Model-driven development of REST APIs

by Tomas Prochazka

TDT4501

Submission date: December 17, 2014
Performed at: Department of Computer and Information Science, NTNU
Supervisors: John Krogstie, IDI, NTNU

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

The report is a first part of the research on applying Model-Driven Development on developing REST APIs. The motivation to do this is because front-end programming is different from the server-side programming and since single-page applications using REST API are becoming more and more popular, a way how to develop high quality REST APIs in the least time is needed. The method should be based on something similar to both client-side and server-side developers. Model-Driven Development is a perfect candidate to solve this. The article describes the notation for REST endpoints connected to data structure they work with. An editor has been developed to support this notation. This editor exports the model to JSON so the model can be used in code generation. This paper covers the implementation of the editor and shows the functionality including JSON output. A study case has been used to evaluate the editor’s functionality and ability to keep the important information related to the case. As has been already mentioned, this is the first part of the research and it prepares the ground for the code generation of REST APIs.
Preface

This paper has been created as a report to the developed software I was working on in the autumn semester 2014. The whole project including this report was used as a delivery to TDT4501 - Computer Science, Specialization Project at Norwegian University of Science and Technology in Trondheim (NTNU). This paper has been delivered in electronic form.

There are several digital attachments together with this report, including a video and source code. All the attachments are linked in this report so it is possible to access them for the readers of this report.

All my technical decisions were discussed with my colleagues from Searis AS, a small startup developing custom software for aquaculture industry. I still remember the reaction of my colleague Tor-Inge J. Eriksen when I sent him my idea about combining REST APIs together with Model-Driven Development: “I read your mail and I think... I think it is awesome!”. Thank you guys, get the chance to work with you is the second best thing that has happened to me since I moved my life to Trondheim.

I would also like to thank to my supervisor John Krogstie not only for allowing me to get into this topic and supervising my activities related to the project but also for the inspiration to actually come up with such an idea. Thank you!

Trondheim 24.10.14
Tomas Prochazka
# Contents

Abstract  
Preface  
Contents  
List of Figures  
Abbreviations  

1 Introduction  

## 2 Problem Definition  
2.1 Client-server Architecture and REST  
2.2 Communication  
2.3 Problems with REST API  
  2.3.1 Complexity  
  2.3.2 Maintainability  
  2.3.3 Duplicated Code  
  2.3.4 Documentation  

## 3 Background  
3.1 Model-Driven Development  
  3.1.1 Code Generation  
  3.1.2 Model as Documentation  
3.2 Model Notation
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Summary</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>Methodology</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>Model-Driven Development and REST API</td>
<td>11</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Apiary</td>
<td>11</td>
</tr>
<tr>
<td>4.1.2</td>
<td>EMF-REST</td>
<td>12</td>
</tr>
<tr>
<td>4.1.3</td>
<td>New Tool</td>
<td>12</td>
</tr>
<tr>
<td>4.2</td>
<td>Research Question</td>
<td>13</td>
</tr>
<tr>
<td>4.3</td>
<td>Development Stack Research</td>
<td>13</td>
</tr>
<tr>
<td>4.4</td>
<td>Editor</td>
<td>13</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>Front-End Tool Research</td>
<td>15</td>
</tr>
<tr>
<td>5.1</td>
<td>Data structures and behavior</td>
<td>15</td>
</tr>
<tr>
<td>5.2</td>
<td>DOM Manipulation</td>
<td>16</td>
</tr>
<tr>
<td>5.3</td>
<td>Testing</td>
<td>17</td>
</tr>
<tr>
<td>5.4</td>
<td>Quality of Code</td>
<td>17</td>
</tr>
<tr>
<td>5.5</td>
<td>CSS Pre-processing</td>
<td>18</td>
</tr>
<tr>
<td>5.6</td>
<td>Automation</td>
<td>18</td>
</tr>
<tr>
<td>5.7</td>
<td>NodeJS</td>
<td>19</td>
</tr>
<tr>
<td>5.8</td>
<td>Summary</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>Implementation</td>
<td>21</td>
</tr>
<tr>
<td>6.1</td>
<td>Root Directory</td>
<td>21</td>
</tr>
<tr>
<td>6.2</td>
<td>Namespaces</td>
<td>22</td>
</tr>
<tr>
<td>6.3</td>
<td>Data and UI layer Connection</td>
<td>23</td>
</tr>
<tr>
<td>6.4</td>
<td>Calculating Associations Between Entities</td>
<td>23</td>
</tr>
<tr>
<td>6.5</td>
<td>Initial Rendering</td>
<td>27</td>
</tr>
<tr>
<td>6.6</td>
<td>The Wizard</td>
<td>28</td>
</tr>
<tr>
<td>6.7</td>
<td>Working with an Existing Model</td>
<td>30</td>
</tr>
<tr>
<td>6.8</td>
<td>Export to JSON</td>
<td>31</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Client-server communication with REST API ............................... 4
3.1 Notation for entities ......................................................... 9
5.1 The development stack running in 3 terminal windows ................. 19
6.1 The structure of the application divided with namespaces ............... 21
6.2 The structure of the application divided with namespaces ............... 22
6.3 Connection between data and UI layer ................................... 23
6.4 Possible positions of two different entities as a base for drawing associations . 24
6.5 Triangle for calculations between two entities ........................... 25
6.6 Triangle for calculations between two entities ........................... 27
6.7 Diagram for an optimal rendering ...................................... 28
6.8 Interface for adding a new entity ....................................... 28
6.9 Example entity ................................................................. 30
6.10 Model with forms for adding properties and endpoints ................. 31
6.11 Button for triggering JSON generation of the model .................. 32
7.1 Case modeled in Gliffy ....................................................... 34
7.2 Divided endpoints in Gliffy ............................................... 35
7.3 The case modeled in the editor showing properties ....................... 36
7.4 The case modeled in the editor showing endpoints ....................... 36
C.1 Test coverage of the editor ............................................... a-21
D.1 The whole environment in the browser ................................ a-22
Abbreviations

API Application Programming Interface
REST Representational State Transfer
JSON JavaScript Object Notation
DSL Domain Specific Language
NPM Node Package Manager
DOM Domain Object Model
SVG Scalable Vector Graphics
MDD Model-Driven Development
Chapter 1

Introduction

Single-page applications are becoming more and more popular. It is mainly due to the fact that the latest trends are focusing a lot on the user experience and single-page applications have much more potential in this than the traditional web applications. As Daniel F. Pupius has described in his article "even modest improvements in latency can have measurable impact on usage. As such, sites should have aggressive goals for both actual and user perceived latency"[Pupius]. Single-page application is an application that "uses only one HTML web page as a shell for all the application's web pages and whose end-user interactions are implemented by using JavaScript, HTML, and CSS"[Flink and Flatow, 2014]. Any other communication with the server is done via AJAX. The server then retrieves only data (also known as resources) in a standardized format and the knowledge of the format gives the application an ability to present the data to the user. This report is not focusing on single-page applications but on the server side of this communication, on REST APIs.

When developing single-page applications the problem is that these applications need to be developed against something- in most cases REST API. The problem is that server-side programming and client-side programming are two completely different disciplines. A perfect example can be the fact that books like 'JavaScript for PHP developers' are being released. Another problem is that developing REST APIs that are able to scale and have a high level of security is a time consuming task. Time, which should be used on the actual application and not on something that drives it.

I am trying to find the common language for both worlds in this report. I strongly believe that the ability to visualize software in form of models is a skill every programmer has regardless of the platform, language or environment. Therefore it is logical to explore the position of model-driven development in terms of REST APIs.

This report establishes a notation for REST APIs and then it describes a modeling tool which is using this notation. The evaluation of the modeling tool is done with a case given for a real project. The success of the tool is dependent on to what extent it is able to capture the important information front the case and export them into a machine readable format, which can be processed so the desired REST API can be created.

The report is divided into 8 chapters. First chapter contains introduction into the problem area.
The second chapter describes what is the problem this report is trying to attack. Third chapter gives a theoretical background of model-driven development, proposes MDD as a solution of the problem and introduces the notation which is going to be used. Fourth chapter is about the research methodology used to evaluate the results. Fifth chapter introduces the development stack which was used to develop the solution. Sixth chapter shows the implementation. Seventh chapter is about application of the developed solution to the given study case and finally eighth chapter makes a conclusion and presents the limitations and observations.
Chapter 2

Problem Definition

This chapter describes the current situation with the single-page application development in relation with REST APIs and covers the necessary theory. Problems are being established together with an argumentation why this is an actual problem.

2.1 Client-server Architecture and REST

Client-server architecture is a necessary part of nowadays world wide web. It is an architecture where many clients requests a server and then receive an answer which is interpreted to the user. The server waits for a request from the client. The request is proceeded and a response is sent to the client. Client computers provide an user interface which is able to send a request to the server and show the response in an user friendly form. [Britannica] Clients can differ from web browsers to native applications running on tablets or mobile devices. Every server has an interface which is used to request the server. In the case of web servers, this interface is called a web API.

If a web API follows REST (Representational State Transfer) architecture style, it is referred to as an REST API. REST is an architectural style composed of six constrains [Fielding, 2000]. Fielding, an author of REST architectural style, states in his dissertation thesis that the World Wide Web scalability is driven by six constrains:

- Client-server
- Uniform interface
- Layered system
- Cache
- Stateless
- Code-on-demand
2.2 Communication

As mentioned in the previous section, clients of REST APIs covers a whole spectrum of applications such as native applications for mobile platforms and web browsers. This highlights how flexible REST architectural style is. Communication between client and REST API is shown in the picture below.

![Client-server communication with REST API](image)

The client sends a request to the server. This request is sent to an URI which specifies the resource or resources the client wants to access. The server registers this request and sends the response to the client. The response data format may vary but I will focus only on the one serving data in the JSON data format. The client then consumes the data and shows them to the user in the user readable form.

It is obvious that the data transfer between client and server becomes much smaller compared to the traditional web applications. Such applications have to load the entire page every time a request is made. This includes HTML of the page, imported CSS, JavaScript files and images.

2.3 Problems with REST API

When building clients with an intention to consume data, REST API becomes a must. Instead of focusing on the development of the application, front-end and native application engineers have to face problems of server side programming to have a background for the desired application. Development of the server-side takes a significant amount of time and has nothing to do with the application development. Client side developers have to face server side programming which is significantly different from what they are used to. When a quick REST API development has to be considered, I have listed the most common problems developers have to overcome.
2.3. PROBLEMS WITH REST API

2.3.1 Complexity

Developing a back-end applications is a complex process. The following features need to be implemented:

- routing
- controllers with business logic
- API token handling
- data access
- data validation
- error handling

Developing from scratch can lead to a lot of vulnerabilities and weak spots in the application and it is extremely time consuming which in this case is an argument because the focus should be on the client side application. One can use an external library or a module but choosing the right one which satisfies all needs, is updated regularly and is written in a way so it can interact with other modules is a completely different problem.

2.3.2 Maintainability

According to the Separation of Concerns programming paradigm\(^1\) every functionality in any program has to be performed in a one single place. In terms of REST API, following concerns are considered:

- routing
- data validating
- request and response handling
- database manipulating

This means that any maintenance of a single endpoint has to be done on at least 4 different places when following the Separation of Concerns paradigm. The problem with this approach is that once a new endpoint is introduced, developers often have to create a route, create a controller (most of the time an old once is just being copied and the name is changed) and create a new data service. This can be exhausting and again time consuming.

\(^1\)More about separation of concerns can be found here: http://ftp.ccs.neu.edu/pub/people/crista/publications/techrep95/separation.pdf
2.3.3 Duplicated Code

This is a problem especially in controllers which handles requests and responses. It is the controllers responsibility to collect passed parameters, validate them, validate the API token included in the request and then call the right database manipulation function. Also filtering which removes parameters in the result data the authenticated user is allowed to receive and which require higher role in the user hierarchy. The problem with this approach is that the controllers contain a certain amount of code which is the same for every single controller with small changes.

2.3.4 Documentation

Documentation is an extremely important part of the REST API. It is essential so the developers know what kind of endpoints are available, which method to use and what parameters to send in order to get the desired resource. Keeping documentation up-to-date can be again time consuming.
Chapter 3

Background

This chapter proposes a solution to the problem described in the previous chapter. An argumentation of why model-driven development (MDD) is a fitting solution is given.

3.1 Model-Driven Development

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 [Brambilla et al., 2012]. This approach has a couple of characteristics which can help solve the problems mentioned in the previous chapter.

3.1.1 Code Generation

Code generation (also referred to as a model compiling) aims at generating running code from a higher level model in order to create a working application [Brambilla et al., 2012]. Code generation can be partial or complete. I believe that in the case of REST API partial code generation is the better option because an open source, well tested and by the community accepted modules can be used as a skeleton of the REST API. The model compiler can inject information based on the model into the skeleton. Also data structures and their validation rules can be generated. This method of software development would solve the problem with complexity of writing REST APIs mentioned in the previous chapter.

The same method can be applied to solve the problem with maintainability. The amount of code generated is based on the detail of the model. If the model is detailed enough all necessary parts of the code can be generated from the model, which means that if a change needs to be done or a new endpoint needs to be introduced, only one alternation is required from the developer - alteration of the model followed by regenerating the code and re-deployment. Since everything comes from the model, the necessary changes will be done in the model, on one place in an editor. This method solves the maintainability problem where a single request means multiple changes on multiple places that needs to be done to achieve the result.
The same counts for code duplication. Instead of copy-pasting parts from existing code and adjusting the code chunk so it fits to the context, this code can be again generated from a template. This solves the problem with duplicated code.

3.1.2 Model as Documentation

Models are a great source for documentation. In fact the traditional approach of documenting code contains a lot of different models based on the UML\(^1\) specification. Since everything is focused on the model in model-driven development, it forces the developer to model from the beginning of the development process. The modeling process creates a great base for the code generation of the application but also a visual documentation which is easy to understand. This method partly solves the problem with documentation. Only partly because an additional explanation should be provided in case of large models where the idea behind the model is not obvious.

It is clear that in order to model REST API endpoints a new notation has to be established. This notation shall be able to capture resource structures together with endpoints, which operate on the data structures.

3.2 Model Notation

The minimum required information an entity should contain is the resource structure and endpoints accessing this resource. The resource structure is necessary because of validation of the incoming data sent to the REST API. User shall be able to express:

- entity name
- attributes of the entity
- types of attributes (string, integer, double etc.)
- relations between entities (some entities can contain attributes with the type of another entity)
- relation name

All these conditions are satisfied with the relational data model. This concept was first described by E.F.Codd in 1970\(^2\). This gives enough tools to model resource structures and their relations but endpoints to these resources needs to be expressed too. Every endpoint consists of a location to identify the desired resource or resources and an HTTP method to define the

---

\(^1\)http://www.uml.org/
\(^2\)http://www.seas.upenn.edu/ zives/03f/cis550/codd.pdf - A Relational Model of Data or Large Shared Data Banks
action to be performed. If the relational data model notation is extended with the additional information about the endpoints, the entity may look like on the picture below.

![Diagram of entity notation]

Figure 3.1: Notation for entities

With this notation the code base for several processes can be generated:

- routing, because the endpoints are known
- data validation, because the data structures are known
- generating database commands, because of the HTTP method used and the location

### 3.3 Summary

MDD is a great candidate to solve the problems mentioned in the previous chapter. Since there is no graphical notation for REST APIs, I have created one based on the relational data model notation extended by the endpoints. To research the position of Model-Driven development in developing REST APIs an editor supporting this notation needs to be developed. This editor shall allow to model the entities and their relations and is able to export the whole model into a machine readable format, so the output can be used for generating the actual REST API.
Chapter 4

Methodology

This chapter describes the research methodology used. First, currently available tools are reviewed and the need for further research is described. The research itself is divided into two parts—JavaScript libraries and the editor itself. Each part has criteria for evaluation so the study can be considered as success or not.

4.1 Model-Driven Development and REST API

Model-Driven Development is not new in connection with developing REST APIs and this section briefly summarizes the currently available tools and why there is a need for further research in this area.

4.1.1 Apiary

First possibility for prototyping REST APIs is Apiary\(^1\). This tool generates a mocked REST API from a blueprint, which the user writes in a web based text editor. This blueprint is considered to be a text model, where the user specifies what kind of endpoints are available and what responses should be generated when the endpoint is called. This tool not only provides the API so it is possible to use it when developing client-side applications but also generates documentation which can be later used in the development process. Unfortunately Apiary does not offer code generation based on the blueprint so it is a neat way to design and document REST APIs but not to implement and maintain it. Apiary is a nice tool for collaboration of several developers on one project but its purpose is more focused on the documentation and a fast development of clients accessing the API.

\(^1\)Apiary.io - Homepage of the Apiary project
4.1.2 EMF-REST

Another possibility is to use EMF-REST\textsuperscript{2}. EMF-REST is an extension of Eclipse Modeling Framework and it allows to generate a REST API based on the model created with EMF. Apart from that, it also includes a JavaScript library which can be used for accessing the resources in the client application. This can again help with fast prototyping and documentation. This tool has a couple of disadvantages:

\begin{itemize}
  \item the amount of software that needs to be installed
  \item no endpoints management
  \item dependent on Tomcat
\end{itemize}

4.1.3 New Tool

If a new solution is suppose to be developed it shall combine all the best features of these two tools plus some unique characteristics. The following list are the features I have identified as critical:

1. flexible code generation
2. export model to a standardized format
3. graphical model representation
4. endpoints management
5. data structure management
6. documentation generation
7. no need for installing any programs

This report is about developing an editor, therefore I only consider points 2,3,4,5 and 7 to be relevant. I propose an HTML5/CSS3/JavaScript solution for the editor, which satisfies all the requirements on the editor mentioned above. Further research on which libraries and tools shall be used in this case needs to be done.

\textsuperscript{2}http://emf-rest.com/ - Homepage of the EMF-REST project
4.2 Research Question

The research question has been established as the following: Is it possible to have a modeling tool with a sophisticated notation which can capture all important information about REST API in the model and then use this model for creating the actual code without any advanced knowledge of server-side programming? This report is focusing on the first part of the question- a modeling tool with a sophisticated notation which can capture all important information about REST API in the model.

4.3 Development Stack Research

Since the world of web technologies is changing constantly, a research has to be done on what tools and libraries shall be used for the development. The criteria for evaluating has been established as:

- Who is responsible for the tool/library?
- How often is it being updated?
- What does the community say about the tool/library?
- How difficult is it to learn how to use the tool/library?

The development stack with selected tools/libraries will be established and released as an open source project\(^3\) so it can be used by the community in other projects. To prove that the development stack is mature enough to support bigger projects the editor shall be developed on the top of this stack. It shall also help to understand how the development stack is meant to be used and what are the advantages.

4.4 Editor

This report is focusing on the research of an HTML/CSS/JavaScript editor as a part of pre-research of Model-Driven Development of REST API. To evaluate that the editor is mature enough a case study as a research method shall be applied. I have been given the case from the company Searis AS\(^4\) which specializes on a custom made software including REST APIs. The case is a part of this document as an appendix. The research is considered to be successful only if the editor is capable to capture all the artifacts of the case so the ground for the code generation is prepared. The editor shall capture:

\(^3\)https://github.com/RassaLibre/ReactGCKarma - repository of the development stack including a manual how to install and use it

\(^4\)http://searis.no/ - official web page
• entities with the notation mentioned in the previous chapter
• data structure of each entity
• REST API endpoints containing the method and location for each entity
• relations between the entities

In order to be able to apply the code generation on the result model, the model shall be exported as a JavaScript Object Notation (JSON) data structure. The output shall reflect all the points mentioned above which means, that the export can be also used as an import data. This should result in the same model as the model before export. Import functionality is not part of the prototype and shall be implemented later.

4.5 Summary

This chapter described two main parts of the research: establishing a development stack and developing an editor. The criteria for choosing the right libraries and tools has been established. The evaluation of such development stack is subjective, therefore the feedback from the community will be taken as an evaluation. The research method for the editor in the second part has been established as a case study method. The case will be provided by an external company. In order to evaluate the editor, criteria and requirements have been established.
Chapter 5

Front-End Tool Research

This chapter contains an evaluation of available front-end tools and libraries. Apart from that a short description of how the tool or library is used in the project is given.

5.1 Data structures and behavior

JavaScript has a lot of weak spots which can easily lead to chaotic spaghetti code. Namely, it is the lack of namespaces and the lack of any static analysis. There are tools which can shrink the code \(^1\) by removing spaces and comments but none of them really checks the code itself.

There are several frameworks for building JavaScript applications such as AngularJS, EmberJS or BackboneJS. A framework is "a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes" [Johnson and Foote, 1988]. The problem with frameworks is that every JavaScript framework has a certain philosophy. If one does not share the philosophy the framework ends up to be an obstacle instead of a helpful tool. I have chosen to not follow the path of frameworks because:

- I do not want to force programmers to learn notations of the framework in order to extend or adjust the editor.
- Frameworks have a very bad support for other libraries and are generally meant to solve traditional problems.
- Many abstractions in frameworks are no longer necessary because of the fact that the gap between modern browsers is smaller and smaller.\(^2\)

Instead of a framework I decided to use a library. A library is "is essentially a set of functions that one can call organized into classes" [Fowler, 2005]. As Martin Fowler describes in his

\(^1\)For example online minifiers such as this one: http://javascript-minifier.com/ or more advanced UglifyJS - https://github.com/mishoo/UglifyJS

\(^2\)More information about browsers compatibility can be found here http://caniuse.com/
article "InversionOfControl", the biggest difference between a framework and a library is the inversion of control. This means that with libraries a user calls the library’s API and gets a result. With frameworks it is the opposite. The user inserts the code into the framework which then calls the code [Fowler, 2005].

A library fitting to my criteria described in 4.2 is Closure Library. The Closure Library is "a broad, well-tested, modular, and cross-browser JavaScript library. You can pull just what you need from a large set of reusable UI widgets and controls, and from lower-level utilities for DOM manipulation, server communication, animation, data structures, unit testing, rich-text editing, and more" [Google] and it is part of Closure Tools which includes compiler, templates and linter. Google is using these tools in all major applications such as Gmail, Google Docs, Google Calendar and others. The reasons for my choice were the following:

- The Library is maintained by Google.
- The Library has a very sophisticated dependency management.
- Closure Tools offer Closure Compiler for compiling and static analysis of my code.
- Closure Tools offer Closure Linter to maintain standardized coding style across the entire application.
- Only used parts in the library will be included in the resulted JavaScript file. Everything else is thrown away by the compiler.

Google Closure Library is a perfect choice. It has some truly unique features including a compiler with advanced optimization of JavaScript code to reduce the size of the resulted script, lints my code so it has a consistent style based on Google standards and is well tested on large projects used by millions of people all around the world.

5.2 DOM Manipulation

DOM (Domain Object Model) manipulation and how to do it properly, is one of the biggest challenges in web development. JavaScript and DOM are two completely different things and it is mainly the DOM API in JavaScript which is considered to be frustrating in the community.

Part of the Closure tools released and maintained by Google are Closure Templates. Since I decided to use Closure Library it would be logical to use Closure Templates. This tool allows to give templates a proper namespace and generate a JavaScript function which renders the HTML when called. The main problem with Closure Templates is that it contains an extensive set of tags for operations like looping over an array etc. specific only to Closure Templates. This means that an additional knowledge is required in order to develop components.

I have discovered React developed and maintained by Facebook. React is "a library for building composable user interfaces. It encourages the creation of reusable UI components".

---

3https://developers.google.com/closure/templates/
4http://facebook.github.io/react/
which present data that changes over time[3][Hunt]. React does not use any sort of a templates or any special notation. Thanks to the JSX preprocessor written in JavaScript it allows to write JavaScript code in JavaScript and HTML code in HTML at the same time while capturing both in .jsx files. React has a couple of other advantages like component life-cycle, both client-side and server-side rendering and its own Chrome Extension for debugging.

React has one feature which I find extremely useful in terms of the editor. Thanks to the fact that every component has life-cycle events, it is very easy to implement when a component should render. This is useful because in the case of a change in the model, only the components which are affected by the change shall re-render. This technique is called Selective Sub-Tree Rendering\(^5\). Specially in the case of huge models this feature can save a lot of performance.

React is maintained by Facebook, the feedback from the community is very positive and the library is under intensive development so future evolution is guaranteed. These are all reasons why I have chosen this library to my development stack.

## 5.3 Testing

Unit testing is a very important part of software development and it can save a lot of troubles and hours spent on manual testing. Google has a solution for testing called Karma. Karma is a "Test runner that makes testing JavaScript applications in real browsers frictionless and enjoyable[3][Jina] and is used as a testing library for AngularJS framework. Karma has some unique features such as testing in real browsers. This can be achieved by installing one of many extensions. I have also decided to use an extension for test coverage\(^6\). This extension will ensure that my code is properly tested and fully covered by unit tests. The code coverage report is included as an appendix.

Karma is developed and maintained by Google. It is a well tested tool and accepted in the community. All this made my choice very easy.

## 5.4 Quality of Code

JavaScript allows a lot to developers when writing code. This leads to different code styles arguments about whitespaces, tabs, semicolons etc. To be sure that my code is written in the most universal and clean way, Google offers Closure Linter\(^7\). This tool checks the code style and alerts the programmer in case of inconsistency. This leads to a consistent code style which is extremely important in bigger projects. Linter comes with the feature called AutoFix. AutoFix automatically corrects code style inconsistencies such as whitespaces, indenting etc. It also checks annotations for Closure Compiler which is very handy feature.

---

\(^5\) More information on Selective Sub-Tree Rendering can be found here: http://calendar.perfplanet.com/2013/diff/

\(^6\) http://karma-runner.github.io/0.8/config/coverage.html

\(^7\) More about Closure Linter here: https://developers.google.com/closure/utilities/
I have decided to include this tool to my development stack, because the editor will be released as an open source project and it is important that the code has the same coding style regardless of who is author of the code.

5.5 CSS Pre-processing

CSS pre-processing gained popularity because it extends the CSS language for some really useful features such as variables, simpler importing, nesting, mixins etc. It also produces CSS which is compatible with all major browsers. The most popular pre-processors are SASS, LESS and Stylus. This battle does not really have a winner. All the pre-processors have some unique features. I have chosen SASS because I believe it is the most mature and stable pre-processor for this purpose.

5.6 Automation

All the processes described above need to be executed manually after a certain time or ideally after every code change. Thanks to the tool called Grunt all the processes can be automated and triggered every time there is a change registered in my code. This saves a lot of time and it is extremely comfortable. Grunt allows to define different tasks. Every task is represented by an extension. Extensions can be installed via NPM. My main thread runs the following tasks every change registered in my code:

1. Compile React components and place them into the right namespace folder.
2. Fix styles to ensure code quality.
3. Check code styles because not every error can be fixed in the previous step.
4. Compile SASS files and create CSS files.
5. Calculate dependencies of my JavaScript files.
6. Watch for changes and if a change is registered, repeat the whole cycle.

Automation of all the processes is very useful and saves a lot of time during the development. The task above does not include testing because Karma has its own watcher so it runs in parallel with the defined Grunt task.

---

8More about the pre-processors can be found here: http://code.tutsplus.com/tutorials/sass-vs-less-vs-stylus-preprocessor-shootout-net-24320
9http://gruntjs.com/
10https://www.npmjs.org/
5.7 NodeJS

Even when everything runs in the browser, it is always good to run it against a server and serve the application to the browser. I have chosen Node for this purpose. Node is a "platform built on Chrome’s JavaScript runtime for easily building fast, scalable network applications"[Inc.] and it is a standard in the community of JavaScript developers. Since the role of the server is to serve the same static content to any kind of request regardless of what URL has been called there is no reason to do a further research.

5.8 Summary

Some of the choices for the development stack has been established. It is difficult to say what is actually the best set of tools, because a lot of choices rely on developer’s opinion. The development stack has been released as an open source project and can be found on the following URL: https://github.com/RassaLibre/ReactGCKarma. The main focus was to ensure a high quality testable code, which is easy to reuse and maintain. This was the main drive to include tools such as Closure Linter and Karma. The development stack in action can be seen on the picture below. I recommend to use 4 windows layout where 3 of the windows are used and the forth one is good for versioning commands.

![Figure 5.1: The development stack running in 3 terminal windows](image)

The top left window is running Node server, which serves all the static files. Top right window is running the Grunt task defined in the Automation section. This task is forced to re-run every time any change is registered. The bottom left window is running Karma for unit tests. Karma automatically detects changes and re-run the command every time any change is detected.
Chapter 6

Implementation

This chapter describes the implementation and some of the problems I had to face while developing the editor.

6.1 Root Directory

This section is describing the root directory of the entire application.

There are several configuration files and folders. `Gruntfile.js` is where the automation tasks are defined. `server.js` is a file with defined Node server. This server can be run from the terminal by the command `node server.js`. `package.json` is tightly coupled to the `node_modules/` folder. This JSON file specifies what kind of dependencies this project has. By running the command `npm install` all the dependencies are installed into the `node_modules/` folder. These dependencies are for example Karma or all Grunt extensions for the automation task. `sass/` is a folder where all the style files are. These files are compiled into one single `.css` file and moved to the `app/css/` folder. This means that the `app/css/` folder should not be touched by the programmer at all. Folder `karma/` contains the configuration file for the testing environment. Folder `jsx/` contains all React components which are compiled and placed in the `app/` folder.
Folder `coverage/` contains as the name indicates results of unit tests. These tests are divided based on in which browser the testing has been triggered. `closure-library` contains everything related to the Google Closure Library including JavaScript scripts the library is made out of but also Python scripts for calculating the dependencies and compiling itself. The last folder `app/` is the application itself. When the Node server starts, this folder is taken as server root and everything else starts here.

### 6.2 Namespaces

The codebase is divided into namespaces. This makes the structure clean and in a logical order. The root namespace has been chosen as `App` with the intention to change it when the name for this project is established.

The second level has three parts- `App.Core`, `App.Form` and `App.UIC`. `App.Core` is used for all the application logic. This includes two parts `App.Core.GUI` and `App.Core.Property`. `App.Core.GUI` is for all the elements that have a graphical representation like models, associations, properties of models etc. `App.Core.Property` is for different types of properties and their prototype representations are saved here (for example string, integer, double etc.).

`App.Form` and `App.UIC` are both generated React components and should never be edited directly. There is `jsx/` folder in the root of the project which contains all the components. `App.Form` is for forms. It contains forms for adding and editing components on the canvas. By separating all forms into one namespace I could reuse them for example in the Wizard component and then when adding new properties to an existing model. `App.Form.Property_Type` is related to the fact that almost every property has a special set of additional fields, for example when...
a property is a type of integer, then the form for entering minimal and maximal value needs to be shown. The additional fields are saved here.

App.UIC is a namespace for generated React components and it contains a graphical representation of elements in the App.Core.GUI namespace. App.UIC.Wizard is for the set of components which together create the Wizard feature. By isolating all Wizard related components into one namespace it is again extremely easy to use the same feature in some other projects or even replace the Wizard by some other component.

## 6.3 Data and UI layer Connection

The connection between the view and the data layer is simple and straightforward. Every UI component has a defined interface (blue rectangle on the figure below) of parameters which can be injected. ReactJS also supports validation of these parameters. This allowed me to be very accurate with what kind of data should be injected and in what kind of format.

The figure above shows a model `Button` which has a function `clicked` and an attribute called `name`. This is passed to the UI component, which then generates HTML code for the button with the references to the passed properties. This ensures, that the HTML code contains elements of the data model and is therefore the visual representation of the data model.

### 6.4 Calculating Associations Between Entities

The biggest challenge was to draw the associations between entities with pure HTML5/CSS3 without canvas or any other graphical library. Several steps need to be done in order to achieve
this:

1. figure out the mutual position of the entities
2. get the coordinates for the center points of the facing sides
3. calculate the distance between the two points
4. calculate the angle of the association
5. draw the association based on the calculated parameters

To figure out the mutual position of the entities I have established 8 possible positions. The facing sides are selected based on the position of the entities. For example if Entity2 is in position 3 from Entity1 then the facing sides are **bottom** and **right**. This gave me a good foundation to calculate the center points of the facing sides.

![Diagram of possible positions of two different entities](image)

**Figure 6.4: Possible positions of two different entities as a base for drawing associations**

Calculation of the center points is a straight forward task. In order to do this, every entity has a method which receives the side as a parameter and returns an object with x and y coordinates.

```java
/**
   * function calculates and returns a center point of the model
```

```
6.4. CALCULATING ASSOCIATIONS BETWEEN ENTITIES

To calculate the distance between the two center points I have used the Pythagoras sentence. This sentence is applied on right angled triangles. In this case the triangle is established by simply subtracting the coordinates of the two center points. For example if one center point has coordinates 400x200 and the second one 100x100 the sides of the triangle are simply 400-100 and 200-100.

```javascript
/*
 * center point is the middle on the passed side (left, right, top, bottom)
 * @param {String} side
 * @return {Object<string, string>}
 */

App.Core.GUI.Model.prototype.calculate_center_point = function(side) {
  var top, left;
  if (side === 'left') {
    top = this.top + (this.height / 2);
    left = this.left;
    return {top: top, left: left};
  }
  else if (side === 'right') {
    top = this.top + (this.height / 2);
    left = this.left + this.width;
  }
  else if (side === 'top') {
    top = this.top;
    left = this.left + (this.width / 2);
  }
  else {
    top = this.top + this.height;
    left = this.left + (this.width / 2);
  }
  return {top: top, left: left};
};
```

---

Figure 6.5: Triangle for calculations between two entities
The last information missing in the calculation is the angle so the association can be rotated and reach the second center point. The main interest is in the angle shown on the picture above as $\alpha$. To calculate this the arcsine function is used because all three sides are known. This calculation gives the angle in radians so the equation converting radians to degrees needs to be used. The last thing that needs to be considered is that the models have different mutual positions so in certain positions it is necessary to add or subtract 180 degrees. The code bellow shows the implementation of this step.

```javascript
1 /*
2 * function calculates the angle of the association between model1 and model2
3 * model1 is taken as a start so the angle can be between 0 to 360
4 * this should be adjusted so the angle is only between 0 and 180 and then it
5 * switches the beginning point. If so, the text will not be upside down
6 */
7 App.Core.GUI.Association.prototype.calculate_angle = function()
8 {
9     if (!this.length) this.calculate_length();
10     var model1_center_point = this.model1.calculate_center_point(this.model1_position);
11     var model2_center_point = this.model2.calculate_center_point(this.model2_position);
12     var top_distance = model1_center_point.top - model2_center_point.top;
13     var left_distance = model1_center_point.left - model2_center_point.left;
14     var alfa_rad = Math.asin(Math.abs(top_distance) / this.length);
15     var alfa_deg = Math.round(alfa_rad * (180 / Math.PI));
16     if (top_distance < 0) { //model1 is above model2
17         if (left_distance > 0) this.angle = 180 - alfa_deg; // model2 left under
18             else this.angle = alfa_deg; //model2 right under
19         }
20     else { //model2 is above model1
21         if (left_distance > 0) this.angle = 180 + alfa_deg; // model2 left top
22             else this.angle = 360 - alfa_deg; //model2 right top
23         }
24     }
25 }
```

The last step is to draw the association. To do this HTML `<p>` tag is used. Every association has a couple of CSS attributes which create the desired result. To draw the line `border: 1px solid black` CSS attribute is used. Then the calculated length is specified with `width` attribute.
To specify the position of the association, one of the center points from the entities is used as a starting point. The starting point is specified with properties `top` and `left`. CSS native behavior is that the transform origin is in the middle of the element. This needs to be changed to the left bottom corner of the element otherwise the calculations would be useless. This is done by CSS property `-webkit-transform-origin: left bottom;`. This will ensure that the rotation axis is in the bottom left corner of the element. Last property that needs to be applied is the angle. This is done with the CSS property `-webkit-transform: rotate(45deg)`. In this case, the association will have a 45 degrees incline. The complete set of properties can be seen in the picture below.

Figure 6.6: Triangle for calculations between two entities

6.5 Initial Rendering

Initial rendering is challenging because all calculations are done in the data layer. The problem is that the data layer does not know how big are the entities. Setting a fixed size is not a solution, because the text in the entity might overflow from the element. The technique used in the editor is that the entities are generated first. thanks to React component’s life cycle it is easy to catch the moment when the component is rendered. In this moment, the UI layer sends the information about the size of the entity to the data layer and the data layer finds associations related to the entity and recalculates them. When all the entities are rendered, the associations in the data layer have all the correct sizes and can be therefore rendered.
6.6 The Wizard

The Wizard is a set of React components which together create a well structured form for creating new entities. The component can be easily extended since its implementation is divided into separate components (navigation, content, input forms etc.). The Wizard is separated into three parts where every part represents different parts of the entity.

The first part on the figure is for general information. At the moment it contains only the field for entering the name of the entity. The second part is used for specifying the data structure. Wizard offers fields for one property, but this can be easily extended by clicking on the plus.
6.6. THE WIZARD

sign. This action will show another group of inputs for another property. Properties without any name will not be added to the data structure so no minus button is needed. The group of inputs consists of:

- text input for property name
- select for property type
- additional parameter linked to the property type

When a property type is selected additional fields are appearing based on the users choice. The application now supports 6 main data types with following fields:

- string (maximum length and regex)
- integer (minimum and maximum)
- double (minimum and maximum)
- array (key type and the value type)
- timestamp
- GeoJSON

Apart from these 6 choices, the user can also select an existing model as the property type. This is needed when an entity contains another entity (for example entity Order contains entity User as a property named customer). Wizard also supports the option where one entity has an array of type of some other entity (for example entity Cart can contain several entities of Item as a property named goods). This can be achieved by simply picking array as the property type and selecting an entity as a value.

The last section is dedicated to REST API endpoints. Wizard offers just one group of inputs which can be again extended by clicking on the plus sign. As with properties, endpoints without url will not be considered so no minus sign is needed. The input group consists of:

- select for HTTP method
- text field for url (for example users/)

If a parameter is needed in the url a colon is used before the parameter name for example users/:id. This is a standard notation used in modules such as Express\(^1\) for Node\(\text{JS}\)\(^2\) platform.

Wizard offers a nice structured way how to add new entities into the overall model. As a set of React components it can be easier extended. The components can be easily reused so if in the future the form for properties is needed it can be simply required in the future component and used.

\(^1\)http://expressjs.com/ - homepage of Express web framework

\(^2\)http://nodejs.org/ - Node\(\text{JS}\) homepage
6.7 Working with an Existing Model

This section describes how to work an existing model and what are the possible altering options in the current version of the editor.

The picture above shows an example of an entity called Location. This name can be changed by double clicking on the name. The name is changed into an input field so the new name can be entered. The same behavior is implemented with properties and associations. The name of any property can be changed by double clicking on the property name. The same counts for associations. By double clicking on the association name the user can enter a new one. The entire entity can be deleted by clicking on the trash bin icon. This is only possible if the entity has no associations to any other entity in order to keep consistency of the entire model. Any property can be deleted by clicking on the cross icon to the right in the property list. If the property is responsible for an association between two models (if the type of the property is another entity) then the association gets deleted.

The bottom part of each entity has two parts. The part on the left is for switching views between properties and endpoints. The first icon (list icon) serves as a button for showing properties while the second one (arrows icon) serves as a button to show endpoints. The plus button on the right enables the user to add new properties and endpoints. The right form is shown depending on which view is active in the time when the plus button is clicked.
Figure 6.10: Model with forms for adding properties and endpoints

The picture above shows two states—adding a new property on the left and adding a new endpoint on the right. The same React component for adding this information is used as in the Wizard. The only difference are CSS styles used in this case. If the form is shown, the plus button changes to a cross button. The form is closed and the entity is without any changes when the cross button is clicked. The current implementation unfortunately does not support altering of the property details. In that case, the whole property has to be deleted and added again with the desired property details.

6.8 Export to JSON

Probably the most important feature of the editor is that the model can be exported into JSON format. This is done by clicking on the Generate JSON button in the control panel on the left side of the screen (shown on the picture above). If the button is clicked, the control panel changes its content to a text field containing the JSON representation of the entire model.
The logic behind the generation is very simple. Every component (entity, association, property etc.) has a function `to_json()` which returns an object representing the component in JSON format. It is very easy to extend the exported information since everything is happening in one method of each exported component. Since this function respects the hierarchy of prototypes, if it is called on the `App.Core.GUI.Area` prototype a pyramid effect is triggered and the whole model is returned as a JSON data structure including two arrays—models and associations. As the name indicates, array `models` contains all the information about models including properties and endpoints and array `associations` contains all information about associations including between which entities is the association established. An example export can be seen in the A of this document.

6.9 Summary

In this chapter I have summarized the main issues, algorithms and functions I had to face in the development process. The idea of how to connect data layer with the UI layer has been described. React makes it really easy to create reusable components which I found extremely helpful once I had to reuse some component on a different place in the application (especially property and endpoint forms). Apart from that a description of how to create a new entity and how to work with the existing ones has been given. I have made a video showing the interaction with the editor. This video is available at https://www.youtube.com/watch?v=DlFlEdIXJpY. The implementation is available at http://tdt4501.bitballoon.com/ and the source code at https://github.com/RassaLibre/TDT4501.
Chapter 7

Application on the Case Study

This chapter is about applying the developed editor on a real case in order to be able to evaluate the editor and its usefulness in this stage of development. The case has been provided by the company Searis AS and is attached to this report as appendix B.

7.1 Analysis of the Case

The case is a REST API for an inventory system. This inventory system is used in one company producing custom ovens in Norway. The case consists of 4 data structures.

- Oven
- Pallet
- Part
- Location

To briefly summarize the case: Oven is a finished product made out of parts. Every oven has a location which specifies one and only place in the factory. Apart from ovens, each location can also contain a pallet. Each pallet can contain parts but can be also empty.

I have created a data model based on the description and detailed data structures listed in the case description.
There are several observations that need to be taken in account. First, this case covers several different types of associations. The editor should be able to represent this, as well as two different types of including one entity into another—directly or via an array. Another important thing to notice are IDs. This unique identifiers shall be automatically generated in incremental order in the REST API. 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.

The case also describes endpoints for the REST API. All the endpoints can be assigned to a data structure based on the location. The following picture shows the dividing of the endpoints mentioned in the case study.
7.2 Modeling in the Editor

A few assumptions had to be made in order to model the case. No closer specifications of the ranges for values are described in the case, so I had to come up with my own based on the name of the property. If for example the property is name `price` it is logical that it can not be negative. I based the regular expressions of string values on the given examples of data in the case. The end result can be seen on the figures below.

Complex endpoints such as `pallets/:id/parts/:part_id` will require research on how to be structured so the code generation engine knows to which entity which parameter belongs. This research is not part of this report but it will affect the way users enter the endpoints into the editor. It is therefore necessary to count on additional restrictions that might appear in the future. The important thing to mention is, that the editor is capable of modeling this case because all parts of the model are implemented.
Some of the properties contain for example `integer<0,->`. This means, that the value is validated only on its minimal value and that the maximal is not specified. The same counts for `string<30,->`. Such string has maximum length 30 and the regular expression is not specified. This means that the string value shall be validated only on its length.
7.3 Summary

The case has been successfully modeled in the editor. Several assumptions have been made because the case is not described into the necessary details (regular expressions, range for values of properties) and because unique identifiers are present in every entity, therefore it is better to move the responsibility to the code generation. The important thing stated in this chapter is that when the code generation is being researched, some additional constraints may pop up in order to keep the model consistent and ensure, that the code generation knows what for example parameters in location of endpoints mean. Details about how good the result model shows the case is discussed in the last chapter of this document. JSON output of the model is attached to this document as appendix A.
Chapter 8

Conclusion

The research question was—It possible to have a modeling tool with a sophisticated notation which can capture all important information about REST API in the model and then use this model for creating the actual REST API without any advanced knowledge of server-side programming? and I believe that the answer is positive. However the goal of this report was to answer just the first part of the question so there is still research to be done.

The question mentions sophisticated notation as one of the key points. I have suggested the notation based on the Relational data model notation, which glues together data structures and REST API endpoints in form of a closed entity. In the natural language, this notation says:"Here is a data structure and these are the endpoints of this REST API which allow you to access it".

Another key point in the question is modeling tool. I have developed a true HTML5/CSS3/JavaScript based editor, which supports the notation for REST APIs and allows to model entities and relations between them. I have defined criteria for such editor based on the existing tools and their weak sides. The criteria are:

1. export model to a standardized format
2. graphical model representation
3. endpoints management
4. data structure management
5. no need for installing any programs

Export model to a standardized format: This is achieved by the functionality of exporting the whole model to JSON file format. This will ensure that the data are computer readable and can be used as an input for the code generation.

Graphical model representation: This is achieved by using HTML5 and CSS3. Entities and associations have their own graphical representation so the user can interact with the entire model.

Endpoints management: Graphical representation of entities contains endpoints attached to
the data structure. This ensures that the user can add and delete endpoints as needed.

**Data structure management:** Graphical representation of entities also contains properties attached to the data structure. This ensures that the user can add and delete properties as needed.

**No need for installing any programs:** This has been achieved by using web technologies so everything runs in the browser and no additional programs or browser extensions are needed.

I have established a powerful development stack, which has been released as an open source code and anyone can use it (https://github.com/RassaLibre/ReactGCKarma). This step was necessary to make, in order to be sure that the development process and the code itself is based on the available tools in the web development community. To evaluate these tools, I have established a set of criteria. The problem with such a development stacks is that the choices can be very subjective. Therefore I can not say that this is the best way how to write web based applications. The feedback from the community is used as a measure.

I have used case study to evaluate the maturity of the editor. The case has been successfully modeled with my editor and my notation. However this process has shown few weak spots of the editor. First of all, the modeling has been done in Google Chrome 39.0.2171.65 (64 bit). This means that the editor is so far optimized only for Google Chrome. During the time of the development Google Chrome updated from 37 to 39. Therefore I believe that editor should be fully functional for Google Chrome in general. Another weak spot are associations. In the current state the associations does not show cardinality of the connection. This is partly shown by the property type. If the property type is *array* it is obvious that the property can have one or more objects of the certain type. The editor is however not able to capture if the one entity is required or not. It is also important to mention that sometimes the associations does not connect two entities perfectly. It is possible to sometimes observe a few pixels of free space between the end of the association and the entity. This is due to the rounding in the length calculation. This can be solved by showing the association in the layer below models and make it longer for a certain constant. For the sake of research I decided to not do it and keep the solution clean.

Further research should be done on how to implement zooming and scrolling into the editor so the model is not limited by the size of the screen. Another topic could include how to differ between which endpoints and properties are available to an ordinary user and which require authentication token. This is necessary in REST APIs which include authorization. Probably the most interesting field for research now is how to generate a fully functional REST API based on the information from the model.

Unfortunately I still can not answer the question mentioned in the introduction but I can say that this report describes the first step towards the solution of developing REST APIs. I believe that further research on the code generation will give a solid answer.
Bibliography


Appendix A

JSON output

```json
1 { 
2   "models": [ 
3     { 
4       "name": "Oven", 
5       "id": ":0",
6       "properties": [ 
7         { 
8           "id": ":1",
9           "name": "SKU",
10          "type": { 
11            "type": "string",
12            "length": 30,
13            "regex": "^[A-Z1-9]{2}-[A-Z1-9]{2}$"
14          }
15        },
16        { 
17           "id": ":2",
18           "name": "name",
19           "type": { 
20              "type": "string",
21              "length": 30
22           }
23        },
24        
25           "id": ":3",
26           "name": "parts",
27           "type": { 
28              "type": "array",
29              "key": "integer",
30              "value": "Part"
31           }
32        }
33   }
34 }
```

a-1
```json
{
  "id": ":4",
  "name": "status",
  "type": {
    "type": "string",
    "length": 30
  }
},
{
  "id": ":5",
  "name": "createdAt",
  "type": {
    "type": "timestamp"
  }
},
{
  "id": ":6",
  "name": "updatedAt",
  "type": {
    "type": "timestamp"
  }
},
{
  "id": ":7",
  "name": "location",
  "type": {
    "type": "Location"
  }
},
"endpoints": [  
  {
    "id": ":w",
    "type": "GET",
    "url": "ovens/"
  },
  {
    "id": ":x",
    "type": "GET",
    "url": "ovens/:id"
  },
  {
    "id": ":y",
    "type": "PUT",
    "url": "ovens/:id"
  },
]}
```
```json
{
  "id": "z",
  "type": "DELETE",
  "url": "ovens/:id"
},
{
  "id": ":10",
  "type": "POST",
  "url": "ovens/"
}
]
}
{
  "name": "Pallet",
  "id": ":8",
  "properties": [
    {
      "id": ":9",
      "name": "SKU",
      "type": {
        "type": "string",
        "length": 30,
        "regex": "^[A-Z1-9]{2}-[A-Z1-9]{2}$"
      }
    },
    {
      "id": ":a",
      "name": "parts",
      "type": {
        "type": "array",
        "key": "integer",
        "value": "Part"
      }
    },
    {
      "id": ":b",
      "name": "location",
      "type": {
        "type": "Location"
      }
    },
    {
      "id": ":c",
      "name": "createdAt",
      "type": {
        "type": "timestamp"
      }
    }
  ]
}
```


```json
{
  "id": ":d",
  "name": "updatedAt",
  "type": {
    "type": "timestamp"
  }
}

"endpoints": [
  {
    "id": ":11",
    "type": "GET",
    "url": "pallets/"
  },
  {
    "id": ":12",
    "type": "GET",
    "url": "pallets/:id"
  },
  {
    "id": ":13",
    "type": "PUT",
    "url": "pallets/:id"
  },
  {
    "id": ":14",
    "type": "DELETE",
    "url": "pallets/:id"
  },
  {
    "id": ":15",
    "type": "POST",
    "url": "pallets/"
  },
  {
    "id": ":16",
    "type": "POST",
    "url": "pallets/:id/parts"
  },
  {
    "id": ":17",
    "type": "GET",
    "url": "pallets/:id/parts"
  }
]
{
  "id": ":18",
  "type": "GET",
  "url": "pallets/:id/parts/:part_id"
},
{
  "id": ":19",
  "type": "PUT",
  "url": "pallets/:id/parts/:part_id"
},
{
  "id": ":1a",
  "type": "DELETE",
  "url": "pallets/:id/parts/:part_id"
}
],
{
  "name": "Location",
  "id": ":e",
  "properties": [
    {
      "id": ":f",
      "name": "SKU",
      "type": {
        "type": "string",
        "length": 30,
        "regex": "[A-Z]{1}[1-9]{2}-[1-9]{2}\$"
      }
    },
    {
      "id": ":g",
      "name": "name",
      "type": {
        "type": "string",
        "length": 30
      }
    },
    {
      "id": ":h",
      "name": "geoLocation",
      "type": {
        "type": "geojson"
      }
    }
  ]
}
"id": "i",
"name": "createdAt",
"type": {
    "type": "timestamp"
}
},
{
    "id": "j",
    "name": "updatedAt",
    "type": {
        "type": "timestamp"
    }
}
],
"endpoints": [
    {
        "id": "1b",
        "type": "GET",
        "url": "locations/"
    },
    {
        "id": "1c",
        "type": "GET",
        "url": "locations/:id"
    },
    {
        "id": "1d",
        "type": "PUT",
        "url": "locations/:id"
    },
    {
        "id": "1e",
        "type": "DELETE",
        "url": "locations/:id"
    },
    {
        "id": "1f",
        "type": "POST",
        "url": "locations/"
    }
]
},
{
    "name": "Part",
    "id": "k",
    "properties": [
        
    ]
}
{  
  "id": ":l",
  "name": "SKU",
  "type": {
    "type": "string",
    "length": 30,
    "regex": "^[1-9]{2}[-][A-Z1-9]{2}[-][A-Z1-9]{2}$"
  }
},

{  
  "id": ":m",
  "name": "name",
  "type": {
    "type": "string",
    "length": 30
  }
},

{  
  "id": ":n",
  "name": "count",
  "type": {
    "type": "integer",
    "min": 0
  }
},

{  
  "id": ":o",
  "name": "price",
  "type": {
    "type": "double",
    "min": 0
  }
},

{  
  "id": ":p",
  "name": "value",
  "type": {
    "type": "double",
    "min": 0
  }
},

{  
  "id": ":q",
  "name": "createdAt",
  "type": {"type": "timestamp"}}
"updatedAt": {
  "id": ":r",
  "name": "updatedAt",
  "type": {
    "type": "timestamp"
  }
}
],
"endpoints": [
  {
    "id": ":lg",
    "type": "GET",
    "url": "parts/"
  },
  {
    "id": ":lh",
    "type": "GET",
    "url": "parts/:id"
  },
  {
    "id": ":li",
    "type": "PUT",
    "url": "parts/:id"
  },
  {
    "id": ":lj",
    "type": "DELETE",
    "url": "parts/:id"
  },
  {
    "id": ":lk",
    "type": "POST",
    "url": "parts/"
  },
  {
    "id": ":ll",
    "type": "GET",
    "url": "parts/:id/pallets"
  },
  {
    "id": ":lm",
    "type": "POST",
    "url": "parts/:id/pallets"
}
"associations": [
{ "id": "s", "name": "lays on", "model1": { "name": "Part", "id": "k" }, "model2": { "name": "Pallet", "id": "s" } },
{ "id": "t", "name": "made out of", "model1": { "name": "Oven", "id": "0" }, "model2": { "name": "Part", "id": "k" } },
{ "id": "u", "name": "positioned at", "model1": { "name": "Pallet", "id": "s" }, "model2": { "name": "Location", "id": "e" } },
{ "id": "v", "name": "positioned at", "model1": { "name": "Oven", "id": "0" } } ]
"model2": {
  "name": "Location",
  "id": "e"
}
]}
}
Appendix B

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
<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>1</td>
</tr>
<tr>
<td>sku</td>
<td>AA-AA</td>
</tr>
<tr>
<td>name</td>
<td>OCTO 50</td>
</tr>
<tr>
<td>status</td>
<td>delivered</td>
</tr>
<tr>
<td>created_at</td>
<td>2014-11-13T15:15:38Z</td>
</tr>
<tr>
<td>updated_at</td>
<td>2014-11-19T13:52:50Z</td>
</tr>
<tr>
<td>meta.href</td>
<td>/inventory/ovens/1</td>
</tr>
<tr>
<td>meta.mediatypes</td>
<td>application/json;application/xml</td>
</tr>
<tr>
<td>parts.meta.href</td>
<td>/inventory/ovens/1/parts</td>
</tr>
<tr>
<td>parts.meta.mediatypes</td>
<td>application/json</td>
</tr>
<tr>
<td>location.meta.href</td>
<td>/inventory/locations/1</td>
</tr>
<tr>
<td>location.meta.mediatypes</td>
<td>application/json</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Method</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>parts/</td>
</tr>
<tr>
<td>GET</td>
<td>parts/:id</td>
</tr>
<tr>
<td>POST</td>
<td>parts/</td>
</tr>
<tr>
<td>PUT</td>
<td>parts/:id</td>
</tr>
<tr>
<td>DELETE</td>
<td>parts/:id</td>
</tr>
<tr>
<td>GET</td>
<td>parts/:id/pallets</td>
</tr>
<tr>
<td>POST</td>
<td>parts/:id/pallets</td>
</tr>
<tr>
<td>GET</td>
<td>pallets/</td>
</tr>
<tr>
<td>GET</td>
<td>pallets/:id</td>
</tr>
<tr>
<td>POST</td>
<td>pallets/</td>
</tr>
<tr>
<td>PUT</td>
<td>pallets/</td>
</tr>
<tr>
<td>DELETE</td>
<td>pallets/:id</td>
</tr>
<tr>
<td>GET</td>
<td>pallets/:id/parts</td>
</tr>
<tr>
<td>POST</td>
<td>pallets/:id/parts</td>
</tr>
<tr>
<td>GET</td>
<td>pallets/:id/parts/:part_id</td>
</tr>
<tr>
<td>PUT</td>
<td>pallets/:id/parts/:part_id</td>
</tr>
<tr>
<td>DELETE</td>
<td>pallets/:id/parts/:part_id</td>
</tr>
<tr>
<td>GET</td>
<td>locations/</td>
</tr>
<tr>
<td>GET</td>
<td>locations/:id</td>
</tr>
<tr>
<td>POST</td>
<td>locations/</td>
</tr>
<tr>
<td>PUT</td>
<td>locations/:id</td>
</tr>
<tr>
<td>DELETE</td>
<td>locations/:id</td>
</tr>
<tr>
<td>GET</td>
<td>ovens/</td>
</tr>
<tr>
<td>Method</td>
<td>URL</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>GET</td>
<td>ovens/:id</td>
</tr>
<tr>
<td>POST</td>
<td>ovens/</td>
</tr>
<tr>
<td>PUT</td>
<td>ovens/:id</td>
</tr>
<tr>
<td>DELETE</td>
<td>ovens/:id</td>
</tr>
</tbody>
</table>
Appendix C

Test Coverage

Parts Form/, Form/Property_Type/, UIC/ and UIC/Wizard/ are React components and these test coverage report do not cover testing of the components.
Figure C.1: Test coverage of the editor

<table>
<thead>
<tr>
<th>File</th>
<th>Statements</th>
<th>Branches</th>
<th>Functions</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core/GUI/Wea/</td>
<td>83.33% (275 / 330)</td>
<td>59.19% (18 / 31)</td>
<td>84.09% (27 / 34)</td>
<td>84.91% (370 / 318)</td>
</tr>
<tr>
<td>Core/GUI/Association/</td>
<td>86.84% (152 / 152)</td>
<td>60.53% (23 / 38)</td>
<td>92.31% (24 / 31)</td>
<td>88.44% (130 / 147)</td>
</tr>
<tr>
<td>Core/GUI/Endpoint/</td>
<td>91.43% (52 / 52)</td>
<td>100.00% (0 / 0)</td>
<td>78.92% (10 / 13)</td>
<td>91.43% (32 / 36)</td>
</tr>
<tr>
<td>Core/GUI/Model/</td>
<td>63.97% (131 / 150)</td>
<td>69.75% (29 / 42)</td>
<td>85.71% (26 / 30)</td>
<td>85.55% (127 / 152)</td>
</tr>
<tr>
<td>Core/GUI/Property/</td>
<td>96.89% (31 / 32)</td>
<td>50.00% (1 / 2)</td>
<td>90.91% (10 / 11)</td>
<td>96.89% (31 / 32)</td>
</tr>
<tr>
<td>Core/Property/</td>
<td>86.54% (110 / 206)</td>
<td>42.86% (18 / 43)</td>
<td>77.55% (28 / 49)</td>
<td>86.89% (179 / 206)</td>
</tr>
<tr>
<td>Form/</td>
<td>32.26% (10 / 31)</td>
<td>0.00% (0 / 0)</td>
<td>22.22% (2 / 9)</td>
<td>32.26% (10 / 31)</td>
</tr>
<tr>
<td>Form/Property_Type/</td>
<td>52.63% (10 / 19)</td>
<td>0.00% (0 / 0)</td>
<td>44.44% (4 / 9)</td>
<td>50.00% (6 / 12)</td>
</tr>
<tr>
<td>LIC/</td>
<td>13.58% (53 / 405)</td>
<td>0.00% (0 / 0)</td>
<td>7.45% (7 / 94)</td>
<td>13.58% (53 / 390)</td>
</tr>
<tr>
<td>UIC/Wizard/</td>
<td>22.08% (17 / 77)</td>
<td>0.00% (0 / 0)</td>
<td>20.00% (4 / 20)</td>
<td>23.29% (17 / 75)</td>
</tr>
</tbody>
</table>
Appendix D

Entire Environment in the Browser

Figure D.1: The whole environment in the browser