

Inhoudsopgave

Lijst van figuren	3
Lijst van tabellen	4
1 Introduction	6
2 Mapping	8
2.1 Introduction	8
2.2 Current state	9
2.2.1 The Mapper	9
2.2.2 The Model	11
2.2.3 Back end and database	11
2.3 Research	12
2.3.1 Mapping Languages	13
2.3.2 Constraint Languages	15
2.3.3 GUI	16
2.4 Implementation	16
2.4.1 RML Editor	16
2.4.2 YARRRML Editor	17

2.4.3	SHACL Editor	17
2.4.4	RML Mapper	18
2.5	Final Remarks	22
Bibliografie		23
3	Appendix I: A list of listings	24
3.1	Donkey Republic endpoint response	24
3.2	NGSI Bike Hire Docking Station model example	25
3.3	RML mapping from Donkey Republic to ngsi Bike Hire Docking Station	26
3.4	Simplification of the Donkey Republic RML Mapping for the RML Mapper comparison	29
3.5	Simplification of the SHACL proof of concept for the RML Mapper comparison .	30
3.6	Original Babibubebon/vue-shacl-form ¹¹ output for the RML Mapper comparison	30

Lijst van figuren

2.1	The top view of the mapper which defines the endpoint to use	9
2.2	The bottom view of the mapper, where the actual mapping is made	10
2.3	The model creation page	12
2.4	The RML editors work flow	17
2.5	The YARRRML editors work flow	18
2.6	Comparison between the Babibubebon/vue-shacl-form ¹¹ generated data point and an RML graph which maps to such data point.	21
2.7	The component structure of the RML Mapper [1] Not ported functionality	21

Lijst van tabellen

1.1	User stories	6
1.2	Requirements	7
2.1	Questions	8
2.2	Comparison of different mapping methods [1] Not intractable since generated by the form [2] Automatically generated based on the model	22

Lijst van listings

2.1	Data submitted after mapping an Api	11
2.2	An example YARRML mapping, which maps a Donkey Republic bike hire station endpoint to a part of the NGSI BikeHireStation specification	14
2.3	An example SHACL shape graph which validates a part of the NGSI Bike Hire Docking Station model	15
2.4	Subject of the data point	21
2.5	RML Subject mapping	21
2.6	Type predicate object of the data point	21
2.7	Type RML mapping	21
2.8	ngsic:name predicate object of the datapoint	21
2.9	ngsic:name RML mapping	21
3.1	Data received from the Donkey Republic bike hire station endpoint with a single record	24
3.2	An example record compliant with the ngsi Bike Hire Docking Station data model. Provided as example on the data models page2	25
3.3	An example RML mapping, which maps a Donkey Republic bike hire station endpoint to a part of the NGSI BikeHireStation specification	26
3.4	Simplification of the Donkey Republic to ngsi Bike Hire Docking Station Mapping ^{3.3}	29
3.5	A simplification of SHACL proof of concept ^{2.3} which models part of the ngsi Bike Hire Docking Station as using a shapes graph. Only the name is included.	30
3.6	An example of a Turtle serialized RDF data point generated by Babibubebon/vue-shacl-form ¹¹	30

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 **cacheble endpoint** is provided

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 contributor, i can suggest the addition of my endpoint	
1.1.4	As a contributor, i can semantically map my endpoint to an approved data model	
1.1.5	As a contributor, i can receive feedback about the compliance of my mapping	
1.1.6	As a system administrator, i can create allowed data models	
1.1.7	As a system administrator, i can provide restrictions for those data models	
1.1.8	As a system administrator, i can semantically map an endpoint to a data model	
1.1.9	As a system administrator, i can receive feedback about the compliance of my mapping	

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	cell8
1.2.5	cell8
1.2.6	cell8
1.2.7	cell8

As stated in user story 1.1.3, contributors should be able to add their endpoint

2

Mapping

2.1 Introduction

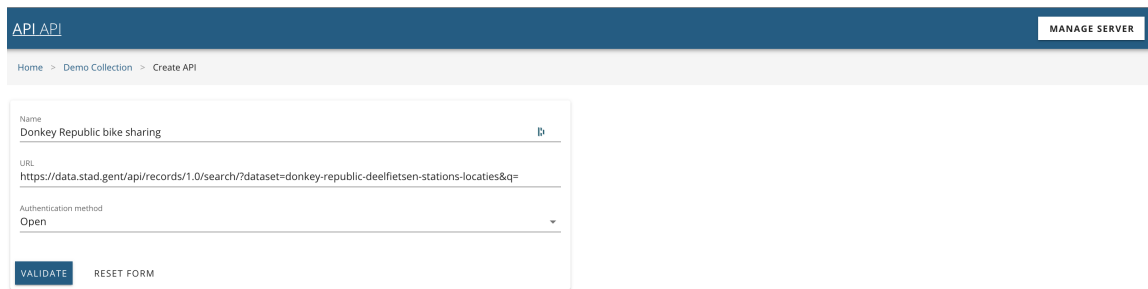
This chapter will cover the Mapping part of ApiApi. The ApiApi framework is built upon the principle of mapping simple data sources to predefined data models. A mapping method from the viewpoint of the user can be defined using 4 characteristics.

Tabel 2.1: Questions

Number	Question
2.1.1	How to define a mapping?
2.1.2	How to input a mapping?
2.1.3	How to define a model?
2.1.4	How to input a model?
2.1.5	How to validate a mapping?

Characteristic 2.1.1

Characteristic 2.1.1 refers to the specification used for defining the mapping information. This holds the information needed for the data transformation.



The screenshot shows the 'API API' web interface. At the top, there's a navigation bar with 'API API' on the left and 'MANAGE SERVER' on the right. Below the navigation bar, there's a breadcrumb trail: 'Home > Demo Collection > Create API'. The main form is titled 'Create API' and contains three input fields: 'Name' with the value 'Donkey Republic bike sharing', 'URL' with the value 'https://data.stad.gent/api/records/1.0/search/?dataset=donkey-republic-deelfietsen-stations-locaties&q=', and 'Authentication method' with a dropdown menu set to 'Open'. At the bottom of the form, there are two buttons: 'VALIDATE' and 'RESET FORM'.

Figuur 2.1: The top view of the mapper which defines the endpoint to use

Characteristic 2.1.2

Characteristic 2.1.2 refers to the method in which the mapping is inputted.

Characteristic 2.1.3

Characteristic 2.1.3 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 2.1.4

Characteristic 2.1.4 refers to the method in which the model is inputted.

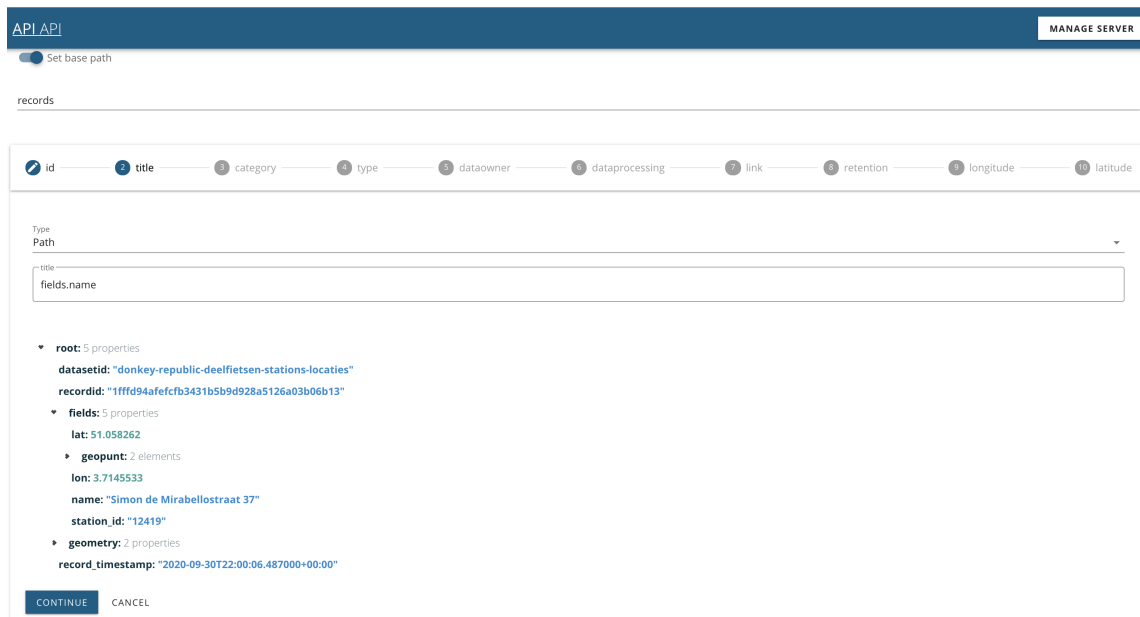
Characteristic 2.1.5

Characteristic 2.1.5 refers to the method used for validating if the mapping is in correspondence with the provided model.

2.2 Current state

2.2.1 The Mapper

ApiApi currently has a visual mapper implementation which maps a JSON endpoint to another JSON format. It does this in several steps. An endpoint is defined using its url and possibly an authentication method if required. Supported authentication methods are: Open, Api key and Custom headers. Open indicates that the api is freely accessible. Api key ... ADD. A name for the Api is required, but serves a purely practical role to differentiate it from other in the collection view. This functionality is displayed in figure 2.2 After defining the endpoint validation should occur by pressing the *Validate* button. During Api validation the mapper checks if the Api's data is accessible by making a REST call to the endpoint which fetches the endpoints data. This



Figuur 2.2: The bottom view of the mapper, where the actual mapping is made

data is then stored to use as a template during the mapping process. After validation the user can select a root for the document. In this context root refers to the key of the JSON property which holds the list of desired items. This is optional. An example of this is displayed in Listing ???. In this example the root should be set to *records*.

Using the base path overrides the current cached API data with the data behind the given JSON property. At the bottom of the page is the actual mapper itself. The tool exists out of several segments. At the top the fields of the outgoing model are displayed. These are the fields were you should map the ingoing model to. Clicking a field will bring you to that property and skip the rest. Using an usual work flow, you iterate over all the outgoing model properties one by one and map them to one of the properties of the ingoing model from the endpoint provided. Iterating can be done using the continue button on the bottom left. Beneath the properties bar is the form where the mapping of the current property is defined. First the field *type* should be filled in. Two options are present: *Constant* and *Path*. Mapping from a constant field means that the outgoing model will have this constant value for this property for each item. Mapping from a path brings up a JSON document beneath the mapper tool. This JSON document represents the first item form your ingoing document. This takes into account the root defined above. Clicking on one of the properties displayed will map that property to the current property of the outgoing model. JSON can be expanded by clicking the arrow next to a nested property. Finally, when reaching the last property of the outgoing model the *continue* option is replaced by a *complete* option. Pressing this will prompt a confirmation and display a pop up with an overview of the mapping you made. The overview holds the names of the properties from the ingoing model you mapped in the correct order. In this overview you can chose to *Submit* or *Reject* the mapping. Submitting

the mapping will send a post request to the server on the *api/api* endpoint. This requests body holds the data as can be seen in code block `??`. *url*, *name* and *authMethod* contain a String with the url, name and authentication method of the endpoint respectively. *forCollection* holds the mongo database id of the Collection which you want to add the Api to. Finally *paths* holds the mapping you created using the Mapper. Paths is implemented as an array of JSON objects, of which each object has the properties: *toPath*, *type* and *value*. *toPath* represents the property of the outgoing model and value represents the name of the ingoing property, or the constant value depending on *type*.

```

apiData () {
  const data = {
    paths: this.devicePaths,
    url: this.url,
    name: this.name,
    authMethod: this.authMethod,
    forCollection: this.forCollection
  }
  return data
}

```

Listing 2.1: Data submitted after mapping an Api

2.2.2 The Model

Model creation is done by providing an example JSON data point. This data point is used to generate a Properties form in which additional information can be inserted. For each property, a hint and default value are supported. Each model also requires a name and a description can be provided optionally.

2.2.3 Back end and database

For persistent storage ApiApi uses MongoDB. A database model is created for both an Api as a Model. When submitting either, a creation request is sent to the back end and a data base entry is created.

write about:
tabase mode
both the api
ping as the r
and the api v
tion

API API MANAGE SERVER

Home > Models > Create Model

Name description

Data Model

```
{
  "id": "102efee90544a3d00943de5fa7b9b3c094e95a12",
  "title": "Real Time data MUV Monitoring Stations",
  "category": "Sensor"
}
```

FORMAT

Properties

path	id	Hint	Default value
path	title	Hint	Default value
path	category	Hint	Default value

SAVE

Figuur 2.3: The model creation page

2.3 Research

Improvements to the mapper and model are defined in user stories: 1.1.4, 1.1.5, 1.1.6 , 1.1.7 , 1.1.8 and 1.1.9. Additionally, requirements 1.2.1, 1.2.2 and 1.2.3 are relevant to this subject. Research for the mapper can thus be split into several topics. **Mapping Languages** covers an exploration of several specifications that can be used for defining a semantic mapping. In **Constraint Languages** SHACL is explored as method to define the data model and imply constraint on such. Finally, **GUI** covers a section dedicated to research involving the creation of a visual mapper.

During this section the Donkey Republic dataset¹ hosted by the city of Ghents data portal will be used to illustrate different concepts. This dataset contains a JSON list of the location of bike hire stations scattered all over the city of Ghent. The dataset holds the record id, location, name and time stamp of the bike hiring stations. A response can be found in listing 3.1

Additionally, some examples require a model to map to. For this the Bike Hire Docking Station model will be used². This model represents a single bike hire station with numerous properties.

¹<https://data.stad.gent/api/records/1.0/search/?dataset=donkey-republic-deelfietsen-stations-locaties&q=>

²<https://fiware-datamodels.readthedocs.io/en/latest/Transportation/Bike/BikeHireDockingStation/-doc/spec/index.html>

This model is a good use case since it incorporates several general linked data model principles such as nested objects, GeoJson and arrays. An example of this model can be found in listing 3.2

2.3.1 Mapping Languages

2.3.1.1 R2RML and variants

R2RML (RDB to RDF Mapping Language)³ is a language designed for creating mapping between relational databases and RDF. These mappings are themselves 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 extensions 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.

2.3.1.2 RML

RML (RDF Mapping Language)⁴ 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 the same syntax as R2RML.

A full specification of RML can be found on its website⁴. 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 I3.3. This proof of concept maps the id, type, name and location of Donkey Republic records to their ngsi counterpart. An RML document 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 record id 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 predi-

³<https://www.w3.org/TR/r2rml>

⁴<https://rml.io/specs/rml/>

cate (`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 its depth is two. To map such complex structure, R2RML provides the option to create a parent-child relation between the base object (the `ngsi:bikeHireDockingStation`) and its child (the `ngsi: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 `lat` 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 `ngsi:GeoProperty` is linked to its child, a `goejson:Point`.

2.3.1.3 YARRRML

YARRRML⁵ 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 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 website⁵. 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:

```

⁵<https://rml.io/yarrml/spec/>

```

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

2.3.2 Constraint Languages

2.3.2.1 SHACL

SHACL⁶ (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 certain id or class will be validated against that shapes graph.

A full specification of the SHACL language can be found on the W3 website⁶. An interesting sideshow to get quick started can be found here⁷. This work will limit itself to describing a proof of concept SHACL definition which validates a part of the ngsi Bike Hire Docking Station.

schrijf over e
ratie en map

```

@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";

```

⁶<https://www.w3.org/TR/shacl/>

⁷<https://www.slideshare.net/jelabra/shacl-by-example>

```

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

2.3.3 GUI

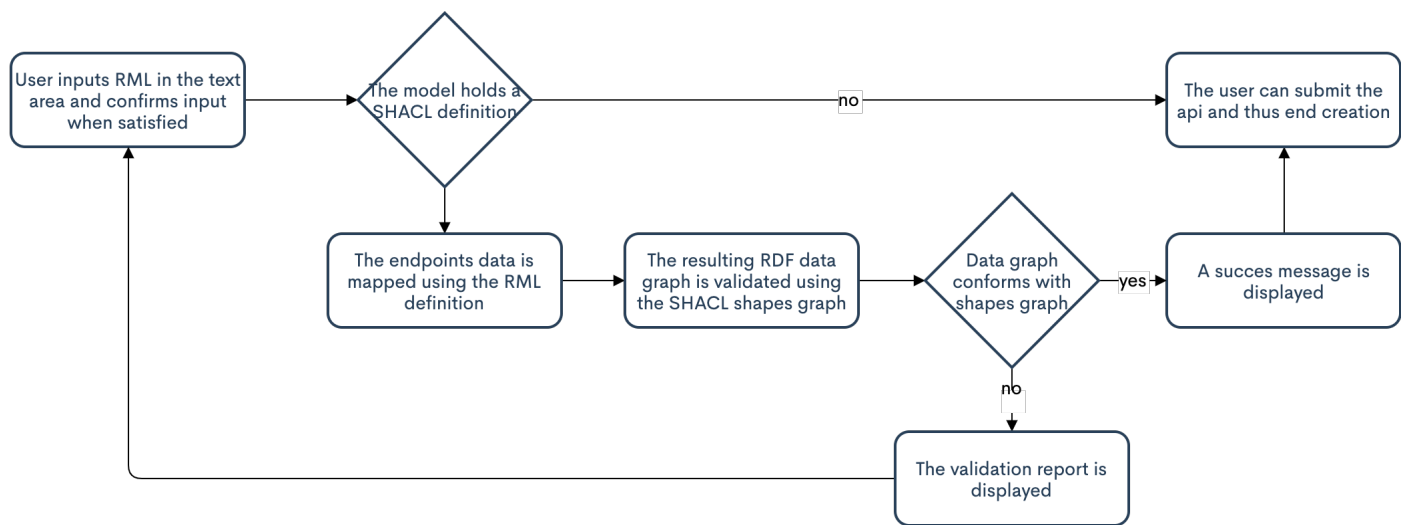
2.4 Implementation

2.4.1 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-js⁸ is used. SHACL validation is done using TopQuadrant/shacl-js⁹. 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.

⁸<https://github.com/RMLio/rmlmapper-java-wrapper-js>

⁹<https://github.com/TopQuadrant/shacl-jsrmlmapper-java-wrapper-js>



Figuur 2.4: The RML editors work flow

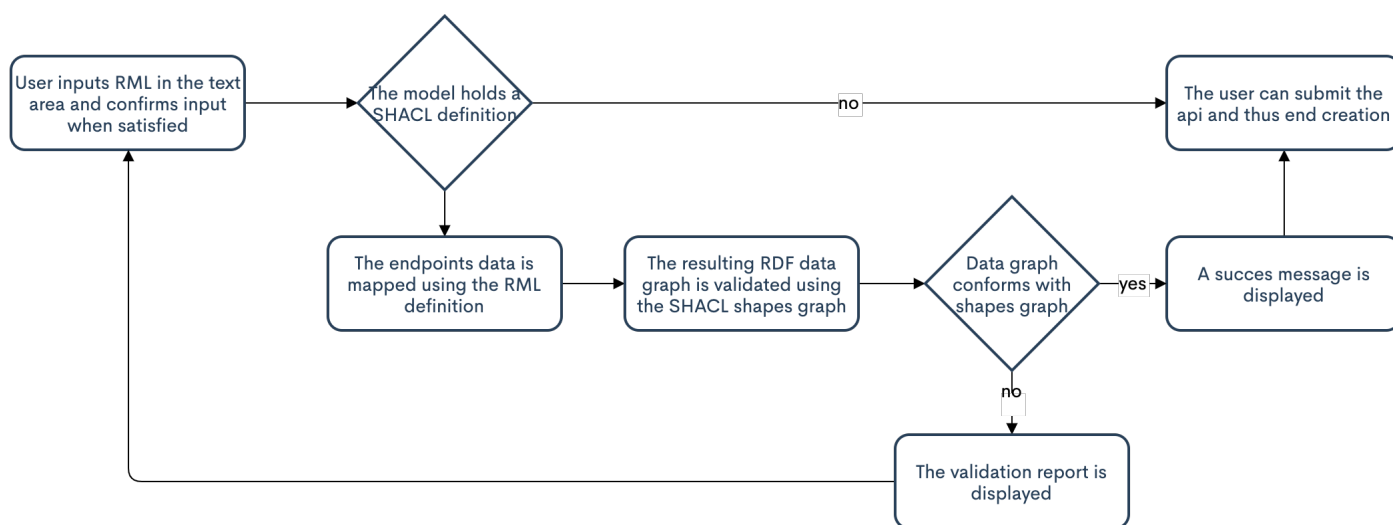
2.4.2 YARRRML Editor

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/yarrml-parser` footnote <https://github.com/rmlio/yarrml-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`⁸ 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
yarrml map

2.4.3 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.



Figuur 2.5: The YARRRML editors work flow

Additionally, this SHACL definition is used to construct the RML Mapper^{2.4.4} when selected in the Api/create component.

2.4.4 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 2.7. 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 path¹⁰ 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

¹⁰information about the JSON path syntax can be found here: <https://restfulapi.net/json-jsonpath/>

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 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.

2.4.4.1 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-form¹¹ which is explained more in detail in section .

[add ref](#)

2.4.4.2 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. 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-form¹¹ 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-

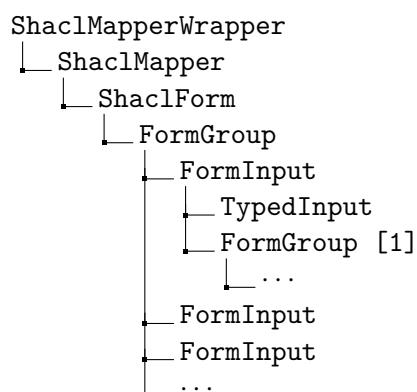
¹¹<https://github.com/Babibubebon/vue-shacl-form>

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 3.4](#). 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 3.5](#) 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^{3,4}. Compare this to a single data point which is the output of Babibubebon/vue-shacl-form¹¹ based on the SHACL shape graph^{3.5}. Although different looking, several similarities can be observed. When excluding the prefix and `rml:logicalSource` tripe, a one on one comparison can be made between the lines of the data point and the remaining RML triples. This is demonstrated in figure 2.6. 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.

Listing (2.4) Subject of the data point	Listing (2.5) RML Subject mapping
<hr/> <code><https://www.donkey.bike/5743></code> <hr/>	<hr/> <code>rr:subjectMap [rr:template "https://www.donkey.bike/{recordid}"];</code> <hr/>
Listing (2.6) Type predicate object of the data point	Listing (2.7) Type RML mapping
<hr/> <code>a ngsi:BikeHireDockingStation ;</code> <hr/>	<hr/> <code>rr:predicateObjectMap [rr:predicate rdf:type; rr:objectMap [rr:constant ngsi:BikeHireDockingStation]];</code> <hr/>
Listing (2.8) ngsi:name predicate object of the datapoint	Listing (2.9) ngsi:name RML mapping
<hr/> <code>ngsi:name "Station Antwerpen-centraal" .</code> <hr/>	<hr/> <code>rr:predicateObjectMap [rr:predicate ngsi:name; rr:objectMap [rml:reference "fields.name"]].</code> <hr/>

Figuur 2.6: Comparison between the Babibubebon/vue-shacl-form¹¹ generated data point and an RML graph which maps to such data point.



Figuur 2.7: The component structure of the RML Mapper [1] Not ported functionality

2.4.4.3 Component structure

2.5 Final Remarks

Tabel 2.2: Comparison of different mapping methods [1] Not intractable since generated by the form [2] Automatically generated based on the model

±	Q 2.1.1	Q 2.1.2	Q 2.1.3	Q 2.1.4	Q 2.1.5
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

ments
itorx, map-

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>

3

Appendix I: A list of listings

3.1 Donkey Republic endpoint response

```
1 {
2   "nhits": 569,
3   "parameters": {
4     "dataset": "donkey-republic-deelfietsen-stations-locaties",
5     "timezone": "UTC",
6     "rows": 10,
7     "start": 0,
8     "format": "json"
9   },
10  "records": [
11    {
12      "datasetid": "donkey-republic-deelfietsen-stations-locaties",
13      "recordid": "1fffd94afefcfb3431b5b9d928a5126a03b06b13",
14      "fields": {
15        "lat": 51.058262,
16        "geopunt": [
17          51.058262,
```



```

18         3.7145533
19     ],
20     "lon": 3.7145533,
21     "name": "Simon de Mirabellostraat 37",
22     "station_id": "12419"
23 },
24 "geometry": {
25     "type": "Point",
26     "coordinates": [
27         3.7145533,
28         51.058262
29     ]
30 },
31 "record_timestamp": "2020-10-31T23:01:08.411000+00:00"
32 },
33 ...
34 ]
35 }

```

Listing 3.1: Data received from the Donkey Republic bike hire station endpoint with a single record

3.2 NGSi Bike Hire Docking Station model example

```

1 {
2     "id": "urn:ngsi-ld:BikeHireDockingStation:Bcn-BikeHireDockingStation-1",
3     "type": "BikeHireDockingStation",
4     "status": {
5         "type": "Property",
6         "value": "working"
7     },
8     "availableBikeNumber": {
9         "type": "Property",
10        "value": 20,
11        "observedAt": "2018-09-25T12:00:00Z"
12    },
13    "freeSlotNumber": {
14        "type": "Property",

```

```

15     "value": 10
16 },
17 "location": {
18     "type": "GeoProperty",
19     "value": {
20         "type": "Point",
21         "coordinates": [2.180042, 41.397952]
22     }
23 },
24 "address": {
25     "type": "Property",
26     "value": {
27         "addressCountry": "ES",
28         "addressLocality": "Barcelona",
29         "streetAddress": "Gran Via Corts Catalanes,760",
30         "type": "PostalAddress"
31     }
32 },
33 "@context": [
34     "https://schema.lab.fiware.org/ld/context",
35     "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
36 ]
37 }

```

Listing 3.2: An example record compliant with the ngsi Bike Hire Docking Station data model. Provided as example on the data models page2

3.3 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;

```

3.3. RML MAPPING FROM DONKEY REPUBLIC TO NGSi BIKE HIRE DOCKING STATION²⁷

```
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 3.3: An example RML mapping, which maps a Donkey Republic bike hire station endpoint to a part of the NGSi BikeHireStation specification

3.4 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 3.4: Simplification of the Donkey Republic to ngsi Bike Hire Docking Station Mapping^{3.3}

3.5 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 3.5: A simplification of SHACL proof of concept^{2,3} which models part of the ngsi Bike Hire Docking Station as using a shapes graph. Only the name is included.

3.6 Original Babibubebon/vue-shacl-form¹¹ 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 3.6: An example of a Turtle serialized RDF data point generated by Babibubebon/vue-shacl-form¹¹