NTNU – Trondheim
Norwegian University of
Science and Technology

Model-Driven Development of REST APIs

Tomás Procházka

Master in Information Systems
Submission date: June 2015
Supervisor: John Krogstie, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

The domain of REST APIs contains highly repetitive code which needs to be written every time there is a need for a new REST API. When developing client-side applications, like web applications or native applications, one must also create a robust REST API in order to share the data between all the clients. The server-side technology takes a huge part of the budget and in most cases it is not the main product- the client-side application is. The author is trying to solve this problem by applying the Model-Driven Development paradigm. A code generator specifically designed for the domain is proposed and developed. The code generator has some unique features such as GitHub integration and mechanisms to structure the files in the same way as human would do. This has been achieved by analyzing the human created file structure. The generated REST API comes also with an option of automatically testing behavior compared to best practices. This is done by taking an existing solution and further researching its use.

Preface

This report has been created as a documentation of my Master Thesis in Information Systems at the Department of Computer and Information Science at the Norwegian University of Science and Technology. The thesis has been written in cooperation with Searis AS.

I would like to thank my supervisor professor John Krogstie for inspiration, patience, and mainly a lot of useful advise. A huge thank you also belongs to the guys from Searis AS for their useful comments, evaluation and mainly the smooth cooperation I had with them. Last but not least to my family since this would not be possible without them. A special thank you also goes to my girlfriend Lisa, because she had to handle me in moods she will hopefully never ever see me in again. I am glad I was lucky enough to surround myself with such awesome people.

Tomas Prochazka

Table of Contents

Abstract i

Preface ii

Table of Contents v

List of Tables vii

List of Figures x

Abbreviations xi

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Research Questions . . . 2
- 1.3 Thesis Outline . . . 2

2 Background 5
- 2.1 REST . . . 5
  - 2.1.1 Best Practices When Creating URIs . . . 6
  - 2.1.2 HTTP Methods . . . 7
- 2.2 REST API Notations . . . 9
  - 2.2.1 RAML . . . 9
  - 2.2.2 Swagger . . . 9
  - 2.2.3 API Blueprint . . . 10
  - 2.2.4 JAX-RS . . . 11
- 2.3 Projects . . . 11
  - 2.3.1 Apiary . . . 11
  - 2.3.2 Restlet . . . 11
  - 2.3.3 Project Texo . . . 12
  - 2.3.4 EMF REST . . . 12

2.3.5 Web Based Editor . . . 13
2.4 Model-Driven Software Development . . . 13
2.4.1 Template deriving . . . 14
2.4.2 Protected areas . . . 14
2.5 Code generators . . . 15
2.5.1 Acceleo . . . 15
2.5.2 Project Matilda . . . 15
2.6 Conclusion . . . 15

3 Research **17**
3.1 Design Science . . . 17
3.2 Guidelines . . . 17

4 Goals and Requirements **21**
4.1 Goals . . . 21
4.2 Requirements . . . 22
4.2.1 Functional Requirements . . . 22
4.2.2 Non-Functional Requirements . . . 24

5 Implementation **25**
5.1 Architecture . . . 25
5.2 Technology . . . 25
5.2.1 NodeJS and NPM . . . 26
5.2.2 MongoDB . . . 26
5.2.3 MochaJS . . . 27
5.2.4 GruntJS . . . 27
5.2.5 UnderscoreJS . . . 27
5.2.6 Dredd . . . 28
5.2.7 Git . . . 28
5.2.8 Why not Java? . . . 28
5.3 Generation Process . . . 29
5.3.1 Meta-Model . . . 31
5.3.2 Scope . . . 32
5.3.3 Templates . . . 33
5.3.4 Templates Load . . . 36
5.3.5 Templates Execution . . . 37
5.3.6 Code Beautification . . . 39
5.3.7 Templates Saver . . . 39
5.3.8 Generator's Configuration File . . . 40
5.3.9 Generator's Endpoints . . . 40
5.3.10 Git Integration and Versioning . . . 42
5.3.11 Download, Installation, Run and Test . . . 44
5.4 REST API template . . . 44
5.4.1 Folder Structure . . . 44
5.4.2 Filtering and Sorting . . . 45
5.4.3 Behavior prediction . . . 46

5.4.4 API Blueprint . . . 47
5.5 Template Development Workflow . . . 47
5.5.1 Prerequisites . . . 48
5.5.2 Template Development Process . . . 48
5.6 The Editor Extensions . . . 49
5.6.1 Endpoint handling . . . 50

6 Evaluation 51
6.1 Functional Requirements . . . 51
6.1.1 Meta-Model Independence . . . 51
6.1.2 Platform Independence . . . 52
6.1.3 Endpoint Completeness . . . 53
6.1.4 Model Completeness . . . 56
6.1.5 Best Practice Behavior . . . 59
6.1.6 Code Quality . . . 60
6.1.7 Folder Structure Quality . . . 63
6.1.8 Re-Generating Management . . . 64
6.1.9 Template Re-Usability . . . 65
6.1.10 Output Testability . . . 65
6.2 Non-Functional Requirements . . . 66
6.2.1 Cross-Platform . . . 66
6.2.2 Output Testability . . . 67
6.2.3 Template Sharing . . . 67
6.2.4 Speed . . . 67
6.3 Limitations . . . 68

7 Discussion 69

8 Conclusion and Future Work 71

Bibliography 72

Appendix A- Study Case 75

Appendix B- Videos 84

Appendix C- External Assessment 85

Appendix D- Unit Tests 87

List of Tables

2.1 Expected behavior of a request with a combination of a method and URI . 8

4.1 Functional requirements 23
4.2 Non-functional requirements 24

6.1 Comparison of the modeled endpoints with the generated ones 56
6.2 Expected behavior related to the Dredd test results 60

List of Figures

2.1 Valid URI illustrating the rules (red numbers) 7
2.2 RAML standard in action (source: http://raml.org/index.html) 9
2.3 Swagger standard in action (source: http://editor.swagger.io/) 10
2.4 API Blueprint in Apiary online editor 10
2.5 The process of template deriving, source: (Stahl and Völter, 2005) 14
2.6 An example of a protected area in a template, source: (Stahl and Völter, 2005) 14

4.1 The goals and derived requirements 22

5.1 The generation process 30
5.2 The implementation of the Meta-Model 32
5.3 The usage of a scope helper and its definition 33
5.4 The template types 34
5.5 The template types 35
5.6 The template snippet showing the template notation 36
5.7 The processing of duplicated templates 38
5.8 The endpoints of the generator, their inputs and the output 41
5.9 Merging process with custom code changes 43
5.10 Template behavior prediction 46
5.11 The recommended setup for template development 48
5.12 The template development process (Source of GruntJS logo:`http://gruntjs.com/`, MochaJS logo: `http://mochajs.org/`, NodemonJS logo:`http://nodemon.io/`) 49
5.13 The implemented extensions in the editor 50

6.1 The modeled endpoints in the use case 54
6.2 The entity Oven in the use case model 57
6.3 The actual object of type Oven returned from the generated REST API . . 58
6.4 Best practices checked with Dredd 59
6.5 The REST API structured by a programmer 63

6.6 File structure of the generated REST API 64
6.7 Dredd output 66

Abbreviations

JSON	=	JavaScript Object Notation
REST	=	Representational State Transfer
API	=	Application Programming Interface
MDD	=	Model-Driven Development
MDSD	=	Model-Driven Software Development
DSL	=	Domain Specific Language

Chapter 1

Introduction

1.1 Background and Motivation

Application development has been divided into two main fields: web based development and native application development. Web based applications are in general applications which run in the browser and allow the user to interact with the interface shown as a web page. Native applications are applications written for a specific platform or device. For example iOS from Apple, Android developed by Google or Microsoft's Windows Phone. One of the keys to success for technology focused companies is to support all the platforms in order to target a wide audience.

When targeting several platforms, one of the biggest challenges is to synchronize the data between all applications so available data are the same on all the platforms. The ability to cooperate with other systems, in this case different clients, is called interoperability. To be exact, interoperability is the "ability of a system or a product to work with other systems or products without special effort on the part of the customer"(IEEE, 2010) and that is exactly what is required from the system.

Web APIs solve this problem as they have a high level of interoperability. All the applications have only one data source, in this case a server, which provides data to all clients and makes sure that the data are up-to-date. The ability to communicate with the server is achieved by standardization. It is a simple mechanism where the server has a standardized interface and the clients implement this interface and use it. This is not an easy task and the distributed system can get very complex. Roy Fielding, the author of REST, therefore collected a number of constraints in order to get certain values from the system. These constraints are known as a Representational State Transfer architectural style or simply REST.

REST APIs can be very complex in terms of software architecture, error handling, security, access rules etc. To implement this solution according to all the best practices is time consuming and it is an expensive part of every software project. This time and money should be used on the client development which is usually the desired end product. Projects like Apiary, Restlet or Backendless try to attack this problem and make the server-

side development process as smooth as possible, but none of them focuses on the actual code generation that would give a solid base for the REST API development or even better: generate the entire API.

In this report, I will attack this problem with the Model-Driven Development paradigm because the domain of REST APIs has a lot of repetitive code and MDD can abstract from that. Model-Driven Development is a development paradigm that uses models as the primary artifact of the development process. Usually, in Model-Driven Development the implementation is semi-automatically generated from the model(Marco Brambilla, 2012). Code generation has been introduced in the new version of Golang [1] and also in the latest specification of ECMAScript 6, as can be seen in the presentation[2] of Axel Rauschmayer from the Rolling Scopes conference in March 2015. This shows that the idea of generating code is not only present in the traditional programming languages such as Java. This report therefore proposes a JavaScript based code generator tailored to generate REST APIs.

1.2 Research Questions

1. Is it possible to generate production ready, structured and testable REST APIs based on a data model and endpoints?
2. How does one share the knowledge of best practices about REST APIs, so others without knowledge can use it?

1.3 Thesis Outline

This report is structured into following chapters:

- **2. Background** - This chapter describes what has been already done and summarizes why and what should include the new solution.
- **3. Research** - This chapter describes the research method of this report.
- **4. Theory** - This chapter gives an introduction into the theory behind code generation, model-driven development and REST API best practices.
- **5. Requirements** - This chapter specifies the requirements to the code generator.
- **6. Implementation** - This chapter describes the implementation itself, the architectural decisions that have been made and the usage of the code generator.
- **7. Evaluation** - This chapter describes the evaluation of the implementation.
- **8. Discussion** - This chapter describes the findings in relation to what has been already done and how it improves the current status of the addressed problem.

[1] https://blog.golang.org/generate - The Go Blog - Generating code

[2] https://www.youtube.com/watch?v=Fg3bEZIcnUw - Dr. Axel Rauschmayer - Using ECMAScript 6 today in 9:24

- **9. Conclusion and Future Work** - This chapter describes the possibilities for future research and wraps up the report.

Chapter 2

Background

This chapter describes already existing knowledge and the gaps this report is trying to fill in. The chapter has three sections. First, the REST API standards are presented. In order to show what has been already done, a couple of projects are presented. These projects are somehow related to the domain of REST APIs and are worth to mention. At the end an existing code generators are presented.

2.1 REST

REST is an acronym for REpresentational State Transfer and it is an ”architectural style for distributed hypermedia systems”(Fielding, 2000). REST is not considered to be an architecture but it is described as a ”set of constraints applied to elements within the architecture”(Fielding, 2000). Fielding describes the constraints in his dissertation as:

- **Client-Server** - This is nothing else than separation of concerns. The client should focus on how to show the data and the server on how to provide and store the data. There is no space for overlapping.
- **Stateless** - The client should provide all the information needed to handle the request. There should be no dependency on the previous communication between these two.
- **Cache** - The data provided by the server shall be cacheable. This means that the data shall be marked with *cacheable* or *non-cacheable*. If the response is cacheable, the client can store the data and use it later. This constraint improves the efficiency of the network.
- **Uniform interface** - All the services shall have an uniform interface. The only drawback of this is that the data are sent in a standardized form which does not always correspond with the application’s needs.

- **Layered system** - Every component of the system shall know exactly one layer of the system. This helps encapsulate the legacy services and legacy clients. The drawback of the layered systems is the fact that they suffer from latency. This problem can be solved with caching the data.
- **Code-on-demand** - This is an optional constraint of REST. Fielding says that the clients shall be able to download scripts from the server and execute them in order to simplify the clients.

By applying these constrains to a distribution system, one gets certain properties from the system. Pete Hunt, engineer at Facebook, describes in his talk the properties as following(Hunt, 2014):

- performance
- scalability
- simplicity
- modifiability
- visibility
- reliability

As it can be seen, one of the constraints is an uniform interface. This is a very important point and there are lot of misconceptions about how to design the interface according to best practice. The next section therefore describes what is the best practice in this field. This is highly relevant because the generator can then rely on the practices and predict the behavior of each endpoint.

2.1.1 Best Practices When Creating URIs

Mark Mass describes in his book several rules which should be considered when creating URIs(Massé, 2012):

1. Forward slash separator (/) must be used to indicate a hierarchy
2. A trailing forward slash (/) should not be included in URIs
3. Hyphens (-) should be used to improve the readability of URIs
4. Underscores (_) should not be used in URIs
5. Lowercase letters should be preferred in URI paths
6. File extensions should not be included in URIs
7. A singular noun should be used for document names
8. A plural noun should be used for collection names

9. A plural noun should be used for store names
10. A verb or verb phrase should be used for controller names
11. Variable path segments may be substituted with identity-based
12. CRUD function names should not be used in URIs

The following figure illustrates the rules on a valid URI.

9 11 10
/magazines/3804/published-articles/238/rank
8 1 3 2

Figure 2.1: Valid URI illustrating the rules (red numbers)

The URI above is using slashes as a separator (rule number 1), has no slash at the end (rule number 2) and is using hyphens instead of underscores (rule number 3). As it can be seen, only the plural nouns are used (rule number 8 and 9) and the action on the resource is specified by a verb (rule number 10). The last rule, number 11, is illustrated with using IDs as a variable.

There is a well-known GitHub repository in the web community with REST API standards[1]. The standards described here are in most cases in harmony with the one Mark Mass suggests, the only difference is that Carden allows to use file extensions at the end of the URI (rule number 6).

According to both resources there are three different types of URIs:

1. URIs with a plural noun at the end returning a collection for example */cars* or */universities/123/departments*
2. URIs with a variable at the end returning a single object for example */cars/342* or */universities/123/departments/456*
3. URIs with a verb at the end describing an action */cars/123/wreck* or */universities/123/rank*

Another very important part of a request is a method which specifies what should be done with the targeted resource. The next section describes how each methods shall be used. This is again highly relevant because it can help predict the desired behavior of an endpoint in the generator.

2.1.2 HTTP Methods

The HTTP protocol has 9 different methods. Only 6 of them are widely used: GET, POST, PUT, DELETE, HEAD and OPTIONS. HEAD and OPTIONS are special methods. HEAD is used to return only the headers of the response and OPTION is for getting allowed methods on a resource(Massé, 2012). The others are used for operating with resources. Mass

[1] https://github.com/WhiteHouse/api-standards - REST API standards by Travis Carden

xxx	*/universities*	*/universities/233*	*/universities/293/rank*
GET	returns a collection of universities	returns a single university object	returns error
POST	adds a new university object	returns error	executes the controller attached to the URI
PUT	edits all universities in the collection	edits a single university	returns error
DELETE	deletes the entire collection	deletes a single university	returns error

Table 2.1: Expected behavior of a request with a combination of a method and URI

in his book describes a couple of rules which should be followed when using them(Massé, 2012):

1. GET and POST must not be used to tunnel other request methods
2. GET must be used to retrieve a representation of a resource
3. HEAD should be used to retrieve response headers
4. PUT must be used to both insert and update a stored resource
5. PUT must be used to update mutable resources
6. POST must be used to create a new resource in a collection
7. POST must be used to execute controllers
8. DELETE must be used to remove a resource from its parent
9. OPTIONS should be used to retrieve meta-data that describes resource's available interactions

Carden's description in general agrees with the one Mass gives, the only difference is that Carden suggests to use PUT only to update a resource. An interesting use of PATCH method can be seen in Vinay Sahni's post, who suggests to use PATCH method for a partial update[2] . The same approach is suggested by Mario Cardinal in his presentation about architecting a Pragmatic Web API[3] . The following table shows the behavior of each method with one of the three types of URI mentioned above.

[2]http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api - Best Practices for Designing a Pragmatic RESTful API by Vinay Sahni

[3]http://www.slideshare.net/mario_cardinal/best-practices-for-designing-pragmatic-restful-api - Best Practices for Architecting a Pragmatic Web API by Mario Cardinal

2.2 REST API Notations

Introducing MDD into REST API development is not a new idea and there are already some existing standards for expressing models.

2.2.1 RAML

RAML is a REST API Modelling Language which provides a way to practically describe REST API parameters and endpoints. The philosophy of RAML is to have a platform independent standard for describing REST APIs and do not include all the constrains of the REST architectural style(MuleSoft, 2015). RAML is based on YAML data serialization standard and JavaScript Object Notation also known as JSON. The picture below shows an example of RAML document structure.

```
/{songId}:
  get:
    responses:
      200:
        body:
          application/json:
            schema: |
              { "$schema": "http://json-schema.org/schema",
                "type": "object",
                "description": "A canonical song",
                "properties": {
                  "title":  { "type": "string" },
                  "artist": { "type": "string" }
                },
                "required": [ "title", "artist" ]
              }
          application/xml:
```

Figure 2.2: RAML standard in action (source: http://raml.org/index.html)

There are several tools which benefit from the RAML notation. The tools can be divided into 3 groups:

1. tools for generating the client-side code based on the model
2. tools for mocking the REST API based on the model
3. tools supporting the user experience when creating the model
4. other tools for validation, continuous integration and bridges

The biggest problem with the notation is the lack of readability for humans. Some of the parameters in the specification are not clear which can lead to a misuse of the standard. A huge plus is the syntax used. RAML has an indentation based syntax so it does not force the user to use any curly brackets.

2.2.2 Swagger

Swagger is presented as ”Framework for APIs”(Reverb, 2015). It has a YAML based notation. The syntax of Swagger is shown below.

```
/products:
    get:
      summary: Product Types
      description: |
        The Products endpoint returns information about
the *Uber* products
        offered at a given location. The response includes
the display name
        and other details about each product, and lists
the products in the
        proper display order.
      parameters:
        - name: latitude
          in: query
          description: Latitude component of location.
          required: true
```

Figure 2.3: Swagger standard in action (source: http://editor.swagger.io/)

There are a lot of tools available including client-side SDK generators and server-side generator called Swagger-Codegen. Swagger-Codegen is able to generate a server based on the specification. It uses MustacheJS templates as a template engine. This solution has several disadvantages:

- it does not support nested templates
- it supports a limited set of logic in the templates
- there is no way to inject custom helpers
- generated code has a very human unfriendly structure

This code generation is good for mocking the server but not for production. In general Swagger is very similar to RAML, but it has better tool support.

2.2.3 API Blueprint

API Blueprint is a specification developed by a company called Apiary[4]. API Blueprint has been developed, because CEO Jakub Nesetril found other notations very complicated(Nesetril, 2012). The notation is based on Markdown markup language. An example of the notation can be seen in the figure below.

```
## Notes Collection [/notes]
### List all Notes [GET]
+ Response 200 (application/json)

        [{
          "id": 1, "title": "Jogging in park"
        }, {
          "id": 2, "title": "Pick-up posters from post-office"
        }]

### Create a Note [POST]
+ Request (application/json)

        { "title": "Buy cheese and bread for breakfast." }

+ Response 201 (application/json)

        { "id": 3, "title": "Buy cheese and bread for breakfast." }
```

Figure 2.4: API Blueprint in Apiary online editor

[4] https://apiary.io/

API Blueprint has been designed to be understandable for computers and humans at the same time(Nesetril, 2012). There is a wide range of support tools available for this notation including mocking of a passive server and automated testing of REST APIs against a passed blueprint model.

2.2.4 JAX-RS

JAX-RS is not really a notation, but it is a set of annotations, which allow the developer to annotate Java code and create REST web services. It aims to make development of RESTful web services simple and intuitive (Burke, 2009). JAX-RS is tightly coupled to the Java stack. Since I have decided to not use Java, JAX-RS is not an option due to its tight dependency on Java. More about the technological decisions is in the section 5.2.

2.3 Projects

This section describes projects which are relevant to the REST APIs and their generation.

2.3.1 Apiary

Project Apiary has been established to solve 3 problems(Nesetril, 2012):

- waterfall programming cycle
- documentation
- API support

Apiary[5] has a text editor which allows the user to specify the REST API by using API Blueprint notation. While the user is typing, a documentation is generated based on the textual model. Also a mocked REST API is generated according to the specifications in the model. This makes collaboration between several developers on one API very easy. It is also possible to develop client-side applications against the mocked API, but at the end it does not solve the problem that somebody has to develop the server-side part.

Apiary is a great tool for developing REST APIs but does not attack the problem of code generation. Thanks to the fact that they open sourced API Blueprint there are several very handy tools developed by the community. The complete list can be seen here: `http://apiblueprint.org/`.

2.3.2 Restlet

Restlet[6] is an interesting project combining 3 products:

- Restlet Framework
- APISpark

[5] http://apiary.io/ homepage of the Apiary project
[6] http://restlet.com/ homepage of Restlet

- Restlet Studio

Restlet Framework is a Java based framework for developing REST APIs in Java. This framework helps developers to build API according to the REST API standards. APISpark is a platform for creating new APIs. This tools are able to turn for example a Google Spread sheet into a REST API. The last product, Restlet Studio is a visual studio which allows to create Swagger and RAML files without any knowledge of the formats.

2.3.3 Project Texo

Project Texo[7] ”provides annotation driven code generation for server-side web application environments”(Taal, 2015). Texo is implemented in Java and is tightly coupled to the Eclipse Modeling Framework, Eclipse in general and Tomcat. It generates a REST API according to an *ecore* model. It also has some handy features such as dummy data generation. Texo is attacking the problem of auto generating REST APIs but has a lot of drawbacks. They together create a strong argument why not to use the tool. The drawbacks are:

- very strongly dependent on Java, Eclipse, EMF, JPA and Tomcat
- there is only one person actively improving the project which can really easily affect the future of the project
- the generated API does not follow the best practices in creating URIs as can be seen in the Texo's demo video[8]

There is still a lot of work that needs to be done with Texo and at the moment it does not seem mature enough for bigger projects.

2.3.4 EMF REST

EMF REST is a framework build on the top of the Eclipse/Java/EMF development stack and it transforms an *ecore* model into a functional REST API. Eclipse Modeling Framework is a ”code generation facility for building tools and other applications based on a structured data model”(Fouquet and collective, 2012). This is a very nice solution for people familiar with EMF and *ecore* models. It also provides a JavaScript library for the generated API so the user can include this library and use it as a middle-man in the communication between the server and the client. It is meant to be a solution useful for prototyping and validation purposes(Ed-douibi and collective, 2015). This solution has a couple of drawbacks:

- tight dependency on Eclipse, EMF and Tomcat
- no support for custom endpoints, only CRUD operations are supported
- it is not obvious if the solution is using any database or the returned data is just static

[7] https://projects.eclipse.org/projects/modeling.emft.texo homepage of the project Texo
[8] https://www.youtube.com/watch?v=shyas-9gvg8 - Demo video of the Project Texo

- The supported HTTP methods are POST, PUT, DELETE and GET(EMF-REST, 2015). This means that it does not support preflighted requests which first ask the server for available methods with an OPTIONS call(Mozilla, 2015). This is crucial because most client-side libraries are using preflighted requests, namely AngularJS, EmberJS or even jQuery.

2.3.5 Web Based Editor

I have created a web based editor in the course TDT4501 Specialization Project at Norwegian University of Science and Technology. This editor allows to model data together with their endpoints and then export the entire model into a JSON data format. This format can be then used for code generation of any kind.

2.4 Model-Driven Software Development

This section describes some of the key concepts of MDSD. MDSD is a software development methodology which aims to solve following problems(Stahl and Völter, 2005):

- increase development speed
- enhance software quality
- centralize the place for making changes
- higher level of re-usability
- manageability of complexity
- interoperability and portability

In the traditional software development models are only part of the documentation. MDSD is looking at this differently. It embraces the models and includes them as a part of the software(Stahl and Völter, 2005). This is possible due to transformations. Model transformations ”allow the definition of mappings between different models”(Marco Brambilla, 2012). There are two major categories of the model transformation identified(Czarnecki and Helsen, 2003):

- model-to-model transformation
- model-to-code transformation

Model-to-code transformation is the relevant one for this report. This transformation can be divided into two different categories: visitor-based approaches and template-based approaches. The template-based approach uses templates which ”usually consist of the target text containing splices of meta-code to access information from the source and to perform code selection and iterative expansion”(Czarnecki and Helsen, 2003).

2.4.1 Template deriving

The traditional and most effective approach of how to identify and create templates is to analyze an existing application and divide the code base into 3 different groups(Stahl and Völter, 2005):

- Generic Code - this code is the same and does not change in any way.
- Schematic Repetitive Code - this code is possible to divide into patterns and replace the key information with the meta-code in order to create a template.
- Individual Code - this code is impossible to generate and the programmer needs to fill in the code.

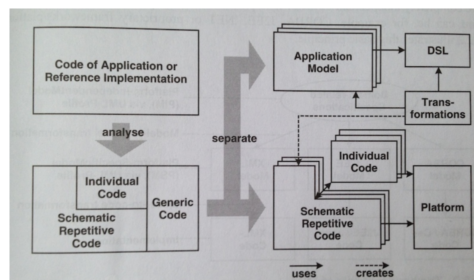

Figure 2.5: The process of template deriving, source: (Stahl and Völter, 2005)

The picture above shows the entire process of template derivation. An existing application is shown in the top left corner. This application's code is divided into the 3 groups and then separated to an Application Model and the Schematic Repetitive Code. This code is created by the transformation of the model described in the domain specific language. By attaching the individual code identified in the first phase one gets the same application like at the beginning, but with all the parameters and characteristics MDSD offers.

2.4.2 Protected areas

Protected areas are well known in the Model-Driven Development and they "are used to mark sections in the generated code that shall not be overriden again by subsequent generator runs"(Stahl and Völter, 2005). This is achieved by marking a certain part of the template with *protected* keyword as it can be seen in the figure below. The purpose is to protect manually inserted code.

```
[template public javaMethod(op : Operation)]
  public [op.type/] [op.name/]() {
    // [protected (op.name)]
    // Fill in the operation implementation here!
    [returnStatement(op.type)/]
    // [/protected]
  }
[/template]
```

Figure 2.6: An example of a protected area in a template, source: (Stahl and Völter, 2005)

2.5 Code generators

This section describes existing code generators.

2.5.1 Acceleo

Acceleo is an Eclipse based code generator which can also be used as a stand-alone application, but its functionality is in that case limited. It is an interesting generator, that treats templates as if they were regular modules. It has support for so called *queries* which lets you define a module and then reuse it in other modules. Acceleo is generating any kind of programming language so it is not strictly tailored to Java. As an input, it not only supports *ecore* models, but also UML models and XMI models. The latest release of Acceleo has been published in 2011 and it seems like no other releases are planned. I have also checked the service `http://www.stackoverflow.com` and there are only 103 questions asked during the 9 years of existence. This indicates that the product is not widely used and partially abandoned. The biggest problem I can see with Acceleo is that it does not support any template sharing. I have tried to search for some templates I could reuse but the result was disappointing. Due to these facts, I do not think that Acceleo is a vital growing project.

2.5.2 Project Matilda

Matilda is a Java based modeling framework developed at Massachusetts Institute of Technology and it is addressing the following problems(Hiroshi Wada and Oba, 2010):

- abstraction gap between modeling and programming layers
- a lack of traceability between models and programs
- a lack of customizability

Matilda accepts UML and BPMN models and transforms them into runnable code. It has a distributed pipeline system of plugins which allows collaboration on building and integrating models between developers over a network. The important part of Matilda is a meta-model which is used for validating the incoming models. The biggest weakness is the number of people involved in the development. There is also no possibility to share templates therefore an extensive knowledge of model transformation is required. Matilda counts on 4 different roles in the process of development and covering all the roles by a one developer creates a huge knowledge gap that needs to be filled.

2.6 Conclusion

The field of Model-Driven Development in REST APIs is known, but I believe it can be brought even further. There are several notations which allow to model the important aspects of the REST API. There are also several open-source tools build on the notations, so supporting at least one opens the door to already existing tools. The same counts for

already existing projects which build on the notations. Apiary is generating a really nice documentation from the API Blueprint file and also provides the first steps to make an HTTP request regardless of which platform is used. It seems like that the world of Java is much further in terms of code generation. There are code generators but they are in general lacking the opportunity to share the knowledge and let the people cooperate as a community. The projects are more or less abandoned and their future is therefore highly unstable. The community cooperation is much more established in the world of web technologies as it can be seen in the blog post by Adam Bard comparing the amount of new repositories based on the language[9]. I see an opportunity here for research on the position of MDD in web technology, and how far one can go.

[9] http://adambard.com/blog/top-github-languages-2014/ - it is important to consider that CoffeeScript is Javascript. It is just a syntax sugar for JavaScript

Chapter 3

Research

This chapter shows the guidelines I have decided to follow and introduces the design science framework. There are seven guidelines in the framework and each guideline is commented.

3.1 Design Science

I have decided to follow the design science research framework published by Alan R. Hevner, Salvatore T. March, Jinsoo Park and Sudha Ram. This framework was published in 2004 and distinguishes between two different types of research in the discipline of information systems:

- behavioral science
- design science

Behavioral science ”seeks to develop and verify theories that explain or predict human or organizational behavior”(Hevner et al., 2004). Design science on the other hand ”seeks to extend the boundaries of human and organizational capabilities by creating a new and innovative artifacts”(Hevner et al., 2004). The following section introduces the framework's guidelines and gives a description on how the guidelines are fulfilled.

3.2 Guidelines

1. Design as an Artifact - *Design Science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.*

This report describes a code generation engine which has been also implemented. The code generator fills the gap between already known tools for developing REST APIs. It is also using some of the well established tools for testing, versioning and distributing

purposes. Other artifacts are two templates compatible with the code generator's metamodel- a template for REST API written in NodeJS and a template for generating valid API Blueprint.

2. Problem Relevance - *The objective of design-science research is to develop technology-based solutions to important and relevant business problems.*

There are several reasons why to pay attention to the development of REST APIs:

- REST APIs are needed when developing any kind of client-heavy application. It is a best practice derived from separating data and their representation.
- They are not very often the main focus in a project. It is just something which is needed to power the developed applications.
- It is crucial to follow the best practices and have an up-to-date documentation.

The problems, when working on a project including REST API and a client-side application, are as follows:

- The server-side programming is different from the client-side and introduces completely new challenges into the development.
- The development is a time consuming process. Time should be invested into what brings value: the client-side application.
- Introducing a new endpoint very often means several changes in different places in the code. These changes are done by copying existing code and making small edits like a different name of the database table, a different controller name etc.
- There is no easy way how to share best practices between developers. This can lead to poorly designed APIs which later make the client-side development costly.
- Ensuring an up-to-date documentation is again costly and in many cases it is omitted. This is a problem since the documentation is crucial for the clients communicating with the REST API.

This is a real problem and the fact that there are projects like Apiary which try to make the development process as smooth as possible with mocking APIs, providing a standardized form of documentation and helping developers to take the first steps when working with the REST API. According to my correspondence with Lukas Linhart, CTO of the project Apiary, they have reached 100.000 registered APIs. This proves that such services are high in demand.

3. Design Evaluation - *The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.*

The evaluation will be done based on the requirements in chapter 4.2. Each requirement shall have a separate section with all the evaluation details. The requirements are based on goals. A goal is considered to be achieved only if all the requirements are fulfilled.

4. Research Contributions - *Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.*

The contributions are summarized as:

- development of an artifact for generating production-ready REST APIs
- generation of a API Blueprint file based on the used notation for testing purposes and integration with already existing tools. The API Blueprint can be a part of any template set to ensure testability of the generated REST API.
- shortening the time needed to develop REST APIs respecting the best practices, including documentation and automated testing
- advanced templating system tailored to the needs of REST APIs
- research on best practices in the world of REST APIs

5. Research Rigor - *Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.*

The construction of the artifact is based on the research on what has been already done in order to identify the gap between the existing solutions. Finding the gap was an important process to establish the requirements. The evaluation of the software artifact is based on the requirements. There are separate chapters to show the evaluation and the discussion about how this report satisfies the research questions.

6. Design as a Search Process - *The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.*

This report is describing a second iteration over MDD of REST APIs. The first iteration was published in December 2014 and it is focused on establishing the correct notation and the implementation of an editor which supports this notation. The second iteration is described in this report. The main focus is to develop a generator capable of generating REST APIs based on a model with the notation described in the first iteration. The implementation shall be evaluated based on the requirements. The results shall be used to determine the focus of the future iterations.

7. Communication of Research - *Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.*

The whole research process is documented in this report. A technology-oriented audience can study the Implementation chapter which documents and explains all decisions that were made during the development phase. This chapter also includes all problems I ran into so it is easy to extend the research by choosing a different path or trying a different solution. The code is commented, fully open sourced and available on GitHub.

A management-oriented audience can benefit from the Theory chapter, where the basic principles are described. This can give a basic understanding of the technological part which can be beneficial when working on a related project.

I have also decided to send a proposal to three large JavaScript conferences to present my findings and the tool I have developed.

Chapter 4

Goals and Requirements

This chapter describes the goals and derived requirements for the developed solution. The goals are presented first and then I derive the requirements.

4.1 Goals

This section describes the goals. These goals have been established based on what has already been done in the field of Model-Driven Development, the best of modeling REST APIs and newly existing services which can be potentially helpful, but have not yet been connected to Model-Driven Development of REST APIs. The main goal has been established as *Model-based generation of REST APIs*. This goal has 6 sub-goals:

1. **Independence** - The generator shall not be tightly coupled to one meta-model. This will allow to reuse the existing notations described in 2.2. It generator shall also be able to generate any kind of programming language.

2. **Output Quality** - The generated code shall be as close as possible to a human written code including its structure. There shall be an automatic way of testing the generated code. This is because the programmer shall be able to immediately understand how the code is structured and mainly what has been generated and what needs to be manually added.

3. **Validity** - The generated REST API shall have the behavior based on best practice. The motivation for this is to increase the percentage of REST APIs with well designed and intuitive interface.

4. **Completeness** - The model shall be reflected in the generated REST API. This means that the code shall have all the endpoints specified in the model and the data structures shall be the same. Nothing shall be skipped so the user does not need to investigate what has been generated and what has not.

5. **Maintainability** - The generator shall be able to deal with the situation, when the user generates code, makes custom changes in the generated code, changes the model and then generates the code again. The custom changes shall not disappear. The motivation for this is that the underlying model might change as the development goes through different phases.

6. **Shareability** - The generator shall support sharing of the templates and reusing them without any manual download from the internet. The motivation is to distribute the work to experienced people. The templates shall be written by experienced people within the field of back-end development.

4.2 Requirements

This section describes the requirements derived from all the goals mentioned in the previous section.

4.2.1 Functional Requirements

This section shows the concrete requirements derived from each goal. The goal is considered to be satisfied only if all the derived requirements are fulfilled. The figure below shows the main goal and the sub-goals (green color) together with the derived requirements (blue color).

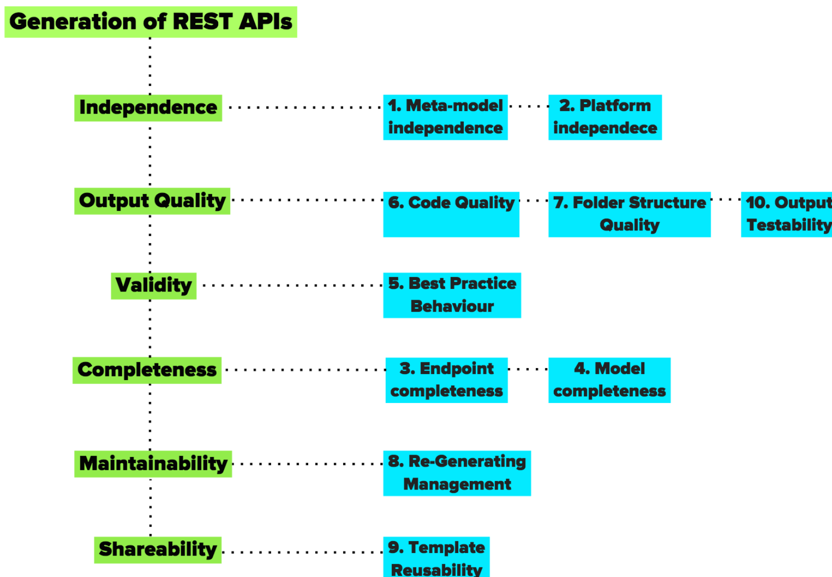

Figure 4.1: The goals and derived requirements

A more detailed description is given in the following table.

Name	Description	Priority
1. Meta-model independence	The generator shall be independent on the concrete meta-model	medium
2. Platform independence	The generator shall be able to generate any kind of code regardless of the programming language	high
3. Endpoint completeness	The generated endpoints shall reflect the modeled endpoints in the model	high
4. Model completeness	The data structures the generated REST API is working with shall reflect the entity structures in the model	high
5. Best Practice Behavior	The behavior of the generated REST API shall reflect the best practices of REST APIs	high
6. Code Quality	The code shall be structured as if human writes it	high
7. Folder Structure Quality	The folder structure of the REST API shall be the same as if a human structures it	medium
8. Re-Generating Management	The generator shall support custom changes in the code which shall not disappear when the generation process is run again	high
9. Template Re-usability	The generator shall support template sharing	low
10. Output Testability	The generated REST API shall be testable	medium

Table 4.1: Functional requirements

4.2.2 Non-Functional Requirements

There are several non-functional requirements which put constraints on the system's functionality. The non-functional requirements are shown in the table below.

Name	**Description**	**Priority**
N11. Cross-Platform	The generator shall work on OS X, Windows and Linux	medium
N12. Output Testability	The tests of the generated REST API shall be automatic	medium
N13. Template Sharing	The template sharing shall be done without any manual download	medium
N14. Speed	The generator shall generate the REST API in a reasonable time	medium

Table 4.2: Non-functional requirements

Chapter 5

Implementation

This chapter summarizes the technical decisions I have made and describes the complete implementation of the generator and the example template.

5.1 Architecture

I have decided to implement the generator as a REST API because of several reasons:

- The generator has a couple of options when generating the code. By separating these options into different endpoints I have achieved a cleaner implementation than if it would be for example an executable file which adjusts its behavior based on passed flags. Every endpoint has its pipeline of instructions and there are no branches in the code representing any exceptional behavior.
- The generator has a better public access as a REST API. Even tools like Google Closure Compiler, which is normally distributed as a *.jar* file, has its own server-side implementation and can be used by simply inserting the code to an online form in order to get the compiled version.
- It can be run as a localhost and easily modified or tailored to the special needs one can have. Modification of an executable file is impossible.
- Requesting a REST API is much more friendly to client-side developers than a potential executable script.

5.2 Technology

This section describes what kind of technology I have decided to use and why. The entire stack is based on JavaScript, open-source tools and libraries. This section also contains reasoning why I decided to not use Java.

5.2.1 NodeJS and NPM

NodeJS is "a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications"(Joyent, 2015). It was presented in 2009 and since that it has revolutionized server-side programming. There are several reasons why I decided to use this platform over another:

- NodeJS is using JavaScript as a primary language and the passed model to the generator is in JSON format. JSON is a shortcut for *JavaScript Object Notation*. According to Douglas Crockford, an author of many JavaScript books- "JSON is JavaScript without functions". This language compatibility gives a huge freedom when operating with the model.

- NodeJS is using NPM as a package manager. This makes it very easy to resolve all dependencies when including third party libraries for example for unit testing or pluralization. NPM has a command line tool so installing any dependencies is very easy.

- NodeJS has a huge active community and there are currently over 120,000 available packages developed by the community around JavaScript. Thanks to that, there is a huge chance that somebody has already written a module for a desired functionality and this module has been tested and improved by the programmers.

NodeJS module patterns

Since the implementation is based on NodeJS I followed some of the common patterns when writing modules for this environment. There are in total seven possible patterns one can follow. These patterns can be found in this presentation[1]. There are no rules on which pattern should be used when and it is highly dependent on the programmers taste. All of my modules are using the pattern number 6: Export an anonymous prototype. Every module is structured as a prototype with functions and properties and this prototype is then exported, so any other module can used the prototype under the specified name. I have chosen this pattern because it allows me to use inheritance and I believe that structuring modules as prototypes improves the readability of the code.

The only exception are configuration files. Configuration files are exported as pattern number 4 - Export an anonymous object. This pattern simply exports an object with attributes specifying the configuration.

5.2.2 MongoDB

MongoDB is "a schema-free document database written in C++ and developed in an open-source project"(Strauch, 2012). MongoDB is not the dependency of the generator itself but of the example template, which is tailored to generating NoSQL queries. There are several reasons why to use MongoDB:

[1] http://darrenderidder.github.io/talks/ModulePatterns/ - NodeJS Module Patterns - Using Simple Examples by Darren DeRidder

- MongoDB stores all the data in JSON format. Since the model is in JSON and the REST API shall return data in JSON data format, it is logical to store them in the same format.
- There are a lot of modules which allow a smooth cooperation between NodeJS and MongoDB.
- MongoDB does not rely on a fixed relational model. The documents are stored in collections and if there is an attempt to add a document to a non-existing collection, this collection is created on the fly. This very flexible behavior allows me to abstract from the traditional relation-based databases where the schema is a must.

The template is connected to the MongoDB database hosted on MongoLab[2]. MongoLab offers a free MongoDB-as-a-Service subscription so no installation is needed.

5.2.3 MochaJS

MochaJS is a "feature-rich JavaScript test framework running on node.js and the browser" (Holowaychuk, 2011). MochaJS is a standard between NodeJS testing frameworks, and it is well tested and accepted by the community. There was no doubt about MochaJS, because it is a simple, flexible and an easy to use testing framework. It is installed via NPM which fits into my development stack as I used NPM for other modules.

5.2.4 GruntJS

GruntJS is a JavaScript-based task runner which allows to define various tasks and then automatically trigger them on a certain event. This event can be for example a change in a file. GruntJS is a very popular and reliable tool with over 4,000 available plugins. Plugins are representing different tasks and these tasks can be chained in the Grunt's configuration file. Everything is manageable via NPM so it again fits into my development stack.

5.2.5 UnderscoreJS

UnderscoreJS is a JavaScript library that provides useful functional programming helpers without extending any built-in objects(Ashkenas, 2015). UnderscoreJS was officially released in 2009 and it is a very handy set of various functions which include the best practices in handling with objects and arrays. This library plays a huge role in the generator because it has a very simple templating function which has become the templating engine in the generator. There are several reasons why I decided to use UnderscoreJS as my templating engine:

- It allows to use the entire logic of JavaScript in the templates. There is no special templating language with a limited set of keywords, it is just pure JavaScript. This breaks the entry barrier most templating languages have.

[2] https://mongolab.com/

- It compiles templates in two steps. First, it creates a function from the template and then it executes the function on a certain object. The fact that the templating language is JavaScript and that the templates become functions allows to execute templates inside of other templates. The reason why this is so beneficial is explained in the separated chapter about templates.
- It is well tested library which has been available for over 6 years. The library's last version was released in August 2014 and it has over 13,000 stargazers on GitHub. This shows how popular, used and well tested the library is.

5.2.6 Dredd

Dredd is a very useful JavaScript tool which is used for testing REST APIs based on the passed API Blueprint. Dredd is a command line tool so it can be run from the terminal but it also has a GruntJS plugin so it can be automated by adding this task into the pipeline of plugins. This is a very handy functionality and the reason to include this tool is more than obvious: automatic testing of the generated code is crucial to be sure that the generation has been successful.

5.2.7 Git

Git is "a free and open source distributed version control system"(Git, 2015). I am using Git for versioning of the generator itself, but the generator also has some Git related behavior. The motivation for including Git and the behavior itself is described in section 5.3.10.

5.2.8 Why not Java?

Model-Driven Development is well established with Java programming language. The very common development stack includes Eclipse as an IDE and Eclipse Modeling Framework plugin all dependent on Java. This section therefore clarifies some of the reasoning why I decided to not follow the same path. According to W3Techs[3] Tomcat is the most popular Java server used. There are a couple of speed tests, like this one[4], which clearly shows that NodeJS is 20% faster than Tomcat. I believe that the speed should not be the main factor when developing a generator, but 20% is a huge difference for any kind of application. Another interesting comparison has been done by PayPal[5]. PayPal claims that the NodeJS application "double the requests per second vs. the Java application"(Harrell, 2013). There has been also "35% decrease in the average response time"(Harrell, 2013). Another very important reason is the fact that Java is being replaced by JavaScript as it can be seen in PayPal[6], but also in Norwegian BankID[7]. The new version of BankID is

[3] http://w3techs.com/technologies/overview/web_server/all - Usage of web servers for websites

[4] http://blog.shinetech.com/2013/10/22/performance-comparison-between-node-js-and-java-ee/ - Performance Comparison Between Node.js and Java EE For Reading JSON Data from CouchDB by Mark Fasel

[5] https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/

[6] https://www.youtube.com/watch?v=tZWGb0HU2QM - Clash of the Titans - Releasing the Kraken by Bill Scott

[7] https://www.bankid.no/ - BankID homepage

completely Java free due to the problems many users all over Norway have encountered.

Java definitely is an interesting piece of technology but I think I can get much closer to the targeted audience by using JavaScript. I personally believe that the ”one-platform focus” is the main reason why Model-Driven Development is not that widely spread in new companies and very often is not even considered as an option when deciding on the development method.

5.3 Generation Process

The process of generation is divided into several steps. The following figure shows the entire process.

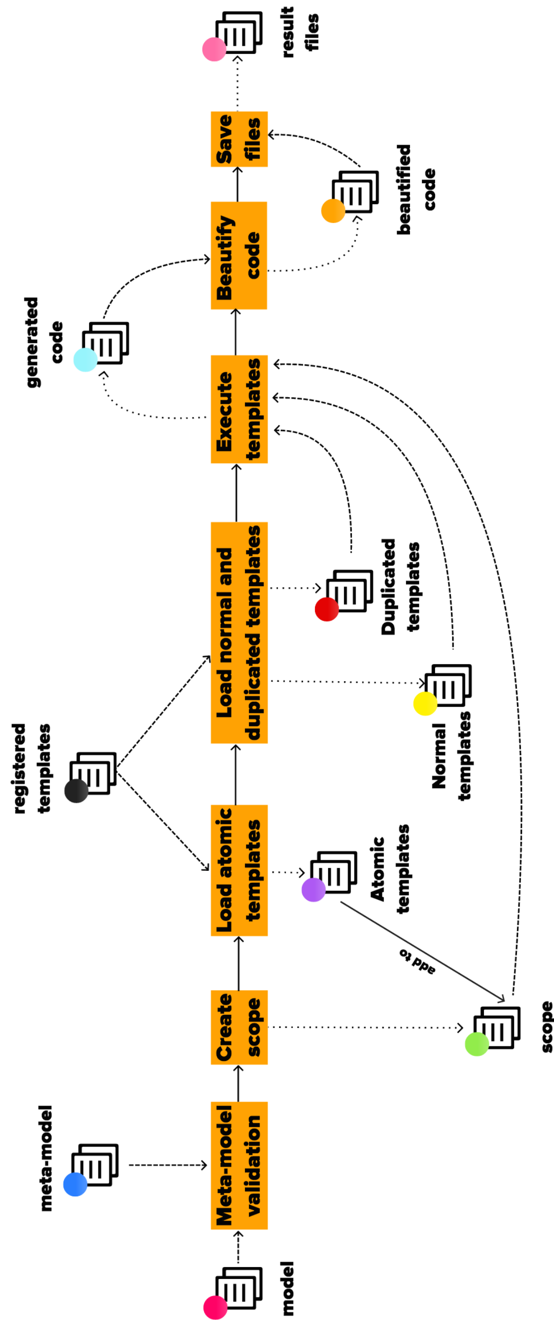

Figure 5.1: The generation process

The first step is a validation of the passed model against the generator's meta-model.

once the model is valid a scope is created. The passed model is attached to the scope. The scope is an object used when executing the templates and it makes sure that the model is accessible from the templates. The next step is to load atomic templates. These templates are used as a sub-templates and they are attached also to the scope so they can be accessed from the normal and duplicated templates. The normal and duplicated templates are loaded and executed by using the scope object. Next step is to beautify the result. This step is different for each file type. The last step is to make actual files out of the executed and beautified templates and store them.

The following sections describe each components in the process in more depth to clarify the implementation and behavior.

5.3.1 Meta-Model

The meta-model is located in *app/metamodel* and its only concern is to make sure that the passed model is valid and complete. It is a set of complex *if* statements. The generator is independent from the meta-model and the only function required is *valid()*. This function returns *true* if the model is valid and *false* if not. To make the meta-model flexible there is an abstract prototype *Abstract_Metamodel* which shall be extended in case of a new meta-model. The current meta-model checks the following characteristics of the model:

- Type checking of all parts of the model.
- Additional attributes of the model properties. This means for example that if a property is a string then it should have an additional property *regex* and *length*.
- All endpoints have one of the valid types specified in the meta-model.
- All associations are done between existing models and there is no association with a dead end.

The implementation is really straight forward and can be seen in the figure below. The *Abstract_Metamodel* prototype has a function *valid()*, which is overwritten in the concrete meta-model defining the model notation.

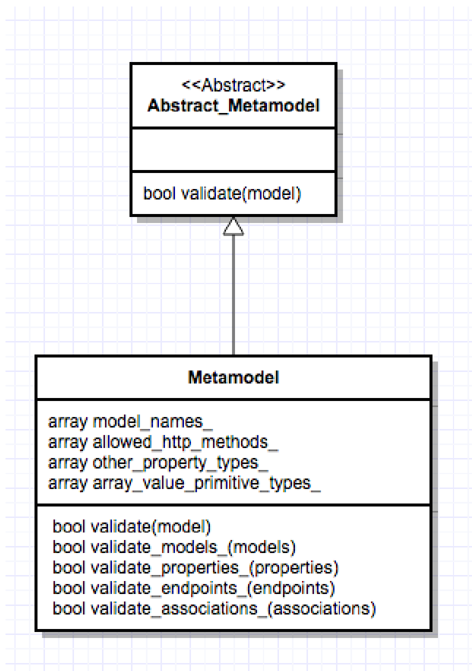

Figure 5.2: The implementation of the Meta-Model

The figure above shows that the meta-model has several properties defining allowed HTTP methods and primitive properties. The validation process is separated into smaller functions for better readability of the code. The functions with an underscore at the end of the name are considered to be private according to the Google Closure annotation rules[8].

5.3.2 Scope

Scope is an object passed to every template so the template can access the content and operate with it. Its implementation is located in *app/scope*. It contains:

- the model itself
- scope helpers
- atomic templates
- pluralization object
- data generation function

[8]https://developers.google.com/closure/compiler/docs/js-for-compiler - Annotating JavaScript for the Closure Compiler

Scope helpers are functions defined in *app/scope_helpers* and their purpose is simple: preserve a complex logic from the template. There are cases when the model needs to be altered in a certain way or when one specific entity from the model is needed. This logic should be kept in a separate scope helper function. This ensures that there are no long chunks of code in the templates and it also allows to reuse the logic in other templates. Since the logic is in a separate function it is possible to unit test it. The unit tests can be found in *test/scope_helpers.js*. The figure below shows the usage of a scope helper in a template (the left part of the picture) and its definition (the right part of the picture).

```
<%
  /**
  * format the endpoints so it is possible to loop over URLs and
  * get methods which shall be called
  */
  var formated_endpoints = scope.blueprint_friendly_models(model);
  var formated_keys = Object.keys(formated_endpoints); //get only
%>
```

```
/**
* function gets the model and returns the endpoints in a blueprint friendly
* way. This function is only used in the blueprint template
* @param {Object} model
* @return {Array}
*/
helpers['blueprint_friendly_models'] = function(model){
  var formated_endpoints = [];
  for(var i = 0; i < model.endpoints.length; i++){
    if(_.isUndefined(formated_endpoints[model.endpoints[i].url])){
      formated_endpoints[model.endpoints[i].url] = [];
    }
    formated_endpoints[model.endpoints[i].url].push(model.endpoints[i]);
  }
  return formated_endpoints;
}
```

Figure 5.3: The usage of a scope helper and its definition

Atomic templates are a very powerful feature of the generator. It allows to use templates inside of another template. The templating engine goes through two phases when executing the templates. First, it creates a function out of all templates and then it executes this function on the passed scope object. The atomic templates are loaded first. After they are loaded, the generator creates functions out of them and appends them to the scope object. A detailed description of the atomic templates is given in the next chapter.

The pluralization object is used for handling pluralization. I have used PluralizeJS library written by Blake Embrey[9]. This library has two main functions- *plural(singular)* which returns a pluralized noun of the passed singular noun and *singular(plural)* which returns a singular noun of the passed pluralized noun. This object solves the problem which can be seen for example in the Texo project where URIs do not follow the REST API standards because it uses the singular form.

The data generation function is based on the library RandExpJS written by Ben Burkert[10]. It is a simple function which requires a regular expression and it returns a string. This string fits to the passed regular expression. This functionality is required for the generation of random data in the API Blueprint file. More about API Blueprint is described later in the report.

5.3.3 Templates

Templates are the essential part of the code generation. The rule in general is- the better templates the better is the code. They are located in the *templates/* folder. The generator distinguishes between three different types of templates:

[9] https://github.com/blakeembrey/pluralize - PluralizeJS on GitHub
[10] https://github.com/benburkert/randexp - RandExpJS on GitHub

1. **Normal templates** - These templates has no special characteristics and are used most of the time.
2. **Atomic templates** - Atomic templates are loaded first and the template function is appended to the scope so these templates can be called from the normal and duplicated templates. The initial idea was to make them as a final templates but they can also be called from within another atomic templates. The first intention was to make these templates really atomic, but that would limit the possibilities. The name ”atomic” can be then a bit misleading.
3. **Duplicated templates** - Duplicated templates are specific for this particular domain. They are able to create multiple files based on the specified array of objects.

Duplicated templates are used for example for data models. Every data model (user, order, product etc.) has specific properties and all incoming data to the REST API should be validated by this data model to prevent harming attacks or data inconsistency. The standard is to use a single file per data model. This would be impossible to generate without knowing how many of the data models are in the model because one would have to create the same number of normal templates. Also the templates would all be the same which is against the MDD paradigm. Another solution is to dump all the data models into one single file but in case of a large model this file would really quickly become unmanageable. The solution to this is duplicated templates as they are able to create several files according to the rules specified in the configuration object. The following figure shows the template types in a schematic view.

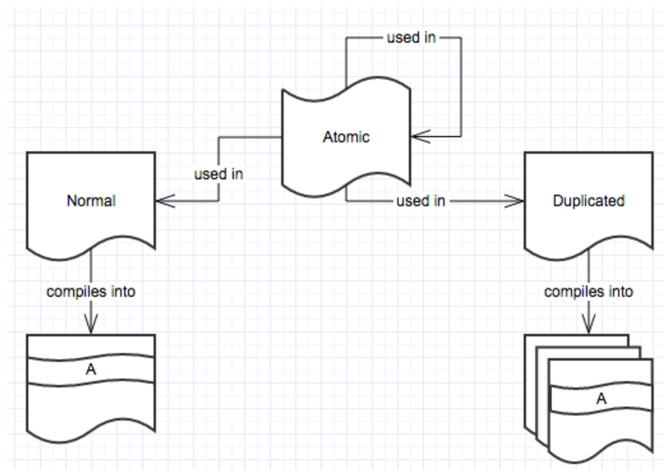

Figure 5.4: The template types

Every set of templates has a configuration file. This file must be located in the root directory of the templates and it contains references to all the template files in the set. The name of this file is *config.json* but this can be changed in the generator’s configuration file located in *config/config.js* parameter *TEMPLATE_CONFIG_FILE_NAME*. the configuration file has its own validator located in *app/template_config_validator*. This validator

makes sure that the configuration file is consistent and that all the parameters have only allowed values. It also checks the structure of the file in order to not break the entire generator. I was considering a file name notation as an alternative to the configuration file but there are several reasons why to prefer configuration over convention in this case:

- The configuration file is easy to extend. Any properties can be attached to the template files and modify the generation itself. This opens space for future extensions.
- The duplicated templates would not be possible because they require a configuration which would be impossible to squeeze in the file name.
- It is easier to experiment with the templates just by removing and adding templates into the configuration file, instead of manipulating the file system.

An example of the configuration file can be seen in the figure below.

```
{
  "name" : "NodeJS REST API template collection",
  "description" : "Template collection for creating NodeJS REST API",
  "templates" : {
    "normal" : [
      {
        "path" : "blueprint/blueprint.tmpl.apib"
      },
      {
        "path" : "blueprint/get.tmpl.js",
        "destination" : "example_files/"
      }
    ],
    "atomic" : [
      {
        "path" : "blueprint/random_json_builder.tmpl.apib"
      }
    ],
    "duplicated" : [
      {
        "path" : "api/app/routes/controller.tmpl.js",
        "scope" : "model.models",
        "reference" : "model",
        "name_property" : "name"
      }
    ]
  }
}
```

Figure 5.5: The template types

The header of the configuration file contains the name of the template set and the description. These two parameters are used for documentation purposes. The *templates* parameter contains three different arrays named after the template types. There are two normal templates registered in the figure. The second one has an additional parameter *destination*. Templates in general are executed and then stored in the same folder structure as they have in the template folder. The parameter *destination* allows to overwrite the native folder structure. There is one atomic template called *random_json_builder.tmpl.apib*. This template will be added to the scope as a template function. One then can call this function from templates in the following way *scope.random_json_builder()*. The most complex is

the setup of the duplicated templates. There is one duplicated template shown in the figure. The parameters are as follows:

- **path** - The path to the template file.
- **scope** - The array over which should be looped in order to create a file for each item in the array. The scope is a base for this variable. The value *model.models* means that there is a *model* property of the scope and the property has a property called *models*. The complete chain is translated as *scope.model.models*. Since *scope.model* is a reference to the passed model to the generator and *scope.model.models* is a reference to the array of entities defined in the model, this template will be used for each entity defined in the model. In case of more complex needs, one can create a scope helper function which creates the desired array and appends it to the scope. This function can then be called when the scope is created.
- **reference** - This parameter specifies how to access the item from the passed array. It is logical to make a singular form from the array name. Each item will be assigned to the scope under this name so one can access the item properties in the template by calling for example *model.name*.
- **name_property** - The last parameter specifies which property of the item will be used as a name of the file. In this case it is the *name* property so the generator will generate file names based on the *model.name* property.

The syntax of the templating language is very simple and it can be seen in the picture below.

```
var <%=model.name%>Controller = {

  <% _.each(model.endpoints, function(endpoint){ %>
    <% if(endpoint.type === "GET"){ %>
      <%=scope.set_GET_controller({model: model, endpoint: endpoint, scope: scope})%>
    <% } %>
```

Figure 5.6: The template snippet showing the template notation

The templating engine distinguishes between if one wants to interpolate a variable or not. In the first case the *<%= "Hello Wold" %>* notation shall be used. In the second case *<% var greeting = "Hello World" %>* shall be used. The templating engine allows to modify these marks if needed[11]. As it can be seen in the picture, the *model.name* is interpolated into the template so the variable will become for example *OrderController*. On the other hand the loop command *_.each(model.endpoints, function(endpoint){})* does not interpolate anything therefore the <%%> is used. Since the templating language is pure JavaScript, the native JavaScript functions can be used. This gives the templating language a high expressiveness.

5.3.4 Templates Load

All the templates are loaded based on the configuration file. The generator reads the configuration file and loads all the template files defined by the *path* parameter. The atomic

[11] http://underscorejs.org/#template - changing the templating marks

templates are loaded first as they are needed for the execution. The result of this step is a set of functions. Each function represents exactly one template file.

5.3.5 Templates Execution

This step takes the functions from the previous step and runs them with the scope passed as a parameter. The module responsible for the execution is located in *app/template_executor/*. The execution of the normal templates is simple since there is no special behavior needed. The Template_Executor just loops over all the template functions and executes them with the scope as a passed variable. The execution of the duplicated templates is a bit complex. The process is shown in the picture below.

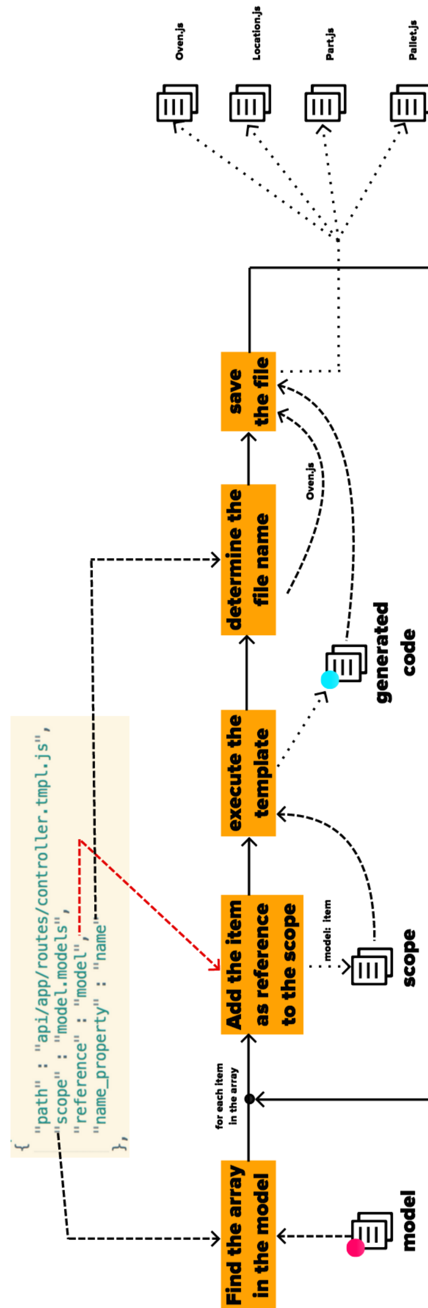

Figure 5.7: The processing of duplicated templates

The dashed connections are interpreted as *used by* while the dotted connections as

produces.The first step is to find the array in the model based on which the whole iteration is done. The implementation does at the moment not support any complex filtering (for example selecting every entity with the letter *S* at the beginning). The workaround for this can be a scope helper which saves the filtered data to the scope and is called when the scope is initialized. Once the array is known then the iteration over each item takes place. The picture shows a snippet from the template configuration file where the duplicated template is specified. The next step stores the item to the scope under the name specified in the *reference* parameter. The template is then executed with the scope enriched by the item. The next step is to determine the file names. This is done by looking in the template configuration file and checking the parameter *name_property*. This parameter contains the property of the item in the array. In case of the picture it is *item.name* which is used as a name. The determined name and the executed template are them used in the last step, where the real file is created. This repeats for each item in the array so the result are multiple files.

5.3.6 Code Beautification

The purpose of this step is to beautify the generated code and make sure that the formatting follows some standards. The module is located in *app/template_beautifier*. The tricky part of this step is that the effect is different depending on the used language. Since the REST API template is written in JavaScript the generator natively comes with the library called JS-Beautify[12]. This is a free open-source library available on GitHub and maintained by Liam Newman. This library has a function which receives the JavaScript code and re-formats the structure of the code so it fits to the passed options. Apart from the other options this library has, it also has an option for JSLint. JSLint is a JavaScript code quality tool which specifies how high quality code should be structured. Therefore I am able to say that the generated JavaScript code follows the standard as one would expect from a programmer.

It is a bit complicated with other languages, because the condition for a nice formatting is of course a tool or a library, which does this or a really precise template. Luckily the modern languages like Golang have their own tools to do that. Golang natively supports this with *go fmt* command[13] so no additional code is necessary. PHP also has its own formatter[14].

5.3.7 Templates Saver

This module simply takes all the executed and beautified templates and saves them into the result folder. This folder is specified in the generator's configuration file located in *config/* folder under the parameter *OUTPUT_DIR*. The module goes through all the templates and checks if there is the parameter *destination* defined. If so, the module saves the template into the path defined there. Otherwise it takes the native template path and creates the folder structure so it can be saved.

[12]https://github.com/beautify-web/js-beautify - JS-Beautify
[13]http://blog.golang.org/go-fmt-your-code
[14]http://beta.phpformatter.com/

5.3.8 Generator's Configuration File

As it has been already mentioned in the previous sections, the generator has its own configuration file and all the necessary configuration must be stored here and not hard-coded in the code. The configuration file is located in the *config/* folder and it acts like every other module, so in can be required inside the other modules. It currently contains:

- *TEMPLATE_DIR* - the name of the folder where the templates are located
- *OUTPUT_DIR* - the name of the folder where the generated code will be stored
- *TEMPLATE_CONFIG_FILE_NAME* - the name of the configuration file for templates
- *TEMPLATE_REPO* - a link to a Git repository from which the templates shall be cloned
- *TEMPLATE_CLONE_TARGET* - the name of the folder where the generator shall clone the templates from GitHub before compiling them
- *COMPRESSED_OUTPUT_FOLDER* - the name of the folder where the compressed generated code shall be stored

5.3.9 Generator's Endpoints

The generator has several endpoints, which can be used to generate the REST API or to generate example data and store them in the database. There are currently 5 available endpoints.

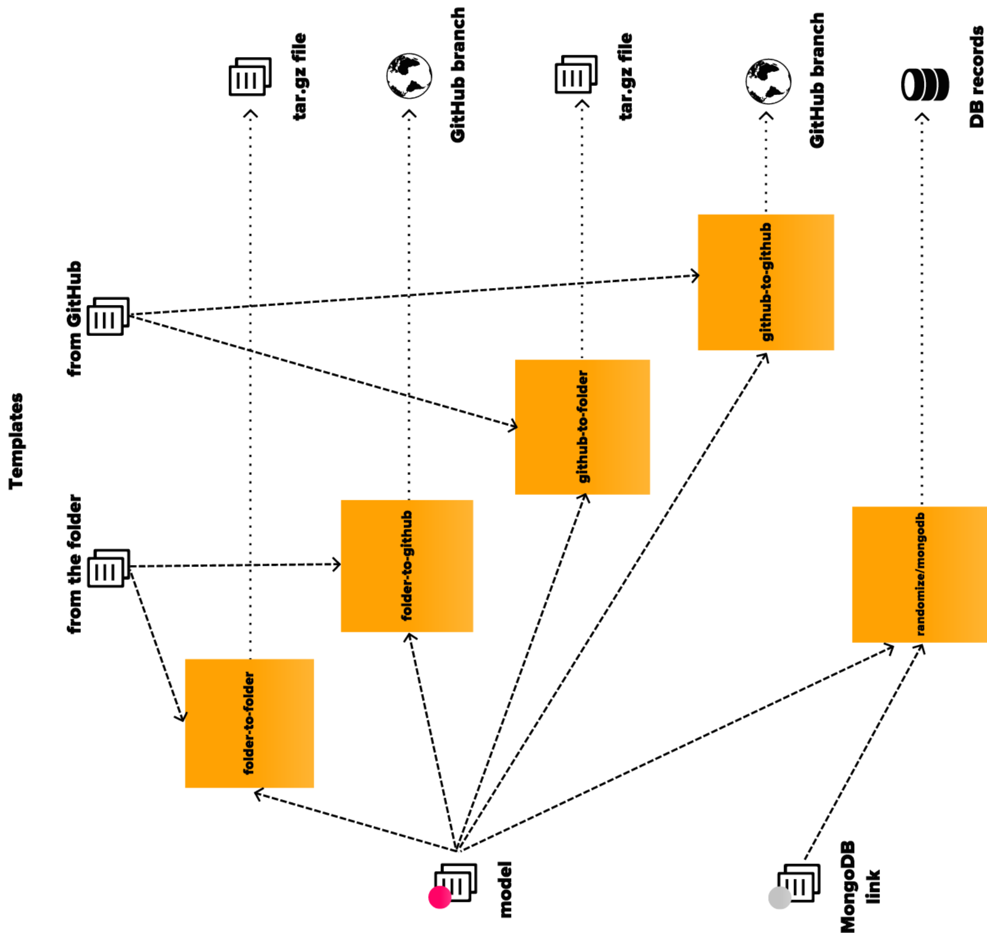

Figure 5.8: The endpoints of the generator, their inputs and the output

The figure shows all the available endpoints (orange squares). A dashed connection is interpreted as *used by* and the dotted as *the output*.

- *folder-to-folder* - This endpoint accepts the model and it uses the templates stored in the template directory on the server. It generates the code, stores it in the output folder, compresses this folder into a single file, stores this file in the compressed folder and then it serves the compressed file to the client. The client is then able to download this file.

- *folder-to-github* - This endpoint accepts the model and it uses the templates stored in the template directory on the server. It generates the code, stores it in the output folder and then it uploads the generated code via Git to GitHub or any other version management tool, for example Bitbucket.

- *github-to-folder* - This endpoint accepts the model and then it clones templates from a specified repository. Once the cloning is done, it generates the code, stores it in the output folder, compresses this folder into a single file, stores this file in the compressed folder and then it serves the compressed file to the client. The client is then able to download this file.
- *github-to-github* - This endpoint accepts the model and then it clones the templates from a specified repository. Once the cloning is done, it generates the code, stores it in the output folder and then it uploads the generated code via Git to GitHub or any other version management tool, for example Bitbucket.
- *randomize/mongodb* - This endpoint accepts the model and a MongoDB link, it generates random data according to the model and then stores the generated data in the specified database. The pluralized lower-cased name of the entities in the model is used as a collection name. The random data are also returned to the client in the JSON format.

5.3.10 Git Integration and Versioning

There are two endpoints that push the generated code to a GitHub repository at the end of the generation. There are three conditions for using this feature:

- Git has to be installed on the machine which is running the generator.
- There has to be a Git repository initiated in the output folder.
- The machine on which the generator is running has to have the credentials to the remote repository service (GitHub, Bitbucket etc.) set as a global settings. This means, that when pushing to a remote repository, Git should not require the credentials.

Once the conditions are met, the versioning to a remote repository can be used. Git is used in the generation process in two different scenarios.

- use a remote repository as a source of templates
- use a remote repository as a storage of the generated code

The motivation for the first case is obvious- reuse of the best practices. The repository can be accessed by many developers who collaborate and maintain the templates so it follows the best practices and evolves in time as the best practices change. The user of the code generator does not even need to know that something has changed as long as the tests for the generated code are passing. It is also a solution for generating for several platforms. If somebody creates templates for a certain language and puts it in a public repository, anybody can access it and reuse the developers knowledge.

The second case is mainly for managing the code base itself. As it has been already mentioned, the protected areas in the code are very important. Instead of introducing a new notation for marking the protected areas, I have transformed this problem into a

well known problem with many automated solutions- merging. When using an endpoint which automatically uploads the generated code, a new branch is created and this branch is then pushed to the remote repository. One can then open the repository in a repository management tool and merge the new branch with the master branch. This is helpful when one performs code generation, then makes some changes in the generated code and then for some reason (new entity, new property of an entity etc.) generates the code again. The changes done between the two generations would be lost, but with the versioning functionality one can simply decide what gets overwritten in the master branch and what stays there. Git has also an automated merging mechanism which in some cases is able to automatically figure this out.

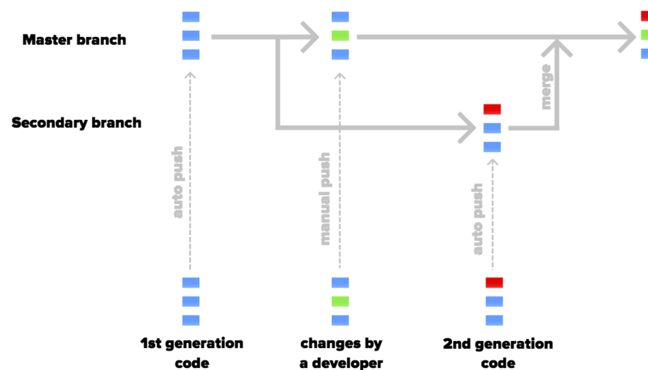

Figure 5.9: Merging process with custom code changes

The figure shows the merging process when a custom code change is done between the two code generations. The first generation is pushed into the repository. The programmer can then clone the repository and do the custom changes (marked as a green rectangle). Then he/she pushes the changes back to the repository. Suddenly a new endpoint needs to be added to the model and therefore a new code generation has to be done. This code will be identical as the one from the first code generation plus the new endpoint (marked as a red rectangle). When the programmer pushes the changes a new branch is automatically created and the code is pushed into this branch. The branch name is based on the date and the time of the generation. The programmer can then manually (or automatically if possible) merge the code including the custom changes with the code from the second generation and get the desired result (shown as a combination of blue, green and red rectangles).

Git is just a beginning. It is very popular for continuous integration[15] and Git is a great tool for this. Every time a new commit is registered in the Git repository, the code is loaded by the integration server, it passes several building steps including testing and then it is deployed to production.

[15]http://www.martinfowler.com/articles/continuousIntegration.html - Continuous Integration by Martin Fowler

5.3.11 Download, Installation, Run and Test

The code generator has been released as an open source project and is available online[16]. To download the code generator, simply clone the repository to your local machine by using *git clone https://github.com/RassaLibre/rest-api-generator.git*. These are the prerequisites to run the generator on a local machine:

- NodeJS with NPM
- GruntJS
- MochaJS

In order to download the dependencies simply run *npm install*. This will install all necessary dependencies the generator needs. To run the generator, simply run *node server.js*. This will start the generator on port 3000. The port can be changed in the *server.js* file. The generator is covered by unit tests. The unit tests can be run by executing the command *mocha -w*.

5.4 REST API template

The generator comes with a template for generating a REST API based on NodeJS and MongoDB. This template can predict the behavior of specified endpoints and has a couple of testing mechanisms in order to get a quick overview over what has been generated. This chapter describes the template and all the ideas implemented in it. The template is available on GitHub[17].

5.4.1 Folder Structure

The template has two folders in the root directory and the configuration file. The *api* folder contains all the templates related to the REST API. The *blueprint* folder contains templates related to the generation of API Blueprint. The purpose of API Blueprint will be described later in this chapter.

The REST API has a main file *api/server.templ.js*. This contains the registration of all endpoints and registers callbacks for handling the incoming requests on each endpoint. This template is registered as a normal template. The folder *api/test/* contains unit tests so the generated code is covered by unit tests. *api/app/Db.js* is a configuration file and it contains the database configuration. *api/app/Error_Map.js* is a module for handling any kind of errors which may occur when resolving the requests.

api/app/models/ is a folder containing data models. This folder contains the template *api/app/models/data_structure.tmpl.js*. This is a duplicated template and it is generated for each entity in the passed model. The generation of this template will result in multiple files each named after an entity reflecting the data structure defined in the passed model. The generated data structures inherit the behavior from *api/app/modes/model.js*.

[16] https://github.com/RassaLibre/rest-api-generator - The code generation repository

[17] https://github.com/RassaLibre/rest-api-generator-templates - The template available for code generation

api/app/models/field_properties.tmpl.js is a simple atomic template used when generating the data structures and its purpose is to manage whether the variable is shown in quotes or not. This is necessary for distinguishing a string variable from integers. The last file *api/app/models/index.tmpl.js* is used as an export of the entire *Models* module.

api/app/nested/ is a folder containing functions for easy handling of the nested objects inside the structures returned from the database. They are all registered as normal templates so the generator places them in the target folder. *api/app/db_query_builder/* folder contains a module which allows filtering and sorting the results. This is possible via additional parameters passed with the URI. This module is covered by unit tests which can be found in *api/test/db_query_builder.js*.

api/app/routes/ is a folder containing all the behavior for resolving incoming requests. *api/app/routes/controller.tmpl.js* is a duplicated template generated for each entity in the passed model. The generation of this template results in multiple files named after each entity in the passed model. Each file contains the logic behind handling a particular request on a particular resource/entity. *app/api/routes/index.tmpl.js* is a normal template exporting the entire module so it can be reached in the *api/server.tmpl.js* file. *app/api/routes/query_selector.tmpl.js* is a simple atomic template which is used to determine how the query to the database should look like. The remaining files *app/api/routes/set_DELETE_controller.tmpl.js*, *app/api/routes/set_GET_controller.tmpl.js*, *app/api/routes/set_POST_controller.tmpl.js* and *app/api/routes/set_PUT_controller.tmpl.js* are atomic templates containing the behavior for handling requests. These templates are called in *api/app/routes/controller.tmpl.js* based on which method is used in the current request.

The *blueprint* folder contains the main template *blueprint/blueprint.tmpl.apib*. *blueprint/endpoint_parameters.tmpl.apib* is an atomic template for inserting a common construction for endpoints with parameters. *blueprint/random_json_builder.tmpl.apib* is a template for generating random data based on the passed entity in the model. The remaining templates *blueprint/array_GET_request.tmpl.apib*, *blueprint/DELETE_request.tmpl.apib*, *blueprint/POST_request.tmpl.apib*, *blueprint/PUT_request.tmpl.apib* and *blueprint/single_GET_request.tmpl.apib* are atomic templates which contain the content of the API Blueprint file based on the HTTP method of each endpoint. These templates are called in the *blueprint/blueprint.tmpl.apib* file.

5.4.2 Filtering and Sorting

The template supports filtering and sorting of the results. This can be achieved by passing additional parameters with the URI for example *ovens?SKU=AA-AA-AA&orderBy=price&orderDirection=desc*. This means that the result will contain only ovens with the property *SKU* equal to *AA-AA-AA*. The result will be ordered by the parameter *price* in the descending order (from the highest price to the lowest price). The logic behind this functionality is very simple. The additional parameters are compared to the entity's properties and if there is a match, the parameter is considered to be a filter. The parameters which do not match any property are simply ignored. *orderBy* and *orderDirection* are special keywords which are used for the ordering and therefore should not be used as a name for a property in any entity.

The filtering is dependent on the property type. If the property is of type *string* then the database query contains a regular expression with the passed value. If the property is of type *integer* then the passed value is parsed into an *integer* so it can be used in the database query. A special case are dates. In that case, the passed value shall start with the $<$ or $>$ sign. This indicates if the user is searching for the documents with a date younger or older than the passed value.

5.4.3 Behavior prediction

The template has build-in behavior prediction. This means that the template is able to recognize what each endpoint shall do. The template currently supports all major HTTP methods- GET, POST, PUT, DELETE. The following types of endpoints are supported:

- endpoints with a noun at the end- for example */ovens* or */ovens/:id/parts*
- endpoints with a parameter at the end- for example */ovens/:id* or */ovens/:id/parts/:id*
- endpoints with a verb at the end - for example */ovens/:id/move*

The template first figures out which method is used and then it matches the URI with a regular expression. If the match is found the template knows which part of the predefined code shall be used. If the URI matches the format *noun1/parameter/noun2* it figures out the entity related to *noun1* and checks, if this entity contains a field in the data structure with the same name as *noun2*. If so, the template classifies the endpoints as an endpoint with a noun at the end. If the fields is not in the data structure, the endpoint is classified as an endpoint with a verb at the end. The process is shown in the figure below.

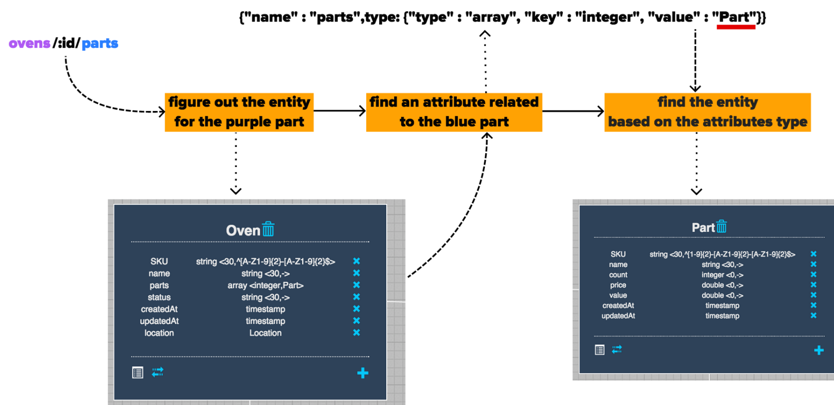

Figure 5.10: Template behavior prediction

The first matching is done based on the first noun (shown as purple in the figure) of the inspected URI and the entities in the passed model. According to the best practice when creating URIs it is good to use lowercase letters and the plural form. The entities shall

always be in the singular form and start with a capital letter. This is not a problem because PluralizeJS takes care of the transition between the plural and the singular version. Capital letters are also not a problem because JavaScript has native functions converting uppercase to lowercase. The second matching is done based on the second noun (shown as blue in the figure) and the name of the property. In this case the best practices for naming are the same so no transformation is needed. Parameters in the URI and the attributes in the data structure should consist of lowercase letters and should both be in a form based on the data type- plural for array, singular for everything else. Since the data type is an *array* the form shall be plural.

The third matching is the easiest one because the attribute's type shall reflect the entity's name so no transitions are needed. If this process fails in any step a general template is applied. This means that the endpoint is not forgotten, it still has its place in the generated code but its implementation is blank and it is up to the programmer to fill in the implementation because the template is not able to predict the behavior of this endpoint. The same counts for the endpoints with the verb at the end. It is impossible to predict any sort of behavior in that case. All the matching functions are stored as scope helpers so they can be covered by unit tests and reused.

5.4.4 API Blueprint

As it has been already mentioned, the second part of the template set is a template for generating an API Blueprint file. This file contains the specification of the generated REST API including all endpoints, URI parameters and incoming and outcoming data. There are several reasons why I have included this file:

- API Blueprint is a well known and standardized notation for describing REST APIs. There are several tools available for this notation[18]. These tools are developed by the community around the API Blueprint and its main purpose is to ease the REST API development. By generating this file I made sure that if there will be an interesting tool developed in the future, the generated APIs will support it.

- The Apiary project is generating documentation based on this file. This means that every generated API shall be provided with documentation for each endpoint. This, in addition to the visual model creates a powerful base for the documentation.

- Apiary offers SDKs for any platform to make an HTTP request to the server. This is very useful for anybody who has never worked with REST APIs and can dramatically reduce the problems one has to overcome.

5.5 Template Development Workflow

This section is about the template development process and the mechanisms the generator has in order to ease this process.

[18]https://apiblueprint.org/

5.5.1 Prerequisites

The prerequisites are that the user has the following packages installed:

- **GruntJS** for file watching the templates. This is run by the command *grunt development* in the root folder of the generator.
- **MochaJS** for running the unit tests. This is run by the command *mocha -w* in the folder with the generated REST API. The *-w* ensures that the tests will re-run on every file change.
- **NodemonJS** for file watching the generated API as a whole. This is run by the command *nodemon server.js* in the folder with the generated REST API.

I recommend four different terminal windows. One is the generator, one for GruntJS, one for MochaJS and the last one for running the generated REST API via NodemonJS. This setup is shown in the following figure.

Figure 5.11: The recommended setup for template development

5.5.2 Template Development Process

The development process is shown in the figure below. The whole chain of processes is triggered once the user makes a change in the templates. This change is registered by the GruntJS task and it grabs the example model located in *test/example_data.js*. The same model is used for the unit tests of the generator. This model is sent to the generator via HTTP request as the editor would do. The generator accepts the model and uses the freshly edited templates to generate the code base of the REST API. The generated code overwrites the previously generated REST API which is running via NodemonJS. The overwriting triggers NodemonJS to restart the generated REST API so the changes made in the templates are immediately available. The overwriting also triggers the unit test to re-run themselves. The described process is shown in the figure below.

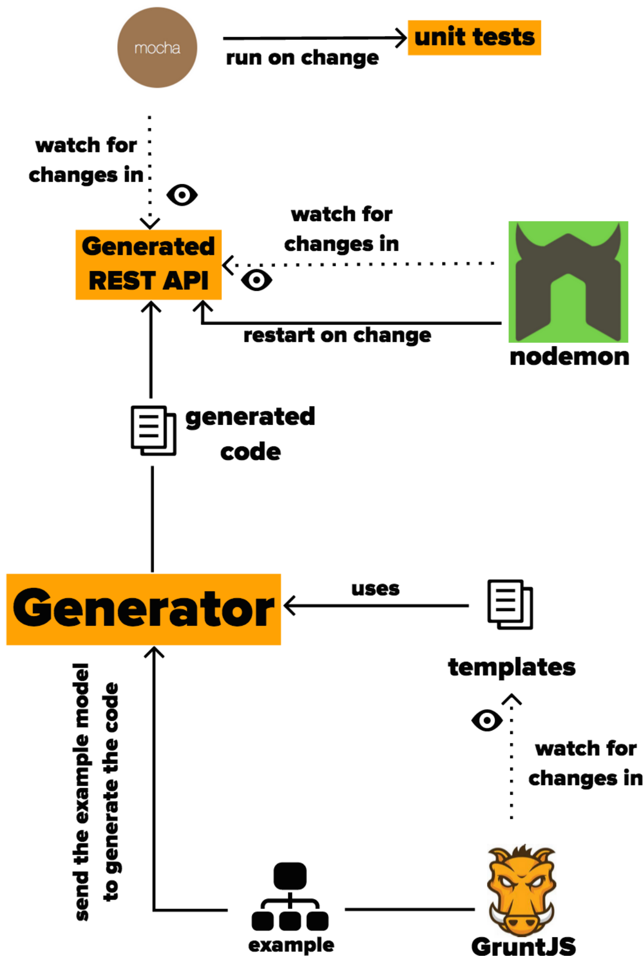

Figure 5.12: The template development process (Source of GruntJS logo:`http://gruntjs.com/`, MochaJS logo: `http://mochajs.org/`, NodemonJS logo:`http://nodemon.io/`)

By following this process, one can get a lot of advantages. It is mainly the fact that the generated REST API gets constantly updated with every saved change in the templates. To check that the changes are correct and do what is expected, the unit tests of the generated REST API are re-run together with the generation. One can then clearly see the impact of the changes. This together with Dredd creates a very strong foundation for the template development.

5.6 The Editor Extensions

In order to support all the functionality of the generator in the editor, I had to extend the editor. This section therefore describes the extension I had to make.

5.6.1 Endpoint handling

There are different endpoints the user can choose between. This resulted in a component with different radio boxes one for each endpoint. A button for firing requests to the generator has been also implemented. If the user selects one of the endpoints which returns the REST API as a compressed file, the file is automatically served after the generation and the browser downloads it. If the user selects one of the endpoints which results in an interaction with GitHub a success message is shown after the interaction. If the user selects ”Data generator” an extra text field is shown because the user needs to enter the MongoDB link to the database. The picture below shows the implemented extensions in the control panel.

Figure 5.13: The implemented extensions in the editor

Chapter 6

Evaluation

This chapter describes the evaluation of the developed software. The evaluation is based on the requirements specified in chapter 4.2. The study case is described in appendix A. The model can be found here [1]. Some of the evaluation points are comparing the actual output to a programmer written REST API. The programmer written REST API is based on one of the production REST APIs developed by Searis AS[2]. The company is focusing on software integration and REST APIs are included in all the projects Searis AS has been working on.

6.1 Functional Requirements

This section shows the evaluation of the functional requirements against the actual implementation.

6.1.1 Meta-Model Independence

Name	Description	Priority
1. Meta-model independence	The generator shall be independent on the concrete meta-model	medium

The concrete meta-model, which fits to the notation the editor is generating is implemented as a child of the prototype *Abstract_Metamodel*. The only dependency the generator has is the *validate(model)* function which accepts the model and returns a boolean variable weather the model is valid or not. This function is in the prototype *Abstract_Metamodel* so if a new meta-model needs to be created, it simply needs to inherit the *Abstract_Metamodel*

[1] http://tdt4501.bitballoon.com/ - The model of the case used to evaluate the result
[2] http://searis.no/ - Searis AS homepage

prototype and then overwrite the *validate(model)* function with custom rules reflecting the new notation.

Since the only dependency is expressed in the abstract prototype it is therefore possible to use custom meta-models, which means that the generator **is not dependent** on the concrete meta-model.

6.1.2 Platform Independence

Name	Description	Priority
2. Platform independence	The generator shall be able to generate any kind of code regardless of the programming language	high

The example template is written in JavaScript and is targeting the NodeJS environment. This is not the generators constraint because it can be used for any kind of language. The code snippet below shows a very simple PHP template.

```
<?php
  $server = new Server();
  <% _.each(scope.model.models, function(model){ %>
    $server->register_model("<%=model.name%>");
  <% }) %>
?>
```

The execution of the template results in the following PHP code.

```
<?php
  $server = new Server();
  $server->register_model("Oven");
  $server->register_model("Pallet");
  $server->register_model("Location");
  $server->register_model("Part");
?>
```

A bit more complex language such as Golang is also possible. Golang is a programming language developed and maintained by Google and its popularity is raising[3]. The following code snippet shows a very simple template for Golang code.

```
<% _.each(scope.model.models, function(model){ %>
  func hello</%=model.name%>(w http.ResponseWriter, r *http.Request){
    io.WriteString(w, "<%=model.name%>")
  }
<% }) %>
```

The execution of the template results in the following code.

```
func helloOven(w http.ResponseWriter, r *http.Request){
  io.WriteString(w, "Oven")
}
```

[3]http://herman.asia/the-popularity-of-go - The popularity of Go by Herman Schaaf

```
func helloPart(w http.ResponseWriter, r *http.Request){
  io.WriteString(w, "Part")
}

func helloPallet(w http.ResponseWriter, r *http.Request){
  io.WriteString(w, "Pallet")
}

func helloLocation(w http.ResponseWriter, r *http.Request){
  io.WriteString(w, "Location")
}
```

It is obvious that the generator is not limited to a certain programming language. It can even generate files which are not related to the code at all. The perfect example is the API Blueprint file which has its own notation based on Markdown notation. The generator is therefore platform independent and **can generate** any kind of programming language.

6.1.3 Endpoint Completeness

Name	**Description**	**Priority**
3. Endpoint completeness	The generated endpoints shall reflect the modeled endpoints in the model	high

The model from the use case contains 27 endpoints in total as can be seen in the picture below.

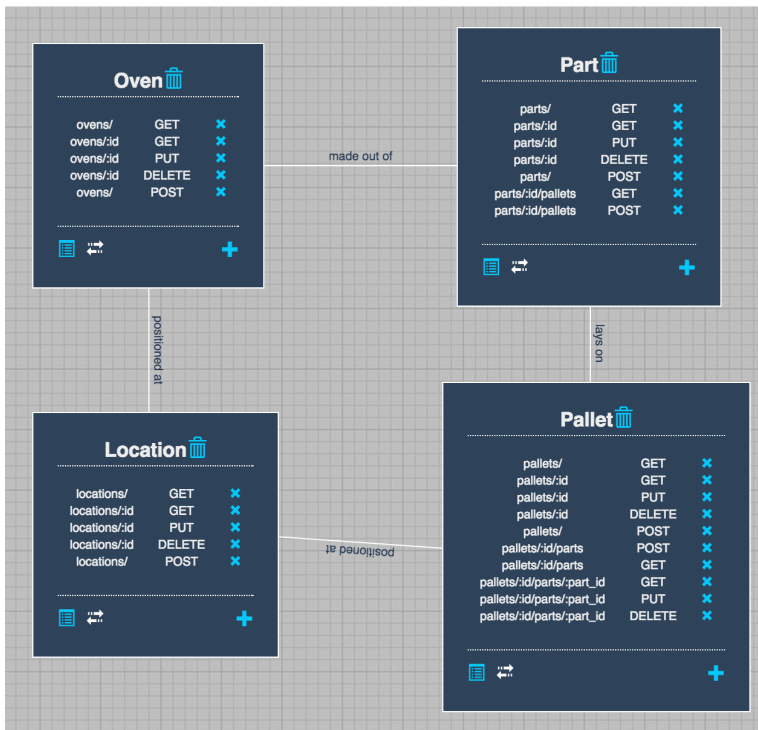

Figure 6.1: The modeled endpoints in the use case

The generated REST API also contains 27 endpoints in total as it can be seen in the code snippet below.

```
exp.get('/ovens/', routes.oven.get_ovens_w);

exp.get('/ovens/:id', routes.oven.get_ovens_x);

exp.put('/ovens/:id', routes.oven.put_ovens_y);

exp.delete('/ovens/:id', routes.oven.delete_ovens_z);

exp.post('/ovens/', routes.oven.post_ovens_10);

exp.get('/pallets/', routes.pallet.get_pallets_11);

exp.get('/pallets/:id', routes.pallet.get_pallets_12);

exp.put('/pallets/:id', routes.pallet.put_pallets_13);

exp.delete('/pallets/:id', routes.pallet.delete_pallets_14);

exp.post('/pallets/', routes.pallet.post_pallets_15);

exp.post('/pallets/:id/parts', routes.pallet.post_parts_in_pallets_16);
```

```
exp.get('/pallets/:id/parts', routes.pallet.get_parts_in_pallets_17);

exp.get('/pallets/:id/parts/:part_id', routes.pallet.
    get_parts_in_pallets_18);

exp.put('/pallets/:id/parts/:part_id', routes.pallet.
    put_parts_in_pallets_19);

exp.delete('/pallets/:id/parts/:part_id', routes.pallet.
    delete_parts_in_pallets_1a);

exp.get('/locations/', routes.location.get_locations_1b);

exp.get('/locations/:id', routes.location.get_locations_1c);

exp.put('/locations/:id', routes.location.put_locations_1d);

exp.delete('/locations/:id', routes.location.delete_locations_1e);

exp.post('/locations/', routes.location.post_locations_1f);

exp.get('/parts/', routes.part.get_parts_1g);

exp.get('/parts/:id', routes.part.get_parts_1h);

exp.put('/parts/:id', routes.part.put_parts_1i);

exp.delete('/parts/:id', routes.part.delete_parts_1j);

exp.post('/parts/', routes.part.post_parts_1k);

exp.get('/parts/:id/pallets', routes.part.get_pallets_in_parts_1l);

exp.post('/parts/:id/pallets', routes.part.post_pallets_in_parts_1m);
```

The following table shows the modeled endpoints in detail and each endpoint has a flag whether there is a generated endpoint which matches or not. The table is based on comparing the two figures above.

URI	Method	Is generated?
ovens/	GET	Yes
ovens/:id	GET	Yes
ovens/:id	PUT	Yes
ovens/:id	DELETE	Yes
ovens/	POST	Yes
locations/	GET	Yes
locations/:id	GET	Yes
locations/:id	PUT	Yes
locations/:id	DELETE	Yes
locations/	POST	Yes
parts/	GET	Yes
parts/:id	GET	Yes
parts/:id	PUT	Yes
parts/:id	DELETE	Yes
parts/	POST	Yes
parts/:id/pallets	GET	Yes
parts/:id/pallets	POST	Yes
pallets/	GET	Yes
pallets/:id	GET	Yes
pallets/:id	PUT	Yes
pallets/:id	DELETE	Yes
pallets/	POST	Yes
pallets/:id/parts	POST	Yes
pallets/:id/parts	GET	Yes
pallets/:id/parts/:part_id	GET	Yes
pallets/:id/parts/:part_id	PUT	Yes
pallets/:id/parts/:part_id	DELETE	Yes

Table 6.1: Comparison of the modeled endpoints with the generated ones

As can be seen in the table, the generator **does** generate all the endpoints with the correct URI and the HTTP method therefore the requirement is satisfied.

6.1.4 Model Completeness

Name	Description	Priority
4. Model completeness	The data structures the generated REST API is working with shall reflect the entity structures in the model	high

For this evaluation I will use just one entity, because all the entities are compiled with the same algorithm. If one is correct the others are too. The entity *Oven* seems to be the most complex one, since it includes the most parameters of different types. The entity is shown in the figure below.

Figure 6.2: The entity Oven in the use case model

The generated data model for this entity is shown in code snippet below.

```
this.fields = {

  SKU: {
    type: 'string',
    length: 30,
    regex: '^[A-Z1-9]{2}-[A-Z1-9]{2}$',
  },

  name: {
    type: 'string',
    length: 30,
  },

  parts: {
    type: 'array',
    key: 'integer',
    value: 'Part',
  },

  status: {
    type: 'string',
    length: 30,
  },

  createdAt: {
    type: 'timestamp',

  },

  updatedAt: {
    type: 'timestamp',
```

```
  },

  location: {
    type: 'Location',

  },

};
```

By comparing the two figures, one can see that the modeled properties of the entity Oven are exactly the same as the one in the generated data structure. It is also possible to observe that all additional fields (min, max, regex etc.) dependent on the main type (integer, string etc.) are also present in the structure. If the field is not filled, then it is simply not in the data structure.

Based on that, it is obvious that the modeled data structure **is reflected** in the data structure the generated REST API is operating with. To be sure that this is correct, lets compare the figures to the output of the generated REST API when the endpoint GET *ovens/:id* is called. The figure below shows the data when such a call is performed.

```
– {
     "_id": "54d409c2b14bec343fd6799d",
     "SKU": "U4-Q1",
     "name": "VZMKCNUKEICACERDKPJJHIVVBYONOK",
   + "parts": [ … ],
     "status": "JFZICYSHDZCHIBNPRDAQFDONTMPCNO",
     "createdAt": 1423182265437,
     "updatedAt": 1423182265437,
   + "location": { … }
  },
```

Figure 6.3: The actual object of type Oven returned from the generated REST API

As can be seen, the structure of the object is the same, but there is an additional property *_id*. This is a unique identifier of the object and it has been said in the report related to the editor that "this parameter therefore shall not be modeled because it does not really describe the reality, it is an unique identifier used for identifying each resource"(Prochazka, 2014). The *_id* parameter is therefore completely valid here.

Based on the comparison of the modeled structure, generated structure and an actual structure of a real object from the generated REST API it is obvious that this requirement is satisfied.

6.1.5 Best Practice Behavior

Name	Description	Priority
5. Best Practice Behavior	The behavior of the generated REST API shall reflect the best practices of REST APIs	high

The best practices have been established in chapter 2.1.2. The chapter contains the table with defined behavior of each HTTP methods related to the URI. These practices were implemented in the API Blueprint. Dredd, the software that uses the API Blueprint to actually test the generated REST API, checks:

- the structure of incoming data
- the structure of outcoming data
- the type of each property (integer, string, array, etc.)

The result of the Dredd testing can be seen in the following picture.

```
pass: GET /ovens duration: 1549ms
pass: POST /ovens duration: 703ms
pass: GET /ovens/54d10585e4b00a11b540d01b duration: 726ms
pass: PUT /ovens/54d10585e4b00a11b540d01b duration: 839ms
pass: DELETE /ovens/54d10585e4b00a11b540d01b duration: 10ms
pass: GET /pallets duration: 747ms
pass: POST /pallets duration: 990ms
pass: GET /pallets/54d0bf38e9cc9063148461e5 duration: 3913ms
pass: PUT /pallets/54d0bf38e9cc9063148461e5 duration: 1375ms
pass: DELETE /pallets/54d0bf38e9cc9063148461e5 duration: 9ms
pass: POST /pallets/54d0bf38e9cc9063148461e5/parts duration: 1094ms
pass: GET /pallets/54d0bf38e9cc9063148461e5/parts duration: 701ms
pass: GET /pallets/54d0bf38e9cc9063148461e5/parts/54d3e26b09d8c002113b0c6e duration: 849ms
pass: PUT /pallets/54d0bf38e9cc9063148461e5/parts/54d3e26b09d8c002113b0c6e duration: 3274ms
pass: DELETE /pallets/54d0bf38e9cc9063148461e5/parts/54d3e26b09d8c002113b0c6e duration: 9ms
pass: GET /locations duration: 2314ms
pass: POST /locations duration: 1822ms
pass: GET /locations/54d10385e4b00a11b540cfff duration: 706ms
pass: PUT /locations/54d10385e4b00a11b540cfff duration: 659ms
pass: DELETE /locations/54d10385e4b00a11b540cfff duration: 6ms
pass: GET /parts duration: 763ms
pass: POST /parts duration: 1359ms
pass: GET /parts/54d3d1d4a554e10f0be08900 duration: 880ms
pass: PUT /parts/54d3d1d4a554e10f0be08900 duration: 3383ms
pass: DELETE /parts/54d3d1d4a554e10f0be08900 duration: 6ms
fail: GET /parts/54d3d1d4a554e10f0be08900/pallets duration: 10ms
fail: POST /parts/54d3d1d4a554e10f0be08900/pallets duration: 7ms
```

Figure 6.4: Best practices checked with Dredd

As can be seen, all the endpoints are passing except two. These two are exceptional because they do not fit to the passed model and require the programmer to fill in the implementation. The following table is the same table as the one in the chapter 2.1.2 but it shows which URI type and HTTP method combinations were tested and what is the result.

As can be seen in the table, there are two combinations which where not tested because there are not covered in the use case, and four combinations are skipped because they do not make sense. All the other combinations are tested and the result is positive.

xxx	*/ovens*	*/ovens/233*	*/ovens/293/parts*	*/ovens/293/parts/676*
GET	**passed**	**passed**	**passed**	**passed**
POST	**passed**	XXX	**passed**	XXX
PUT	*not tested*	**passed**	XXX	**passed**
DELETE	*not tested*	**passed**	XXX	**passed**

Table 6.2: Expected behavior related to the Dredd test results

It is important to mention that the behavior of the endpoints is completely dependent on the templates and not on the generator itself. The generator just executes the templates. The best practice knowledge is hidden in the templates.

Based on the comparison table I am able to say that the generator **is capable of** generating REST APIs with the best practice behavior specified in the chapter 2.1.2. The requirement is therefore satisfied.

6.1.6 Code Quality

Name	**Description**	**Priority**
6. Code Quality	The code shall be structured as if a programmer writes it	high

To measure that the generated code has the same quality as a programmer written code, I will compare the generated code to the hand written code. The following code snippet shows the controller, which returns all objects type of *Parts*.

```
/**
* returns list of parts
*/
getParts: function(req, res){
  mongo_client.connect(mongo_url, function(err, db) {
    if(err) error_handler.send_error(res, 102);
    var collection = db.collection('parts');
    var query_string = db_query_builder.build_db_query(req.query, new
        model.Part().fields);
    collection.find(query_string).toArray(function(err, docs){
      if(err) error_handler.send_error(res, 100);
      res.send(docs);
      db.close();
    });
  });
},
```

The controller accepts a request from the client, executes a database query based on what kind of parameters are attached to the URI and returns the result. The code snippet below shows how the same part looks like when the code is generated.

```
/**
```

```
 * returns list of Parts
 */
get_parts_1g: function (req, res) {
  mongo_client.connect(mongo_url, function (err, db) {
    if (err) error_handler.send_error(res, 102);
    var collection = db.collection('parts');
    var query_string = db_query_builder.build_db_query(req.query, new
        model.Part().fields);
    collection.find(query_string).toArray(function (err, docs) {
      if (err) error_handler.send_error(res, 100);
      res.send(docs);
      db.close();
    });
  });
},
```

As it can be seen, there are some minor differences. In the generated code, the name of the function is *get_parts_1g*. The fact that the original is camel-cased is not a problem, this is just a matter of the notation but the *1g* in the name of the generated function is extra. It is an unique ID in the model and it is there to ensure that the name of the controller is unique. It is a great source of uniqueness and it helps to quickly search for the one endpoint the controller is handling. Apart from that, the snippets are identical.

It is important to understand that the generator itself has a small impact on how the resulted code will look like. It is the templates that can affect the final look. The generator has a beautifier but that is currently available only for JavaScript files. However modern languages have their own style fixers. The reason why the fixers for, for example Golang, PHP or Ruby are not included in the generator is that there is no NPM module which would allow to do that. Therefore this needs to be a separate action after the generation process.

There are cases when the generated code does not look like the hand written one at all. The problem is that the programmer structures the code so it is aesthetically nice for the eye. This does not always match with the best practices included in the beautifier. The following code snippet shows defined fields in a hand written REST API.

```
this.fields = {
  SKU: {type: 'string', length: 30, regex: '^[1-9]{2}-[A-Z1-9]{2}-[A-Z1
      -9]{2}$'},
  name: {type: 'string', length: 30},
  count: {type: 'integer', min: 0},
  price: {type: 'double', min: 0},
  value: {type: 'double', min: 0},
  createdAt: {type: 'timestamp'},
  updatedAt: {type: 'timestamp'}
};
```

This is significantly different from the generated code which can be seen in the code snippet below.

```
this.fields = {

  SKU: {
    type: 'string',
    length: 30,
    regex: '^[1-9]{2}-[A-Z1-9]{2}-[A-Z1-9]{2}$',
```

```
  },

  name: {
    type: 'string',
    length: 30,
  },

  count: {
    type: 'integer',
    min: 0,
  },

  price: {
    type: 'double',
    min: 0,
  },

  value: {
    type: 'double',
    min: 0,
  },

  createdAt: {
    type: 'timestamp',

  },

  updatedAt: {
    type: 'timestamp',

  },

};
```

The generated version is different because the beautifier re-styles the code so it is according to the specification. The hand written version is obviously more aesthetically correct, but not necessarily according to the specification.

This requirement is highly dependent on the templates. The templates determine the main structure of the code. Making a proper template is not always easy because once one starts to use sub-templates, it is hard to determine the indentation of the code. This is solved by the beautifier which is a part of the generator. The problem is that the beautifier changes also other parts of the code so the result **is not** the same as if a programmer writes it. By applying the beautifier on the hand written code I got the same result as when I applied the beautifier on the executed template. One can say that the generator produces even better code than a programmer, because it abstracts from the personal aesthetics and only focuses on the well-known standards.

This requirement is therefore not satisfied but it is important to ask if it is a programmer written code that should be compared to the generated code or if it is a code implementing the style specification.

6.1.7 Folder Structure Quality

Name	Description	Priority
7. Folder Structure Quality	The folder structure of the REST API shall be the same as if a programmer structures it	medium

To evaluate this point, I will compare the generated REST API folder structure to a programmer written one. The following figure shows the REST API structured by a programmer.

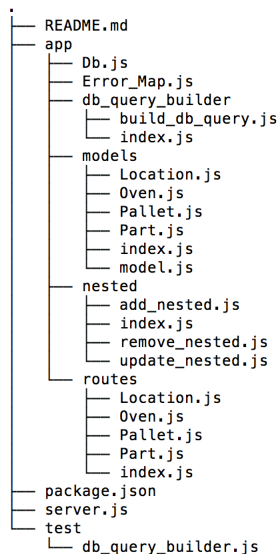

Figure 6.5: The REST API structured by a programmer

The logic is in the *app/* folder. This folder has a couple of sub-folders. *db_query_builder/* for translating the URI parameters to an SQL query. *models/* for storing the data models. *nested/* for operations with the nested objects and finally *routes/* for the controllers related to each endpoint grouped by the data model.

The following figure shows the file structure of the generated REST API.

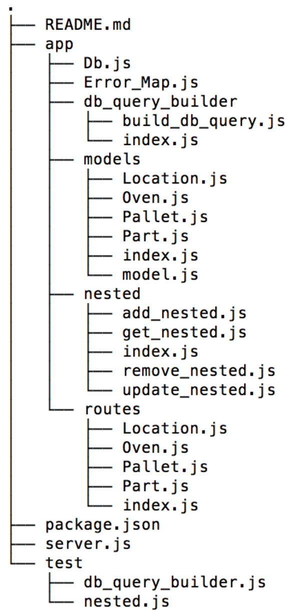

Figure 6.6: File structure of the generated REST API

It is obvious that the file structures are the same. The generated REST API has 2 additional files *test/nested.js* and *app/nested/get_nested.js*. This is due to the fact that the generated REST API includes unit tests for the Nested module and it also has a function for getting a nested object.

It is again important to say that the templates affect the result file structure much more than the generator itself. The generator's part in the process is the fact that the generator supports duplicated templates and is able to handle a different destination specified in the *destination* parameter in the template configuration file. The rest is in the templates.

Based on the comparison of the two file structures I am able to say that the generator *is able* to generate REST APIs with the same structure as programmer written REST APIs have. The requirement is therefore satisfied.

6.1.8 Re-Generating Management

Name	Description	Priority
8. Re-Generating Management	The generator shall support custom changes in the code which shall not disappear when the generation process is ran again	high

This requirement is satisfied because the generator has the ability to create a new branch in a specified GitHub repository and then the user can merge the two branches and pick what should be in the master branch. This functionality is described in section 5.3.10.

6.1.9 Template Re-Usability

Name	Description	Priority
9. Template Re-usability	The generator shall support template sharing	low

The generator supports Git so it is able to clone any repository and then use this repository as a source of templates. The only limitation is that the generator needs to be run on a machine which has Git installed. This feature allows to use any kind of template set from GitHub or Bitbucket. This is very powerful feature because developers from all around the world can collaborate on a template set and anyone can use the template set if the repository is publicly available. This requirement is therefore satisfied because **the generator supports** getting templates from publicly available repositories.

6.1.10 Output Testability

Name	Description	Priority
10. Output Testability	The generated REST API shall be testable	medium

There are two different ways how to test the output of the generator. First of all, there are unit tests. It is a best practice to have them included so once the generator generates the REST API, the programmer can run the unit tests. This will make the programmer sure that the functions in the code are working as intended.

The second way is to use Dredd. By running Dredd, one can immediately see which endpoints are working as intended and which need to be inspected by a programmer. The output of Dredd can be seen on the figure bellow.

```
info: Beginning Dredd testing...
pass: GET /ovens duration: 1185ms
pass: POST /ovens duration: 629ms
pass: GET /ovens/54d10585e4b00a11b540d01b duration: 623ms
pass: PUT /ovens/54d10585e4b00a11b540d01b duration: 631ms
pass: DELETE /ovens/54d10585e4b00a11b540d01b duration: 3ms
pass: GET /pallets duration: 656ms
pass: POST /pallets duration: 623ms
pass: GET /pallets/54d0bf38e9cc9063148461e5 duration: 648ms
pass: PUT /pallets/54d0bf38e9cc9063148461e5 duration: 628ms
pass: DELETE /pallets/54d0bf38e9cc9063148461e5 duration: 6ms
pass: POST /pallets/54d0bf38e9cc9063148461e5/parts duration: 660ms
pass: GET /pallets/54d0bf38e9cc9063148461e5/parts duration: 629ms
pass: GET /pallets/54d0bf38e9cc9063148461e5/parts/54d3e26b09d8c002113b0c6e duration: 626ms
pass: PUT /pallets/54d0bf38e9cc9063148461e5/parts/54d3e26b09d8c002113b0c6e duration: 682ms
pass: DELETE /pallets/54d0bf38e9cc9063148461e5/parts/54d3e26b09d8c002113b0c6e duration: 3ms
pass: GET /locations duration: 630ms
pass: POST /locations duration: 631ms
pass: GET /locations/54d10385e4b00a11b540cfff duration: 639ms
pass: PUT /locations/54d10385e4b00a11b540cfff duration: 628ms
pass: DELETE /locations/54d10385e4b00a11b540cfff duration: 15ms
pass: GET /parts duration: 627ms
pass: POST /parts duration: 634ms
pass: GET /parts/54d3d1d4a554e10f0be08900 duration: 642ms
pass: PUT /parts/54d3d1d4a554e10f0be08900 duration: 627ms
pass: DELETE /parts/54d3d1d4a554e10f0be08900 duration: 1ms
fail: GET /parts/54d3d1d4a554e10f0be08900/pallets duration: 6ms
```

Figure 6.7: Dredd output

The figure shows 26 endpoints which are working according to the best practice behavior and one (marked in the red color) which has failed.

There are two different mechanisms to ensure the user that the endpoints of the generated REST API are working according to the best practices. Both processes are automatic and the only action required from the user is to actually start the testing processes. This requirement is therefore satisfied.

6.2 Non-Functional Requirements

This section provides an evaluation of the developed system against the non-functional requirements.

6.2.1 Cross-Platform

Name	Description	Priority
N11. Cross-Platform	The generator shall work on OS X, Windows and Linux	medium

The generator is a REST API so it does not matter what kind of operational system the client is running. If one wants to run the generator, it requires a NodeJS environment. NodeJS as it can be seen on the official pages[4] has packages for OS X, Windows and Linux. In addition to that it also supports SunOS and one can also download the entire source code of the environment.

Since the only dependency is NodeJS which **is supported** on OS X, Windows and Linux, the requirement is satisfied.

[4]https://nodejs.org/download/ - NodeJS download page

6.2.2 Output Testability

Name	Description	Priority
N12. Output Testability	The tests of the generated REST API shall be automatic	medium

This constraint is fully satisfied because there are two different mechanisms- unit tests and Dredd for behavioral tests and they are both automatic.

6.2.3 Template Sharing

Name	Description	Priority
N13. Template Sharing	The template sharing shall be done without any manual download	medium

The shareability of the templates is done via GitHub. The generator offers endpoints */github-to-zip* and */github-to-github* which clone the template set from the specified repository and then use them as a source to generate the REST API. There is therefore **no manual downloading** and copy-pasting needed in order to use somebody's templates. This requirement is satisfied.

6.2.4 Speed

Name	Description	Priority
N14. Speed	The generator shall generate the REST API in a reasonable time	low

The speed of the generator is dependent on many factors:

- complexity of the templates
- complexity of the model
- network speed

To abstract from the network I will measure the time from the point when the generator receives a request until the time when the generator responds. I used to model from the use case with 4 entities. I did 10 measurements and the average time was **38.5ms**. I also performed a measurement from the point when a request is fired to the point when the response is sent to the client. The average time of 10 requests is **1260.625ms**.

When comparing those two numbers it is obvious that the network plays a huge role in the generation process. I believe that both numbers are reasonable as one can trigger the code generation on every change in the templates and in 1.2s see the impact of the changes. I believe that the validity of the generated code is much more important than the speed of the generator. I therefore believe that this requirement is satisfied.

6.3 Limitations

There are several limitations in the implementation. The entire functionality has been evaluated on one single case. In practice, every case might have different needs, which might not be supported at the moment. The demo videos attached to this report in the appendix B show a different case which does not require anything extra in comparison to the one used for the evaluation.

The current implementation supports following data types:

- primitive types such as String, Integer, Double, GeoJSON and Timestamp
- another entity as a data type
- array of primitive types or another entity type

The current prediction is limited to hierarchical REST APIs. This means that the URIs have a hierarchical structure and reflect the data model (for example *universities/34/faculties/15/institutes/19*). URIs such as *education/universities/34* are not understood and the generator is not able to predict the behavior of such endpoint. It makes sense to create hierarchical REST APIs but it is not a rule. These is another limitation connected to the behavior prediction and that is the first step when matching URIs to the entities in the data structure. This step is based on the naming conventions which can be broken for example with a self-relation. Self-relation is needed for example when entity *Person* is parent of another *Person*. The editor and the NodeJS template set currently does not support that. The good this is that the generator places an empty function instead of the generated code so there is nothing lost in the process of generating the REST API.

Chapter 7

Discussion

This chapter discusses the research questions and whether they were answered or not.

Is it possible to generate production ready, structured and testable REST APIs based on a data model including endpoints?

This question has several parts so I will take each part and answer them separately. A REST API is *production ready* when it can be deployed to a server and can be immediately used for the application development. This is possible but there will always be some minor steps before the deployment. Every REST API needs to be connected to a database therefore database connection information needs to be filled. Some minor configuration might be also required because the templates might require that. Every template set should come with a list of after generation steps, that need to be fulfilled before deployment. The example template set for NodeJS requires only to enter the database settings and then run *npm install* to get the dependencies. Once that is done, the REST API can be used in the template development process which proves that the output of the generator is ready to be immediately used.

structured is a REST API which has a logical file structure and an easy to read code. The user should feel that the code is written by a human and it should feel natural to make changes in the code. This is in general difficult to achieve because every programmer has a different style. The template set should contain an explanation of the file structure and which module is responsible for what. The generator has enough mechanisms to support a wide range of file structures. When it goes about the code style, it is very hard to achieve the human written style. Every programmer has different habits and the desired state would be that the generator would generate the style the programmer is using. This is currently impossible, but the beautifier has a couple of settings every programmer can tweak to come closer to the desired style. In general, the emphasis is on generating code which has the style good enough and follows the general rules of JSLint.

Testability is an important part of the generation process. There are in general two ways how to test the result- unit testing and behavioral testing. Unit tests are just templates being executed and run when needed. The behavioral testing is based on the API Blueprint file. There are a couple of challenges with the API Blueprint based behavioral testing. The

main problem is that the file needs real IDs from the database in order to perform the testing. There are two ways how to pass this information. There is a scope helper function which contains the URIs and real values for each parameter in the URI. The other option is to tweak the API Blueprint file after it has been generated. Dredd has a system of hooks which might be helpful in the future, but this functionality is not currently mature enough. There are therefore still some challenges when it goes about behavioral testing of generated REST APIs.

This report shows that it is possible to generate production ready, nicely structured and testable REST APIs just with a data model and endpoints. There are still some challenges to overcome, but the output of the generator is a way better start for client-side developers than any other available solution.

How does one share the knowledge of best practices about REST APIs, so others without the knowledge can use it?

One shares the knowledge via templates. By applying MDD on the domain of REST APIs the knowledge simply gets divided between two sides- a client-side developer who makes a simple data model and a server-side developer who makes the templates. To make this process even easier for the client-side developers the templates (= server-side programming knowledge) can be shared via GitHub. This opens a huge opportunity for the entire community to create new templates based on always changing best practices and maintain the existing one. Everyone can get involved. Using GitHub is definitely a natural solution for all developers. It is not only GitHub but the Git command itself.

Chapter 8

Conclusion and Future Work

I believe that code generation together with the merging mechanisms is a perfect way to attack domains with a lot of repetitive code. The possibility to include the community as a source of knowledge makes the platform even more flexible and accessible for people without the necessary server-side knowledge. This can dramatically increase the development speed in companies using REST as an architectural style. According to my discussion with Tor-Inge Eriksen from Searis AS the development of REST API takes about 70% of the budget. This number has been estimated based on the time tracked in a time tracking software and compared to the other expenses in the project. The code generation reduces this number dramatically so the budget can be used on the client-side application which is the part of the entire system which matters way more for the client and the end users.

I believe that the smaller the generated part of the entire system is, the more accurate are the results. Another very strong argument is that the architecture of the entire REST API is changing based on its size. This is of course true but the latest trends are focused on micro services which breaks huge REST APIs into small flexible pieces. What I believe is the problem with the generated code is the perception of programmers. Not every programmer is happy with the idea of working with generated code. This is understandable because the programmer suddenly becomes just a maintainer.

The domain of REST APIs has a highly repetitive code and the proposed solution in this report does solve a lot when the REST API is not the main product. It is logical that when the end product is a REST API this tool is not probably the best choice but it can set a good starting point. There are several fields which can be further investigated in order to improve the generator and the generated code:

- **Behavioral testing of the REST API -** Dredd and API Blueprint are very strong combination, but there is still a manual step required in order to fully test the resulted REST API. Dredd does not implement the necessary functionality to correctly order the requests in the blueprint so it is not possible to have the database in the same state before and after the testing. This is only possible by seeding the database and then reseed it to get it into a known state(Cordell, 2015). It would be interesting to see a research on how to avoid the manual step.

- **Syntax highlighting for the templates -** The generator has enough mechanisms to make writing templates easy and in some structure. The only difficulty is when one writes the templates. A research on how to distinguish the template code from the actual code is more then welcome.

- **GitHub authentication and the related security issues -** Currently it is only possible to use GitHub when the generator runs on a local server. There is space for a research on how to enable this functionality with the generator on the server. This means that one would have to pass his/her credentials over the network which has some security risks which needs to be taken in account.

- **Auto-deployment -** There is a huge potential for auto-deployment of the generated code. This is possible with platforms like Heroku and it would be interesting to see different versions of the generated REST API being automatically deployed. This of course has some difficulties which need to be researched first.

- **Behavior prediction -** Probably the weakest part of the template set is the ability to predict behavior. It works but it might break in some edge cases. At the moment it only means that the programmer needs to fill in the implementation even if the behavior is predictable. It is important to realize that the level of understanding can be increased by enriching the model with some additional information that would help the generator in determining the correct behavior. This comes with a price of more possible settings in the editor and therefore a more steep learning curve the user has to overcome. It would be helpful to conduct research on how to extend the model/editor/generator so the prediction works every time and the user is not overwhelmed by options and settings.

- **Formatting API Blueprint -** API Blueprint files has a very specific formatting and I was not able to follow the formatting when generating the file. There is currently no beautifier available so every time one runs Dredd, there are a couple of warnings which point out that the formatting is not correct. It does not affect the functionality of the file but it does not always look aesthetically correct.

Bibliography

Ashkenas, J., 2015. Last checked: 11.3.2015.
URL http://underscorejs.org/

Burke, B., 2009. RESTful Java with JAX-RS. O'Reilly Media, Inc., iSBN: 9781449383053.

Cordell, E., 2015. –sorted does not really sorts how it should. Last checked: 10.5.2015.
URL https://github.com/apiaryio/dredd/issues/170

Czarnecki, K., Helsen, S., 2003. Classification of Model Transformation Approaches. University of Waterloo, Canada, 17.

Ed-douibi, H., collective, 2015. EMF-REST: Generation of RESTful APIs from Models. Tech. rep., Inria, Mines Nantes, LINA.

EMF-REST, Last checked: 26.5.2015 2015. Documentation.
URL http://emf-rest.com/documentation.html

Fielding, R. T., 2000. Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine.

Fouquet, F., collective, 2012. An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Environments. Tech. rep., University of Rennes, SnT University of Luxembourg, SINTEF.

Git, 2015. Git –everything-is-local. Last checked: 24.2.2015.
URL http://git-scm.com/

Harrell, J., 2013. Node.js at PayPal. Last checked: 16.3.2015.
URL https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/

Hevner, A. R., March, S. T., Park, J., Ram, S., March 2004. Design Science in Information Systems Research. MIS 28 (1), 75–105.

Hiroshi Wada, Junichi Suzuki, A. M., Oba, K., 2010. Matilda: A Generic and Customizable Framework for Direct Model Execution in Model-Driven Software Development. Department of Computer Science University of Massachusetts, Boston, 28.

Holowaychuk, T. J., 2011. Last checked: 11.2.2015.
URL http://mochajs.org/

Hunt, P., 2014. React: RESTful UI Rendering. Last checked: 3.5.2015.
URL https://www.youtube.com/watch?v=IVvHPPcl2TM

IEEE, 2010. Standards Glossary. Last checked: 3.5.2015.
URL http://www.ieee.org/education_careers/education/standards/standards_glossary.html

Joyent, I., 2015. Last checked: 11.2.2015.
URL http://nodejs.org/

Marco Brambilla, Jordi Cabot, M. W., 2012. Model-Driven Software Engineering in Practice. Morgan and Claypoool publishers.

Massé, M., 2012. REST API Design Rulebook. No. 9781449310509 in 1. O'Reilly.

Mozilla, May 2015. Preflighted requests. Last checked: 26.5.2015.
URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS#Preflighted_requests

MuleSoft, 2015. About RAML. Last checked: 31.1.2015.
URL http://raml.org/about.html

Nesetril, J., 2012. Avoid API Waterfalls, Build Faster and Better. Last checked: 31.1.2015.
URL https://www.youtube.com/watch?v=tP7fpu4r1wo

Prochazka, T., 2014. Model-Driven Development of REST APIs. Tech. rep., NTNU.

Reverb, 2015. Swagger. Last checked: 31.1.2015.
URL http://swagger.io/

Stahl, T., Völter, M., 2005. Model-Driven Software Development. No. 0470025700 in 1. Wiley.

Strauch, C., 2012. NoSQL Databases. Tech. rep., Stuttgart Media University.

Taal, M., 2015. EMFT Texo. Last checked: 31.1.2015.
URL https://projects.eclipse.org/projects/modeling.emft.texo

Appendix A- Study Case

Inventory system

This material has been given to Tomas Prochazka as a practical case study for his specialization project at NTNU. The following case is part of an inventory system which we developed at Searis AS and which is in daily use by our customers. Our system consists of a backend system which exposes a REST API for interacting with the inventory (add/remove part, move pallet, register new orders etc.) End-users access the system either through an iPad app or through a single-page web application, both of which are developed inhouse.

Within our inventory system we have four main datastructures, which are presented to the end-user in different combination in order to provide the required functionality. The primary datastructures are:

- Oven
- Pallet
- Part
- Location

Firstly we have *Locations*, which represents physical position within a storage area, which in practice either means that they represent an actual rack, or a designated area on the floor. These position are named in such a way that the user can easily locate them within the factory, in addition they are clearly marked with signs. Since all *Pallets* are located at a *Location* users are able to locate individual *Pallets* easily.

Each pallet can contain a quantity of *Parts*, often these will all be of the same type, but we support a single *Pallet* containing different *Parts.* An *Oven* object represents a physical *Oven*, which is made up of a number of different *Parts.* In order to describe an *Oven* we require two numbers, one which specifies the total amount needed of a given *Part*, and another one which tells us how many *Part* of a given type which in actually contained in that *Oven*. This allows us to represent both complete and non-complete *Ovens.* Like *Pallets*, *Ovens* are located at a *Location*.

In order to use as few tables as possible for our database, the system combines the base datastructures in order to represent different use cases. In practice, this means that for a given *Part* (e.g. “Octo 50, standardstein”), we will only have a single entry in the database. When a *Pallet* contains this part, we use a mediary table to combine them and to add the notion of a *count* of *Parts* for that specific *Pallet.* This is also true for *Ovens*, but here we add two numbers, one for *required* and one for *packed*.

Our REST API closely follows a set of guidelines (https://stormpath.com/blog/designing-rest-json-apis/) proposed by *User Management API* company Stormpath. In addition we have added a concept called *Hydration*, inspired in part

by a series (http://openmymind.net/Practical-SOA-Hydration-Part-1/) of blog posts (http://openmymind.net/Practical-SOA-Hydration-Part-2/). These concepts allow us to make a highly structured, and hopefully useable API.

The following tables show the details of each data structure

Oven


```
{
    "id": 1,
    "sku": "AA-AA",
    "name": "OCTO 50",
    "status": "delivered",
    "createdAt": "2014-11-13T15:15:38Z",
    "updatedAt": "2014-11-19T13:52:50Z",
    "meta": {
        "href": /inventory/ovens/1,
        "mediaTypes": "application/json;application/xml"
    },
    "parts": {
        "meta": {
            "href": /inventory/ovens/1/parts,
            "mediaTypes": "application/json"
        }
    },
    "location": {
           "meta": {
               "href": /inventory/locations/1,
               "mediaTypes": "application/json"
           }
      }
}
```

Pallet


```
{
    "id": 1,
    "sku": "AA-AA",
    "createdAt": "2014-11-13T15:15:38Z",
    "updatedAt": "2014-11-19T13:52:50Z",
    "meta": {
        "href": /inventory/pallets/1,
        "mediaTypes": "application/json;application/xml"
    },
    "location": {
        "meta": {
            "href": /inventory/locations/1,
            "mediaTypes": "application/json"
        }
    },
    "parts": {
        "meta": {
            "href": /inventory/pallets/1/parts,
            "mediaTypes": "application/json"
        }
    }
}
```

Part


```
{
    "id": 1,
    "sku": "01-01-01",
    "name": "Teststein 1",
    "count": 250,
    "price": 150,
    "value": 50,
    "createdAt": "2014-11-13T15:15:34Z",
    "updatedAt": "2014-11-15T15:40:47Z",
    "meta": {
        "href": /inventory/parts/1,
        "mediaTypes": "application/json;application/xml"
    }
}
```

Location


```
{
    "id": 1,
    "sku": "A01-01",
    "name": "Uteområde A",
    "geoLocation": {
        "type": "POINT",
            "coordinates": [
                63.1,
                10.2
        ]
    },
    "createdAt": "2014-11-13T15:15:37Z",
    "updatedAt": "2014-11-13T15:15:37Z",
    "meta": {
        "href": /inventory/locations/1,
        "mediaTypes": "application/json;application/xml"
    }
}
```

Our REST API has following endpoints:

Method	URL
GET	parts/
GET	parts/:id
POST	parts/
PUT	parts/:id
DELETE	parts/:id
GET	parts/:id/pallets
POST	parts/:id/pallets
GET	pallets/
GET	pallets/:id
POST	pallets/
PUT	pallets/
DELETE	pallets/:id
GET	pallets/:id/parts
POST	pallets/:id/parts
GET	pallets/:id/parts/:part_id
PUT	pallets/:id/parts/:part_id
DELETE	pallets/:id/parts/:part_id
GET	locations/
GET	locations/:id
POST	locations/
PUT	locations/:id
DELETE	locations/:id
GET	ovens/

GET	ovens/:id
POST	ovens/
PUT	ovens/:id
DELETE	ovens/:id

Appendix B- Videos

- MDD of REST APIs - #1 Modeling - https://www.youtube.com/watch?v=R3xzHKp_RWo
- MDD of REST APIs - #2 Testing - https://www.youtube.com/watch?v=xSsYVvBgsbU
- MDD of REST APIs - #3 Git integration - https://www.youtube.com/watch?v=qc226BCrhnQ

Appendix C- External Assessment

31.05.2015

Often times when writing software, we find ourselves spending a lot of time doing things which seems only remotely connected to what we are actually trying to achieve. In general, only a fraction of the time spent in a software engineering project will be spent implementing the logic and operation that we want, while a lot of time is dedicated to writing the infrastructure needed to support this functionality. This is especially true of backend projects, where functionality such as rate limiting, authentication, authorization, input validation etc. must be implemented before having a production ready system.

At Searis we develop software for internal use, as well as doing work on consulting projects. The type of projects we're involved in often revolves around relatively complex backend implementations, combined with frontend software for the browser, iOS and Android. In order for the cooperation between our back- and frontend developers to work as smoothly as possible, we see that up-to-date documentation as well as regular testing of our APIs is crucial.

Testing endpoints of the API after making changes is especially important as it allows us to ensure that exposed functionality is not accidentally altered. This can be hard to achieve with regular unit testing, as these are often using mocked objects (http://www.ibm.com/developerworks/library/j-mocktest/j-mocktest-pdf.pdf) or in other ways differ from the environment in which our production code run.

By integrating the steps required for data modelling, documentation, testing and deployment we expect to see a reduction in the time spent coordinating work between back- and frontend developers. In addition we hope to effectively eliminate the number of API endpoints which are either undocumented or which has out of date documentation.

The ability to auto generate code for our APIs allows us to quickly prototype APIs which our client software can run against. In addition it has the potential to lower the time and effort required when refactoring, while ensuring compatibility with client software.

We have already tested the software Tomas has developed as part of his thesis and we will be implementing the first full scale prototype using the generator during the fall.

Tor-Inge J. Eriksen
Co-founder / Lead Developer at Searis AS
+47 975 82 216
tor@searis.no

Appendix D- Unit Tests

```
Metamodel
  ✓ should be defined
  ✓ should evaluate the example model as valid
  ✓ should check for blind associations
  ✓ should check that integer/double property has min smaller than max
  ✓ should check that the array value is valid
  ✓ should check that the http method is allowed

Scope
  ✓ should be defined
  ✓ should register new scope variable
  ✓ should register an array as scope variables
  ✓ should not allow to register property named "model" to scope
  ✓ should not allow to register already existing property
  ✓ should return an object
  ✓ should contain the model

Scope Helpers
  ✓ should be defined
  Model by Name (get_model_by_name)
    ✓ should be defined
    ✓ should return the right model
    ✓ should return null if the model is not found
    ✓ should return the model even with lower first letter
    ✓ should return the model even with plural
    ✓ should return the model even with lower first letter and plural
  Is Singular (is_singular_resource)
    ✓ should be defined
    ✓ should return true
    ✓ should return false
  Get Controller Name (get_controller_name)
    ✓ should be defined
    ✓ should return the correct name
    ✓ should accept URLs with slash as a first character
  Get latest param
    ✓ should return the latest parameter name
  Find URL param in properties
    ✓ should find the string in the properties
  Get natural language for endpoint
    ✓ should get the correct natural language for endpoint

Template Config Validator
  ✓ should evaluate the config file as valid

Template_Executor
  Normal templates
    ✓ should execute normal templates
    ✓ should execute atomic templates in normal templates
  Duplicated templates
    ✓ should execute duplicated templates
    ✓ should contain template for each model
    ✓ should execute atomic template in the duplicated templates

Template Loader
  ✓ should be defined
  ✓ should load the config file
  ✓ should determine the right template name
  ✓ should load the atomic templates
  ✓ should load the duplicated templates
  ✓ should load the normal templates

Template Saver
  ✓ should be defined
  ✓ should save the normal templates with a correct name
  ✓ should save the duplicated templates with the correct name
  ✓ should work with normal templates in subfolder
  ✓ should save normal templates according to the destination param
  ✓ should be able to combine destination and subfolder with normal templates
  ✓ should save duplicated templates according to the destination param
  ✓ should work with duplicated templates in subfolder
  ✓ should be able to combine destination and subfolder with duplicated templates
  ✓ should determine the right file suffix
  ✓ should determine the right file name

config
  ✓ should return an object with configuration


53 passing (49ms)
```