRML mapping, which maps a Donkey Republic bike hire station endpoint to a part of the NGSI BikeHireStation specification

2.2 Constraint Languages

2.2.1 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, 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

[4]https://www.w3.org/TR/shacl/

certain id or class will be validated against that shapes graph.

A full specification of the SHACL language can be found on the W3 website4. An interesting sideshow to get quick started can be found here[5]. This work will limit itself to describing a proof of concept SHACL definition which validates a part of the ngsi Bike Hire Docking Station.

ver en gene-
mapping

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix ngsic: <https://uri.etsi.org/ngsi-ld/> .
@prefix ngsi: <https://uri.fiware.org/ns/data-models#> .

<https://example.com/BikeHireDockingStationShape> a sh:NodeShape;
  sh:closed false;
  sh:property [
    sh:description "Name given to the docking station.";
    sh:name "name";
    sh:datatype xsd:string ;
    sh:path ngsic:name
  ], [
    sh:class ngsic:GeoProperty;
    sh:description "Geolocation of the station represented by a GeoJSON (Multi)Polygon
        or Point.";
    sh:minCount 1;
    sh:name "location";
    sh:path ngsic:location
  ];
  sh:targetClass ngsic:BikeHireDockingStation> .
```

Listing 2.2: An example SHACL shape graph which validates a part of the NGSI Bike Hire Docking Station model

2.2.2 SHACL form generation

2.3 Event Streams

[5]https://www.slideshare.net/jelabra/shacl-by-example

3
This Work

3.1 The Mapper

3.1.1 Introduction

This chapter will cover the Mapping part of ApiApi. The ApiApi framework is built upon the principle of mapping simple data sources to predefined data models. A mapping method from the viewpoint of the user can be defined using 4 characteristics.

Tabel 3.1: Questions

Number	Question
3.1.1	How to define a mapping?
3.1.2	How to input a mapping?
3.1.3	How to define a model?
3.1.4	How to input a model?
3.1.5	How to validate a mapping?

Characteristic 3.1.1

Characteristic 3.1.1 refers to the specification used for defining the mapping information. This holds the information needed for the data transformation.

Characteristic 3.1.2

Characteristic 3.1.2 refers to the method in which the mapping in inputted.

Characteristic 3.1.3

Characteristic 3.1.3 refers to the specification used for defining a model to map to. This holds the information which is needed to know what model to map to.

Characteristic 3.1.4

Characteristic 3.1.4 refers to the method in which the model in inputted.

Characteristic 3.1.5

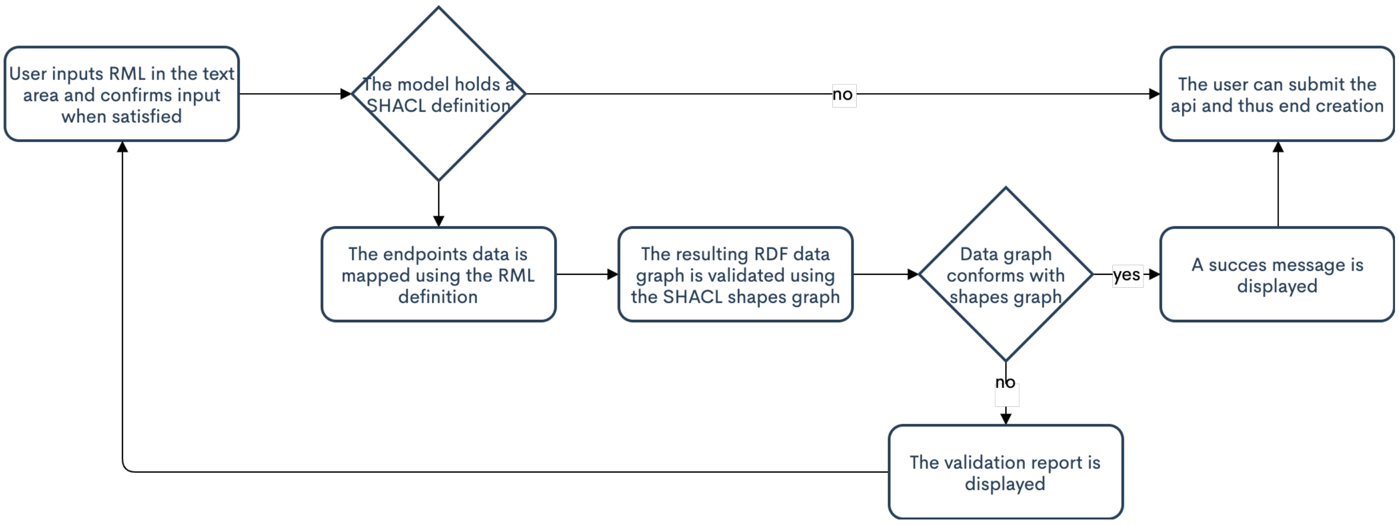
Characteristic 3.1.5 refers to the method used for validating if the mapping is in correspondence with the provided model.

3.1.2 RML Editor

The RML editor is implemented in the Api/create vue as a stand alone Vue.js component. When selecting the option RML, the component is displayed. The component itself is simply a textArea with a complete button. The user can insert his RML definition in the 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. The back hand handles this 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[1] is used. SHACL validation is done using TopQuadrant/shacl-js[2]. When the generated data graph complies with the shapes graph a success message is relayed to the user and the submit button is enable 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.

creenshot
litor

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


User inputs RML in the text area and confirms input when satisfied
The model holds a SHACL definition
no
The user can submit the api and thus end creation
The endpoints data is mapped using the RML definition
The resulting RDF data graph is validated using the SHACL shapes graph
Data graph conforms with shapes graph
yes
A succes message is displayed
no
The validation report is displayed

Figuur 3.1: The RML editors work flow

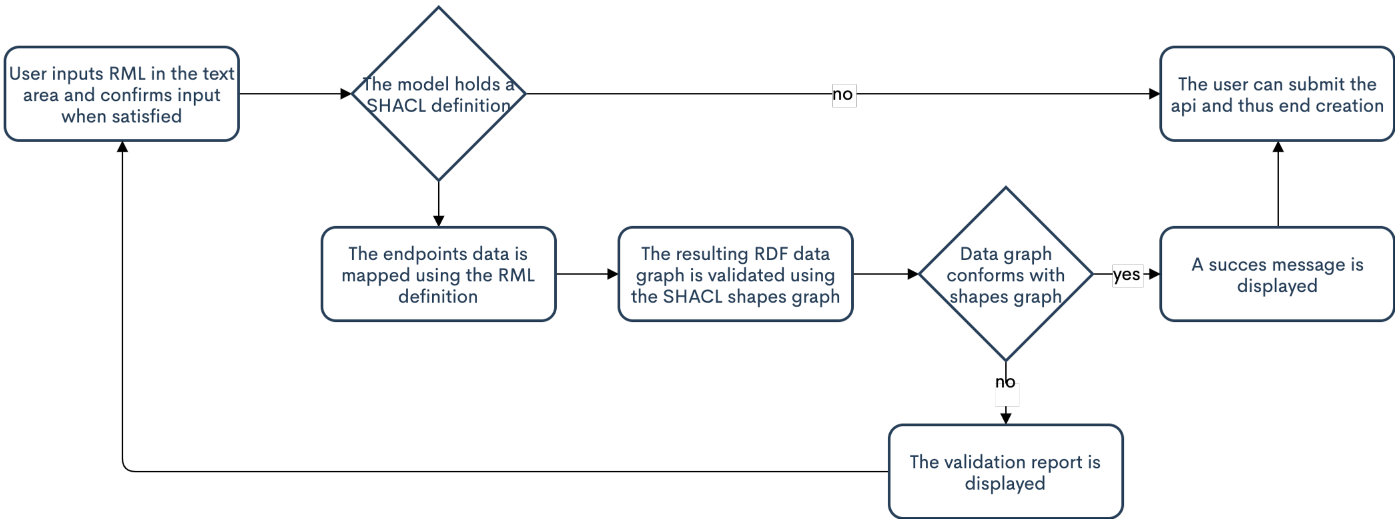
3.1.3 YARRRML Editor

The YARRRML editor is implemented in the Api/create vue as a stand alone Vue.js component. When selecting the option YARRRML, the component is displayed. 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 YARRRML is first converted to RML before mapping using the rmlio/yarrrml-parser footnotehttps://github.com/rmlio/yarrrml-parser. After generating the corresponding RML, the endpoints data is mapped to an RML data graph and validation using the models SHACL shapes graph. As with the RML editor, RMLio/rmlmapper-java-wrapper-js[1] and TopQuadrant/shacl-js[FN:shacl-js] 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 his 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.

insert screen
yarrrml map

3.1.4 SHACL Editor

The SHACL editor is implemented in the Model/create vue as a stand alone Vue.js component. When creating a model, the SHACL Editor is displayed. The editor itself is simply a text area with a submit button. Here the user can create his SHACL definition. When an api is created with this model, this SHACL definition will be used to validate that the mapping conforms.


User inputs RML in the text area and confirms input when satisfied
The model holds a SHACL definition
no
The user can submit the api and thus end creation
The endpoints data is mapped using the RML definition
The resulting RDF data graph is validated using the SHACL shapes graph
Data graph conforms with shapes graph
yes
A succes message is displayed
no
The validation report is displayed

Figuur 3.2: The YARRRML editors work flow

Additionally, this SHACL definition is used to construct the RML Mapper3.1.5 when selected in the Api/create component.

creenshot

3.1.5 RML Mapper

The RML Mapper is implemented in the Api/create vue as a stand alone Vue.js component. When selecting the option RML Mapper, the component is displayed. Unlike the previous described components, the RML Mapper is a more complex and layered component. The RML Mapper is structured is visualized in figure 3.4. Since this component is more complex is requires additional care. 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 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[3] 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

[3]information about the JSON path syntax can be found here: https://restfulapi.net/json-jsonpath/

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 consulted 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.

3.1.5.1 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[4] which is explained more in detail in section . Since this framework add ref

3.1.5.2 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. Technically, the choice makes sense since an RML mapper is already in place because of the RML Editor. So this service can be reused. Additionally, the original Babibubebon/vue-shacl-form[4] 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. The other RDF mapping tools added use RML as underlying specification. The RML Editor directly, and the YARRRML Editor by translation. So a knowledge of RML would suffice to understand the underlying principle of the entire mapping tool. In essence, the RML Mapper serve a similar goal to YARRRML in making an RML mapping more accessible. Because of this, and while not intractable, 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 Editor tool.

Before evaluating the implementation, it should be noted that the RML Mapper does only sup-

[4]https://github.com/Babibubebon/vue-shacl-form

port 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 in section . Additionally, for a more complete experience of the RML mapping functionality, the RML Editor or YARRRML Editor 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. As mentioned above, RML as description for this purpose. This narrows the question down to: How to link data field input to a or several RML triples. The RML mapping language is explored in section where a proof of concept mapping is analyzed. Since the current RML Mapper does not support nested objects, a simplification of this example will be evaluated$^{6.4}$. Compare this to a single data point which is the output of Babibubebon/vue-shacl-form[4] based on the SHACL shape graph6.5. Although different looking, 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 3.3. 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 object. The mapper must then generate a RML triple 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.

3.1.5.3 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[4] component, which it is based on. The ShaclMapper component is added as a top level component.

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 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

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 3.3: Comparison between the Babibubebon/vue-shacl-form[4] generated data point and an RML graph which maps to such data point.

```
ShaclMapperWrapper
└─ShaclMapper
   └─ShaclForm
      └─FormGroup
         ├─FormInput
         │  ├─TypedInput
         │  └─FormGroup [1]
         │     └─...
         ├─FormInput
         ├─FormInput
         └─...
```

Figuur 3.4: The component structure of the RML Mapper [1] Not ported functionality

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 definition. 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[4] 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 is 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.

3.1.6 Final Remarks

Tabel 3.2: Comparison of different mapping methods [1] Not intractable since generated by the form [2] Automatically generated based on the model

±	Q 3.1.1	Q 3.1.2	Q 3.1.3	Q 3.1.4	Q 3.1.5
ApiApi Mapper	Custom[1]	Form[2]	Custom[1]	Form or example	DB validation
RML	RML	Text	SHACL	Text	SHACL
YARRRML	YARRRML	Text	SHACL	Text	SHACL
RML Mapper	RML[1]	Form[2]	SHACL	Text	SHACL

improvement

RMLEditorx
vowl etc

4
Evaluation

5 Conclusion

Bibliografie

[1] 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

6 Appendix I: A list of listings

6.1 Donkey Republic endpoint response

```
{
    "nhits": 569,
    "parameters": {
        "dataset": "donkey-republic-deelfietsen-stations-locaties",
        "timezone": "UTC",
        "rows": 10,
        "start": 0,
        "format": "json"
    },
    "records": [
        {
            "datasetid": "donkey-republic-deelfietsen-stations-locaties",
            "recordid": "1fffd94afefcfb3431b5b9d928a5126a03b06b13",
            "fields": {
                "lat": 51.058262,
                "geopunt": [
                    51.058262,
                    3.7145533
                ],
                "lon": 3.7145533,
                "name": "Simon de Mirabellostraat 37",
                "station_id": "12419"
            },
            "geometry": {
                "type": "Point",
                "coordinates": [
                    3.7145533,
                    51.058262
                ]
            },
            "record_timestamp": "2020-10-31T23:01:08.411000+00:00"
        },
        ...
    ]
}
```

Listing 6.1: Data received from the Donkey Republic bike hire station endpoint with a single record

6.2 NGSI Bike Hire Docking Station model example

```
{
    "id": "urn:ngsi-ld:BikeHireDockingStation:Bcn-BikeHireDockingStation-1",
    "type": "BikeHireDockingStation",
    "status": {
        "type": "Property",
        "value": "working"
    },
    "availableBikeNumber": {
        "type": "Property",
        "value": 20,
        "observedAt": "2018-09-25T12:00:00Z"
    },
    "freeSlotNumber": {
        "type": "Property",
        "value": 10
    },
    "location": {
        "type": "GeoProperty",
        "value": {
            "type": "Point",
            "coordinates": [2.180042, 41.397952]
        }
    },
    "address": {
        "type": "Property",
        "value": {
            "addressCountry": "ES",
            "addressLocality": "Barcelona",
            "streetAddress": "Gran Via Corts Catalanes,760",
            "type": "PostalAddress"
        }
    },
    "@context": [
        "https://schema.lab.fiware.org/ld/context",
        "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
    ]
}
```

Listing 6.2: An example record compliant with the ngsi Bike Hire Docking Station data model. Provided as example on the data models page

6.3 RML mapping from Donkey Republic to ngsi Bike Hire Docking Station

```
@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#> .

rl:TriplesMap a rr:TriplesMap;
```

```
  rml:logicalSource [
    rml:source "data.json";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.records[*]"
  ];
  rr:subjectMap [
    rr:template "https://www.donkey.bike/{recordid}"
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [
      rr:constant ngsic:BikeHireDockingStation
    ]
  ];
  rr:predicateObjectMap [
    rr:predicate ngsic:name;
    rr:objectMap [
      rml:reference "fields.name"
    ]
  ];
  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:GeoJson a rr:TriplesMap;
  rml:logicalSource [
    rml:source "data.json";
    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 "data.json";
    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"
    ]
  ].
```

Listing 6.3: An example RML mapping, which maps a Donkey Republic bike hire station endpoint to a part of the NGSI BikeHireStation specification

6.4 Simplification of the Donkey Republic RML Mapping for the RML Mapper comparison

```
@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/> .

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

Listing 6.4: Simplification of the Donkey Republic to ngsi Bike Hire Docking Station Mapping[6.3]

6.5 Simplification of the SHACL proof of concept for the RML Mapper comparison

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix ngsic: <https://uri.etsi.org/ngsi-ld/> .
@prefix ngsi: <https://uri.fiware.org/ns/data-models#> .

<https://example.com/BikeHireDockingStationShape> a sh:NodeShape;
  sh:closed false;
  sh:property [
    sh:description "Name given to the docking station.";
    sh:name "name";
    sh:datatype xsd:string ;
    sh:path ngsic:name
  ];
  sh:targetClass ngsic:BikeHireDockingStation> .
```

Listing 6.5: A simplification of SHACL proof of concept[2.2] which models part of the ngsi Bike Hire Docking Station as using a shapes graph. Only the name is included.

6.6 Original Babibubebon vue shacl form output for the RML Mapper comparison

```
@prefix ngsic: <https://uri.etsi.org/ngsi-ld/> .
@prefix ngsi: <https://uri.etsi.org/ngsi-ld/>

<https://www.donkey.bike/5743>
  a ngsi:BikeHireDockingStation ;
  ngsic:name "Station Antwerpen-centraal"^^xsd:string .
```

Listing 6.6: An example of a Turtle serialized RDF data point generated by Babibubebon/vue-shacl-form[4]