UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Integration of "JSONForms" with the OpenAPI Specification

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Héctor Fornes Gabaldón

Tutor: Francisco Daniel Muñoz Escoí

Curso 2015-2016

Abstract

The goal of this thesis is the integration of JSONForms with the OpenAPI Specification to automatically create web UIs that communicate with REST services.

JSONForms is an AngularJS framework which allows describing web form-based UIs in a declarative way (using JSON and JSON Schema) rather than coding them manually with HTML.

This project adds to JSONForms the capability to communicate with the backend, automatically deriving all the needed widgets from an OpenAPI definition.

Keywords : JSONForms, OpenAPI Specification, backend.

Resumen

El objetivo de este Trabajo de Fin de Grado es la integración de JSONForms con la OpenAPI Specification, para crear automáticamente interfaces de usuario web capaces de comunicarse con servicios REST.

JSONForms es una librería para AngularJS que permite describir interfaces de usuario web basadas en formularios de forma declarativa (utilizando JSON y JSON Schema), en lugar de programándolas manualmente con HTML.

Este proyecto añade a JSONForms la capacidad de comunicarse con un servidor, derivando de forma automática todos los “widgets” necesarios a partir de una definición OpenAPI.

Palabras clave : JSONForms, OpenAPI Specification, servidor.

Table of contents

Table of figures

1. Introduction

There exist different theories about the origin of the language, but they all agree on one thing: the emergence of language was a revolution for the humanity. Communication allowed the creation of increasingly complex societies, and was an ideal tool for the thought and the knowledge transmission. With the emergence of the agriculture and the first commercial exchanges, oral language started to show some limitations. It was necessary to gather some information, structure and store it so that it was possible to access it again. That's how writing was invented.

Since then, many improvements have appeared, such as the invention of the ink, the paper or the printing. But it has been in the last 100 years when our way of treating and processing the information has radically changed. Computers, mobile devices and the Internet have led to the Information Age [1], in which virtually everything is digitalized and open to everybody.

One of the examples of this digitization, and getting closer to the topic of this thesis, are forms. A form is simply a document in which a user must introduce some data in a structured way. They first appeared in the 19th century to simplify the task of drafting legal documents or gathering technical data in the factories [2]. And if before all of them were in paper, now it's the digital format which predominates. Applications for booking hotel rooms or others of business administration, just to say some, virtually all the applications need forms.

However, programming forms is not that easy, as it entails certain difficulties:

- There exist relations and dependencies among different fields.
- Some of the introduced data needs validation.
- They are subject to continuous changes in their structure and design.

In order to solve these and other problems, some libraries and frameworks have been appearing over the years, easing and speeding the process of creating form-based user interfaces.

This is the case of JSONForms [3], an AngularJS framework developed by EclipseSource München, the enterprise where I have spent my internship for the last semester. JSONForms allows the creation of web forms by defining them in a declarative way, using JSON and JSON Schema. Basically, the web developer includes the JSONForms directive in the HTML code, and supplies it with 3 schemas:

- The Data Schema, which represents the model of the form. This model is a definition of the different fields of the form: name, type, format, possible values, etc.
- The UI Schema, which describes the UI of the form, i.e. the layout structure and the position of the different fields.

- The actual data object, which contains the information with which the form will be filled.

From these schemas, JSONForms is capable of generating the needed HTML code and actually rendering a fully functional and nice-looking web form. JSONForms also has other interesting features as data-binding, input validation and rule-based visibility. It can also be extended with custom widgets and styles.

Until now, we have analysed how the Information Age has changed the process of information gathering, focusing our attention in web forms. However, it has also brought a revolution to many other aspects, like information storage and sharing. When Internet was invented [4], it only consisted of a couple of computers located in different universities and sharing pieces of scientific researches. But now it contains billions of web pages, and it's used by millions of users daily. This incredible growth has been possible due to many reasons. One of them has special interest for this thesis: the global adoption of common standards and protocols defining the functioning of the Internet. This allowed the connection of different kinds of machines, as well as the exchange of all types of data. Some examples of these protocols are IP, TCP, DNS, HTTP, etc.

Another example, and introducing one of the main topics of this thesis, is REST [5]. REST has become the software architectural style of the World Wide Web, overcoming other approaches like SOAP or RPC. This is due to many reasons: it was designed to work with HTTP seamlessly, it makes web services simpler to design and more scalable, etc. Web servers need application programming interfaces to the different resources and services they offer. In RESTful services [6], these APIs basically consist in a set of HTTP requests and their associated responses.

However, RESTful APIs used to be described in heterogeneous ways, which complicated their understanding by potential consumers and incremented the amount of implementation logic needed to interact with different services. In order to solve these problems and standardize the process of defining RESTful APIs, some proposals emerged. The most popular one is the "OpenAPI Specification" [7].

So now that JSONForms and the OpenAPI Specification have been introduced and put into context, it's time to state the purpose of this thesis: build a web application that, given the OpenAPI definition of a web service, automatically creates a web UI which allows a user to interact with the resources of that service and to understand its different capabilities. JSONForms is used to create those UIs. From now on, we will refer to this application as "jsonforms-swagger". The idea behind it and its main requirements were conceived at EclipseSource München.

In this thesis we describe the whole process of building jsonforms-swagger, as well as we discuss the different challenges and problems experienced during that process. The specific sections in which the rest of this document is structured are:

- General concepts: overview of the main ideas and the basic notions analysed in this thesis.

- Related work: list of other RESTful APIs definition approaches and other declarative form-based UIs generators. Brief description of the most interesting ones.

- Applicability: possible uses and benefits of jsonforms-swagger

- Requirements: description of the process from the conceptual idea of the application to the formal model of all its functional and non-functional requirements.

- Design: discussion of the chosen system architecture and technologies, as well as the list of used design patterns

- Implementation: analysis of the different implementation tasks, together with their problems and challenges.

- Acquired competences: description of the knowledge and competences acquired during the development of jsonforms-swagger.

- Conclusions: short review of the main topics covered in the previous sections, objectives achieved and what can be done in the future.

2. General concepts

This section discusses the general concepts that will be useful for the project. They should be fully understood before reading on.

Angular

Angular [8] is a development platform for building mobile and desktop web applications. It includes features like two-way data binding, dependency injection and dynamic templates.

Typescript

Typescript [9] is a superset of Javascript based on the concepts of Object Oriented Programming. Its main feature is the introduction of types. It gets compiled to Javascript by the build tool.

JSON Schema

JSON Schema [10] is a JSON based format for defining the structure of JSON data. JSON Schema provides a contract for what JSON data is required for a given application and how to interact with it. JSON Schema is intended to define validation, documentation, hyperlink navigation, and interaction control of JSON data.

JSONForms

JSONForms [3] is a web framework based on Angular that generates fully functional web forms from JSON definitions. Some of its extra features are conditional rendering based on rules and data validation.

REST

REST [5] stands for Representational State Transfer. It's an architectural pattern widely used in the web, as it establishes a lightweight connection (generally based on HTTP) between the server and the clients.

OpenAPI Specification / Swagger

The OpenAPI Specification [7] is an emerging standard that describes REST APIs. Its goal is to allow both humans and machines to understand web services without needing access to its source code.

3. Related work

3.1. Specification formats for REST APIs

There exist other specification formats for defining REST APIs apart from the OpenAPI Specification. They all share a similar idea and structure, but differ in some syntax details and in their level of adoption.

The most popular alternative to the OpenAPI Specification is RAML (RESTful API Modeling Language) [11]. Its definitions are written in YAML, a more human-readable language that extends JSON. This format makes it easier to represent the hierarchy of the operations and to reuse some parts of the code. It also has some interesting tooling: automatic documentation generator, online editor, code generator for Java and HTTP, etc. However, the OpenAPI Specification has even more support and tooling, because it was the format chosen by the Linux Foundation.

Other alternative is API Blueprint [12]. Its main difference is that it uses Markdown as its specification language, which makes it easier to learn and to understand. However, its level of adoption is low, so it lacks a lot of tooling and more community support.

3.2. Declarative form-based UIs generators

JSONForms is not the only solution trying to simplify the process of coding forms. There exist more frameworks that use declarative languages for defining the model and the layout of the forms.

The first solution to be analysed is EMF Forms [13], also developed by EclipseSource. JSONForms principles and features are based on EMF Forms. But while EMF Forms is designed for building desktop UIs, JSONForms is basically an adaptation for the web. The main difference between them is the language used to describe the model. As its name suggests, EMF Forms uses Ecore (the metamodel of Eclipse Modeling Framework) instead of JSON. The level of adoption, tooling and support of EMF Forms is much higher, as JSONForms is a relatively new project and a derivation from the first one.

In the field of web development, and with a very similar approach and name, there is JSON Form [14]. In this case, it's a general Javascript client-side library, unlike JSONForms which is specific for Angular. Other difference is that JSON Form defines the model and the layout of the form in the same schema, while JSONForms uses two separated schemas. JSON Form has more available widgets, like for example submit buttons. A downside is that it's more difficult to extend and customize, as JSONForms was designed specifically with this purpose in mind.

3.3. UIs generators based on an OpenAPI definition

The most similar solution to jsonforms-swagger is Swagger UI [15]. It’s a project from the same developers which created the OpenAPI Specification. It has the same main functionality than jsonforms-swagger: automatically generate a UI which represents an OpenAPI definition and is capable of communicating with its associated REST service. Part of the UI decisions of jsonforms-swagger are based on the style of the UIs generated by Swagger UI. For example, the idea of using a sandbox for showing and hiding the details of a specific entity type.

However, jsonforms-swagger introduces some additional features:

- It supports the customization of the generated UI, allowing the user to choose which operations to show and how to group them. Swagger UI just includes all the operations listed in the OpenAPI definition, what can be a problem for really big definitions with hundreds of operations.
- Besides that, jsonforms-swagger includes a sidebar for better organizing the UI, letting more available space for the interaction with the generated forms.

4. Applicability

This section provides a list of the possible uses and the main benefits offered by jsonforms-swagger.

The main feature of jsonforms-swagger is that it provides a visual representation of the OpenAPI definition of a RESTful service. In this way, a potential consumer of that service can understand what it offers and how to use it much more easily than reading a JSON file.

Jsonforms-swagger also allows the interaction with the different endpoints and resources of a backend, through its defined operations. Thanks to this functionality, a user can consume the services offered by the backend in a simple way. And this can be also useful for the backend maintainer. He can test the consistency of the actual implementation of the backend with its OpenAPI definition, which might not be up-to-date.

Another feature of jsonforms-swagger is that it generates a UI based on an OpenAPI definition automatically, and then it allows to customize this UI. This implies some benefits for web developers. A developer can choose which operations to show and how to group them, and then include the generated UI in his web site. Also, as the UI is automatically generated and it's totally functional, the developer saves a lot of implementation time and effort.

5. Requirements

5.1. Elicitation

In the first stages of the development process, I had some meetings with my supervisor in EclipseSource München in which we discussed the purpose and the requirements of the application using scenarios, use cases and mockups.

Scenarios

After discussing some possible usage scenarios of the application, we agreed on having two different actors:

- User: simply uses the application to interact with a backend through a form.
- Developer: wants to generate a custom form-based UI capable to interact with an OpenAPI backend.

In order to give form to the idea of the application, we used as a basis the "Swagger UI" demo (http://petstore.swagger.io/). It basically consists in a client which interacts with a simulated pet store backend. The next scenarios are based on the same "pet store" example too, and they illustrate how the actors described previously would use the application:

Scenario name	changePetName
Participating actor instances	John: User
Flow of events	1. John is using a jsonforms-swagger instance that interacts with a pet store backend. He wants to change the name of a pet with a known id. 2. He finds the pet with that id, and its information is received from the backend and shown in a form. 3. He changes the name of the pet and the information is stored back to the backend.

Scenario name	generateViewForUpdatingPets
Participating actor instances	Bob: Developer
Flow of events	1. Bob is developing a web application which needs to communicate with the backend of a pet store. 2. He introduces the API url of that backend in jsonforms-swagger, and some actions and their respective forms are automatically generated. 3. He wants to add an action for updating pets which has not been automatically generated. So he enters the name of the action and selects the corresponding API operation. 4. He gets a form UI capable of updating pets, and ready to be used by his web application users.

It’s worth to say that these scenarios were not only used conceptually, but they have been actually implemented in jsonforms-swagger.

Use cases

The next use cases are an abstraction of the previous scenarios examples. Each of them describe a class of similar scenarios:

Use case name	CreateEntity
Participating actors	Initiated by User
Flow of events	1. The User selects a “create” action within an entity type. 2. jsonforms-swagger presents to the User a form with all needed fields to create an entity. 3. The User completes the form and clicks on “Create”. 4. jsonforms-swagger sends the information to the backend and notifies the User if the operation has been successful or not.

Use case name	UpdateEntity
Participating actors	Initiated by User
Flow of events	1. The User selects an “update” action within an entity type. Alternatively, the User performs the “FindEntity” use case 2. jsonforms-swagger presents to the User a form with all needed fields to update an entity. If the User performed the “FindEntity” use case, the form will already contain the information of the found entity. 3. The User completes / changes the required / desired fields of the form and clicks on “Update”. 4. jsonforms-swagger sends the information to the backend and notifies the User if the operation has been successful or not.

Use case name	DeleteEntity
Participating actors	Initiated by User
Flow of events	1. The User selects a “delete” action within an entity type. Alternatively, the User performs the “FindEntity” use case 2. jsonforms-swagger presents to the User a form with all needed fields to delete an entity. If the User performed the “FindEntity” use case, the form will already contain the information of the found entity. 3. The User completes the required fields of the form and clicks on “Delete”. 4. jsonforms-swagger sends the information to the backend and notifies the User if the operation has been successful or not.

Use case name	FindEntity
Participating actors	Initiated by User
Flow of events	1. The User selects a “find” action within an entity type. 2. jsonforms-swagger presents to the User a form with all needed fields to find an entity. 3. The User completes the required fields of the form and clicks on “Find”. 4. jsonforms-swagger sends the information to the backend and presents another form with the information of the found entity. If more than one entities have been found, jsonforms-swagger presents them in a table, so the user can select one of them.

Use case name	CreateProject
Participating actors	Initiated by Developer
Flow of events	1. The Developer clicks on the button “Create Project” and inserts a name and the API url. 2. jsonforms-swagger creates a project with the inserted name. jsonforms-swagger also gets an API specification from the inserted url and generates some entity types, actions and operations based on it.

Use case name	EditProject
Participating actors	Initiated by Developer
Flow of events	1. The Developer selects a project among the existing projects. 2. jsonforms-swagger present to the Developer the different entity types of the selected project, and a button to export it.

Use case name	AddEntityType
Participating actors	Initiated by Developer
Flow of events	1. The Developer performs the “EditProject” use case. He clicks on the button “Add Entity Type” and inserts a name. 2. jsonforms-swagger adds an entity type with the inserted name to the project.

Use case name	EditEntityType
Participating actors	Initiated by Developer
Flow of events	1. The Developer performs the “EditProject” use case. He selects an entity type. 2. jsonforms-swagger presents to the Developer the different actions and the properties of the selected entity type, and a button to add new actions. 3. The Developer changes the desired properties. 4. jsonforms-swagger saves the changes and acknowledges the Developer.

Use case name	AddAction
Participating actors	Initiated by Developer
Flow of events	1. The Developer performs the “EditEntityType” use case. He clicks on the button “Add Action” and inserts a name. 2. jsonforms-swagger adds an action with the inserted name to the selected entity type.

Use case name	EditAction
Participating actors	Initiated by Developer
Flow of events	1. The Developer performs the “EditEntityType” use case. He selects an action within the selected entity type. 2. jsonforms-swagger presents to the Developer the different operations and the properties of the selected action, and a button to add new operations. 3. The Developer changes the desired properties. 4. jsonforms-swagger saves the changes and acknowledges the Developer.

Use case name	AddOperation
Participating actors	Initiated by Developer
Flow of events	1. The Developer performs the “EditAction” use case. He clicks on the button “Add Operation” and selects one from the list. Alternatively, he can drag&drop an operation to the selected action container. 2. jsonforms-swagger adds the operation to the selected action.

Use case name	EditOperation
Participating actors	Initiated by Developer
Flow of events	1. The Developer performs the “EditAction” use case. He selects an operation within the selected action. 2. jsonforms-swagger presents to the Developer the different the properties of the selected operation. 3. The Developer changes the desired properties. 4. jsonforms-swagger saves the changes and acknowledges the Developer.

Use case name	ExportProject
Participating actors	Initiated by Developer
Flow of events	1. The Developer performs the “EditProject” use case. He clicks on the button “Export Project”. 2. jsonforms-swagger generates the information to reproduce the shown forms, so it can be embedded in an existing application.

The next use case diagram includes all previous use cases and presents a whole picture of the system:

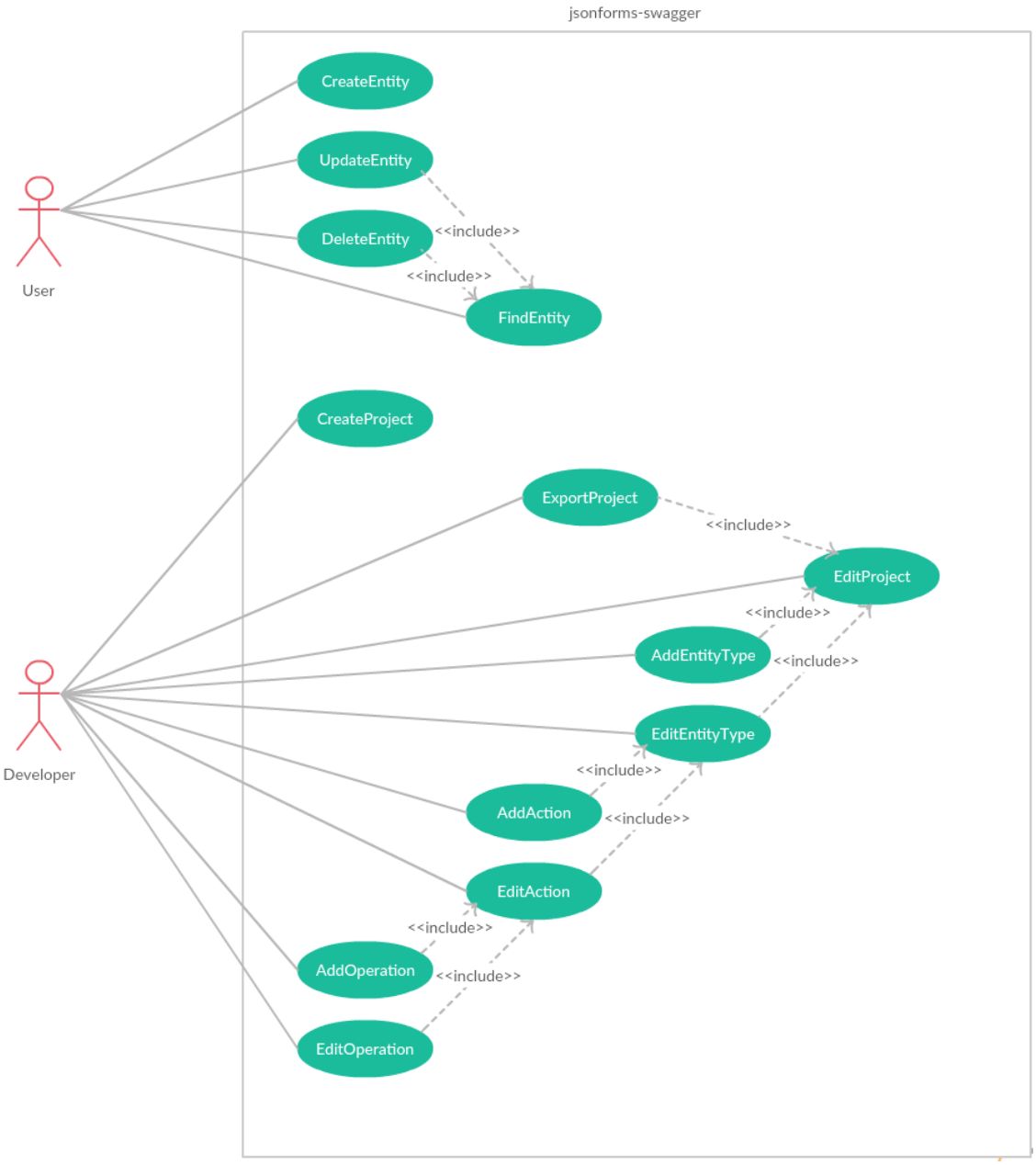

Figure 1: Use case diagram

Mockups

In order to validate if my view of the system was the same than my supervisor's view of the system, I created some mockups representing the functionality of the application. I created different alternatives with different UI structures. Then I showed them to my supervisors, and I made some measurements while they were testing the different mockups. These tests helped to identify omissions and misunderstandings in the requirements list, as well as to choose between the different UI alternatives.

Here are the different alternatives with their respective mockups:

- "Sidebar" alternative. The user can navigate between actions using the sidebar on the left. Actions are grouped in dropdowns, and the selected action is highlighted.

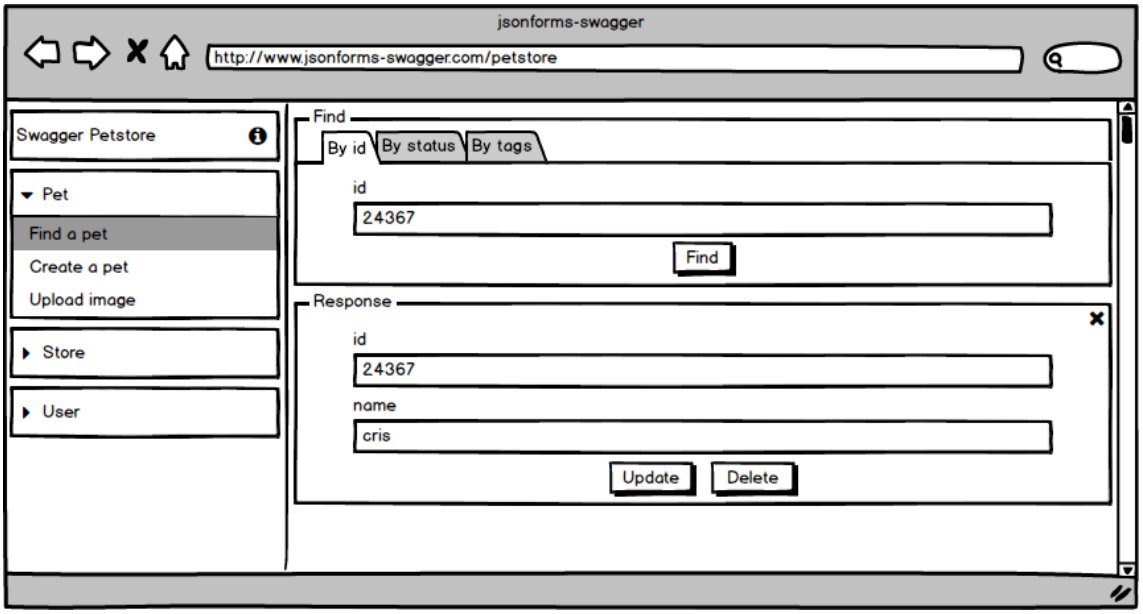

Figure 2: "Sidebar" mockup

- "Single-action" alternative. It's similar to the "sidebar" alternative, because it also contains a sidebar on the left. However, the main difference is that it mixes all actions in a single one by adding more buttons to the response section on the bottom. The user gets the whole picture of the application in a single view, but it can be more confusing.

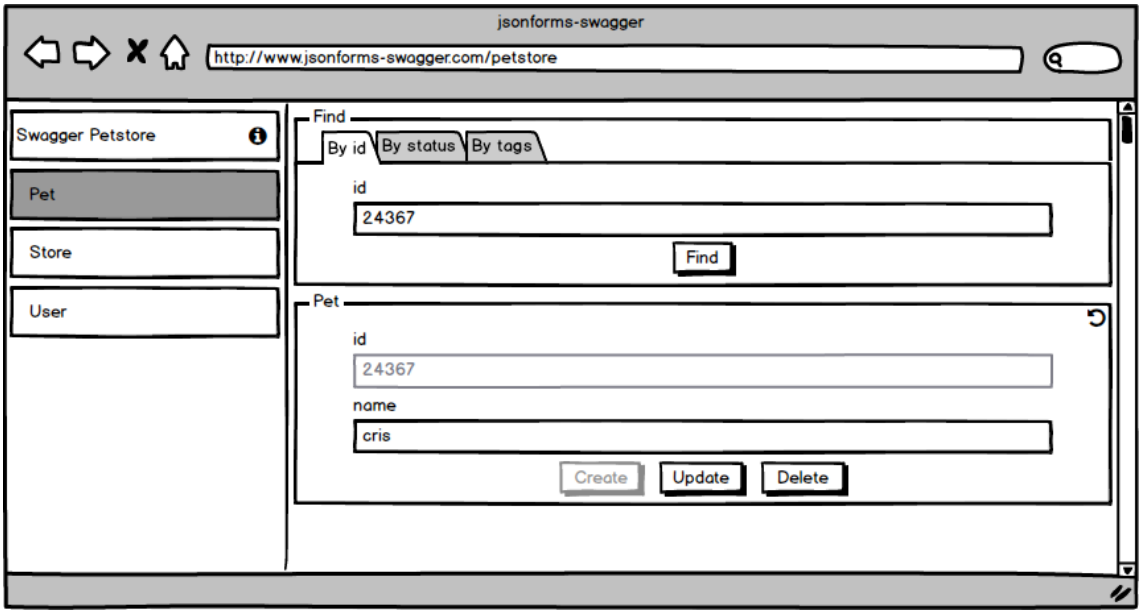

Figure 3: "Single-action" mockup

- "Views" alternative. Instead of using a sidebar, this alternative uses different fullscreen views and offers a breadcrumb to navigate between them. There is more space for showing the forms, but navigation can be harder.

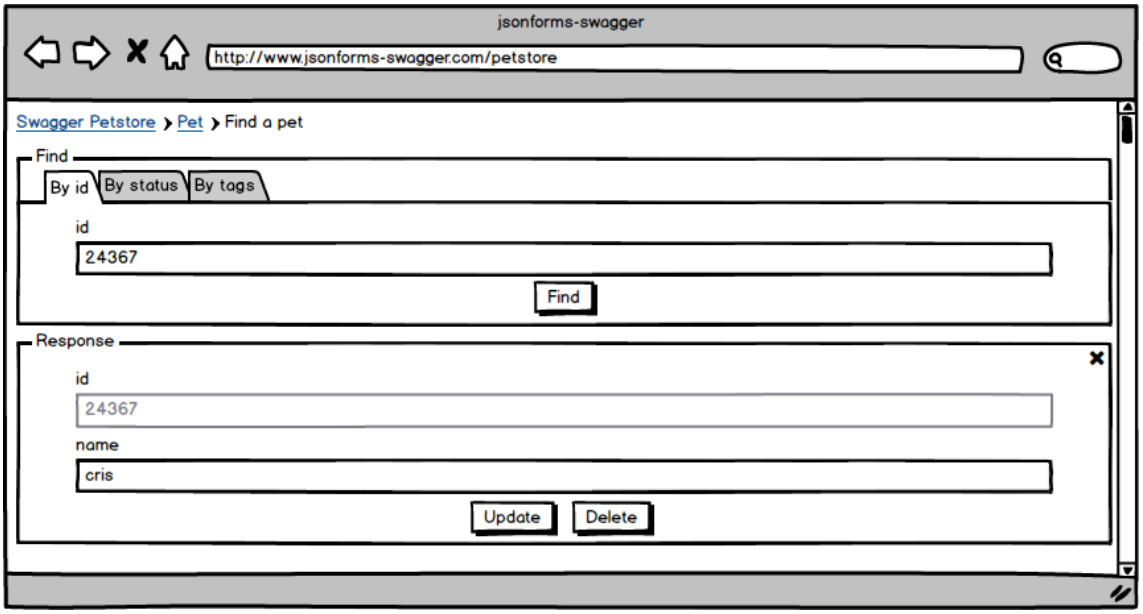

Figure 4: "Views" mockup

After analysing the tests results and the feedback of the different supervisors, we agreed on implementing the "sidebar" alternative, which turned out to be more usable and less confusing than the other two.

5.2. Analysis

The version 2.0 of the OpenAPI Specification can be found at https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md. It describes all the different fields that an OpenAPI definition must or can contain. Jsonforms-swagger uses a minified model of the specification, which only contains the fields which are relevant for the application and the integration with JSON Forms. These fields are:

- API: is the root class, and contains all the needed information about a specific OpenAPI definition: the host and the basePath on which the API is served, its title, a description, etc. It also keeps reference of all the operations listed in a specific OpenAPI definition, and provides methods for searching them by type, path or tag.

- Operation: describes an HTTP operation (GET, PUT, POST, DELETE) on a specific path. It is also composed by the list of parameters needed to perform the operation, and by the possible responses result of its execution.

- Parameter: there exist different types of parameters (path, query, header, body, form), and each of them needs to be sent in a different way.

- APIResponse: describes what and how will respond the server. There can be correct responses (code 200) or error responses (code 400, 500, etc.).

Here is the class diagram representation of the model:

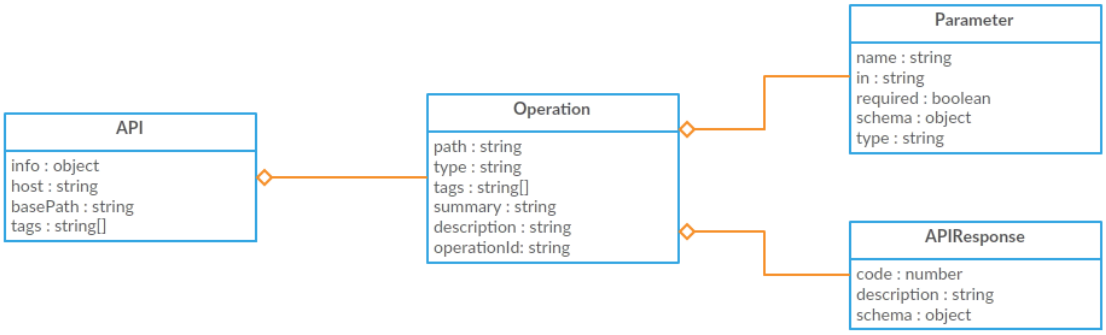

Figure 5: Model class diagram

However, it’s a requirement of jsonforms-swagger to support the customization of the generated UI, allowing to choose which operations to show and how to group them. For this purpose, the previous model needs to be extended with some extra classes:

- EntityType: groups actions which operate over a similar type of entity. For example, an entity type called “Pet” would contain actions to create, find or delete pets.

- Action: groups operations of the same type. For example, an action called “Find pet” could contain a “GET” operation called “getPetById”, and another “GET” operation called “getPetByStatus”.

Here is the class diagram representation of the extended model:

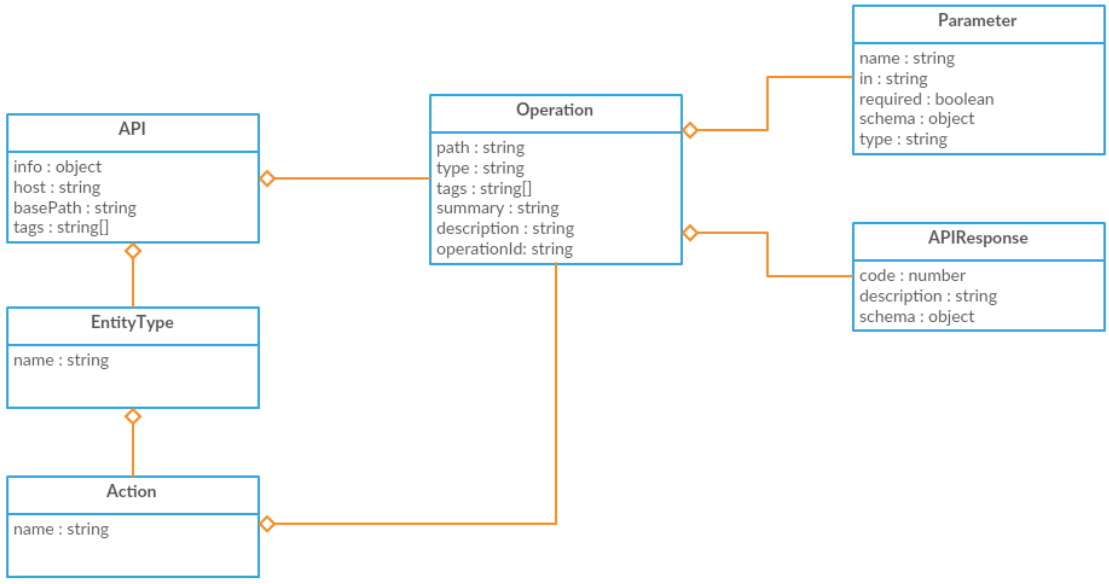

Figure 6: Extended model class diagram

6. Design

6.1. Architecture

The architecture of jsonforms-swagger is mostly inherited from the standard Angular 2 architecture. It follows the conventions for components, services and modules.

Basically, the application is composed of a set of modules, all of them directly related to a section of the UI. Each of those modules is a folder which contains a service (optionally), a component, an HTML template and a css file (optionally), plus any other needed model classes.

Additionally, there is the "core" module, which contains the extra services and connectors that are used by other components.

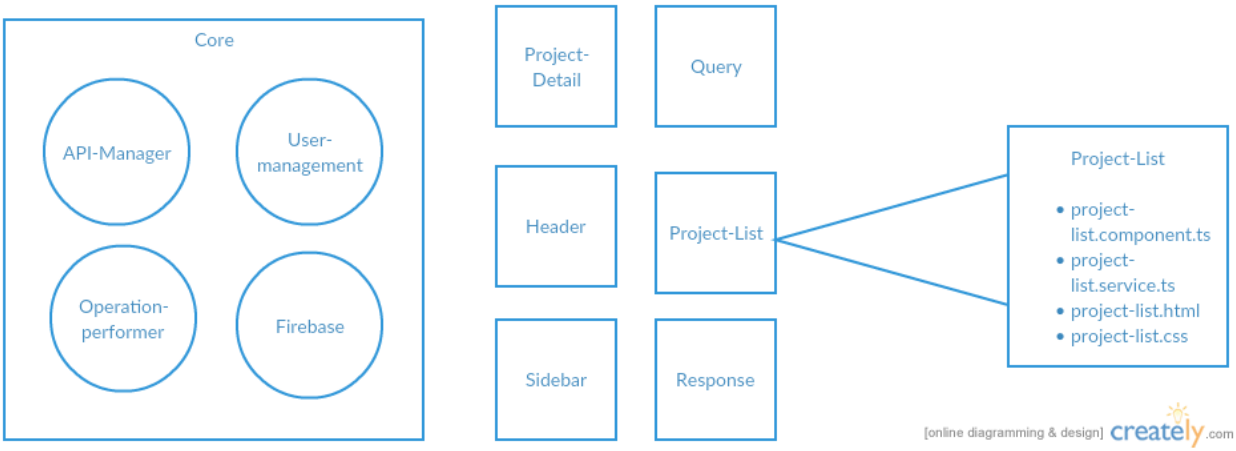

Figure 7: System architecture

This architecture is very effective, as it encapsulates all the functionality of a section of the UI in a single module, and the only public entry of the module is in the service. Also, thanks to the Angular 2 View Encapsulation, we also don't have to worry about polluting the global css namespace. Each css class only affects its associated module.

The communication between modules is achieved via services, with a combination of two design patterns: Dependency Injection and Observables (we will elaborate on this later).

6.2. Technologies

Frontend

We had little room to choose the frontend technology, as JSONForms is only supported on AngularJS. Yet we realised that some of the concepts used by AngularJS were getting outdated, and made the overall development experience more complicated.

At the same time Angular 2 got released, which brought a lot of changes and added many interesting features. This also meant AngularJS would receive less updates and

support in the future, as the team in charge of it was directing its attention towards Angular 2.

Considering all this, we decided that figuring out how to use JSONForms with Angular 2 was worth the effort. For this we used the “upgrade” package, which provided a way to integrate AngularJS and Angular 2 on the same application.

We chose Typescript as the language, as it aligns very well to the vision of making the applications modular and extensible. It’s also the recommended language by Angular 2 developers.

Backend

Initially we didn’t use a custom backend for our application, as it wasn’t a priority. We decided to use Heroku for deployment and a very simple NodeJS server with no custom functionality.

As the project advanced, we realised that a custom backend was needed for the following functionalities:

- Authentication of users
- Storage of projects per user, including project title, url and the extra information for actions and entity types.

For these simple tasks, we chose the Firebase technology. It provides a simple API that covers user Authentication, real-time database and many other features, enough to cover our requirements. It also deploys the app automatically on its own servers, basically removing the need of using Heroku.

6.3. Design patterns

We used a set of design patterns that allowed us to increase the code quality of our application. The most relevant ones are: Dependency Injection and Observables.

Dependency Injection

This pattern allows us to pass services to components, without requiring the components to find or build them. The services are kept in a global store that can be accessed by any component.

To access a service, the component invokes another entity (the injector), which is responsible of finding the service instance in the store and returning it.

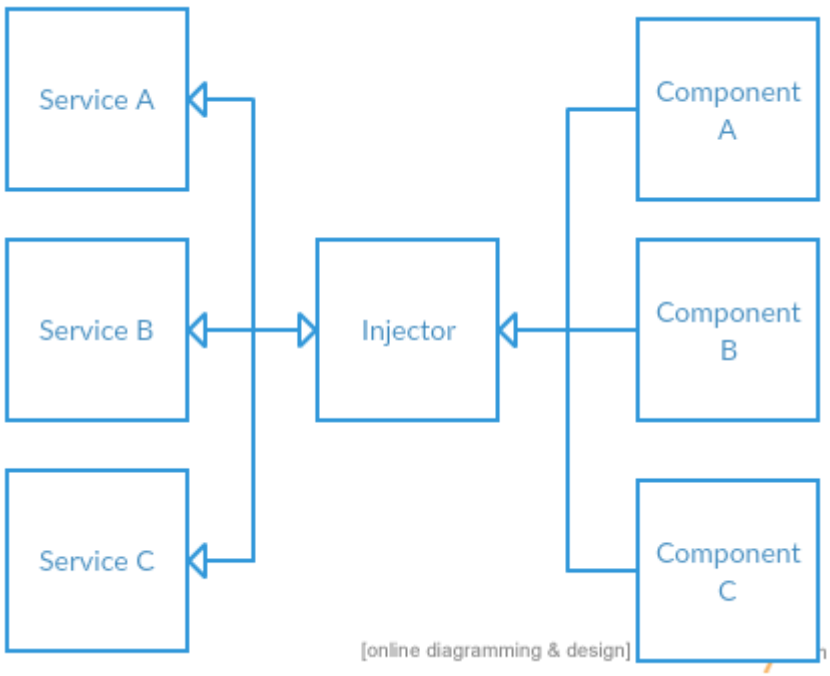

Figure 8: Dependency Injection

The key takeaway of this pattern is that it achieves a decoupling of the main entities of the application (components) and the services used by them. This increases the maintainability and the extensibility of the code.

Observables

We used observables for implementing the communication between services and components.

Basically there is an Observer entity that creates subscriptions to any Observable entity. A subscription is a function that gets invoked when the observed value changes.

This pattern is very useful for reacting to changes on the backend or on different components instantly and reflecting those changes on the UI.

We used the RxJS [16] library.

7. Implementation

The code base of jsonforms-swagger can be found at https://github.com/eclipsesource/jsonforms-swagger. The whole application has been coded by my mate Francisco Rubin Capalbo and me. Here is a link to all my contributions: https://github.com/eclipsesource/jsonforms-swagger/commits?author=hecforga.

As a web application, we had to handle many different implementation topics: UI design, communication with the server, navigation, authorization, validation, etc. However, in this section I'm going to focus my attention in how we implemented the transformation of an OpenAPI definition into the different schemas needed by JSONForms, which is more in line with the purpose of this thesis.

This transformation can be divided in two different stages: from the OpenAPI definition to the application model, and from the application model to the JSONForms schemas.

7.1. From the OpenAPI definition to the application model

If we go back to the "requirements analysis" section, we can recall that jsonforms-swagger uses a model to represent some relevant information from an OpenAPI definition. Here is the class diagram of the model again:

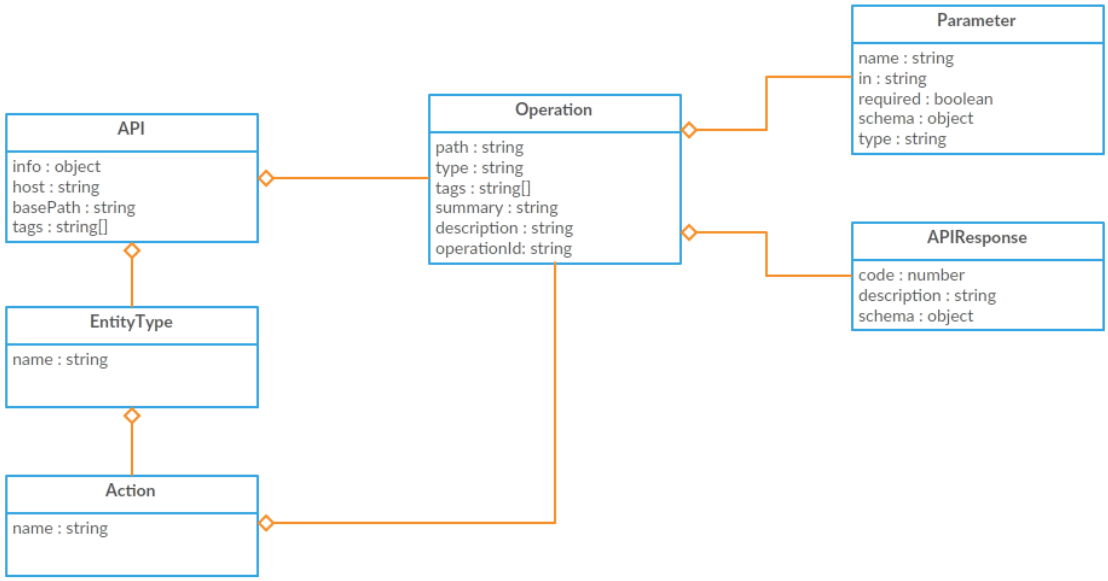

Figure 9: Model class diagram

In order to generate this model, jsonforms-swagger requires a url pointing to the JSON file of an OpenAPI definition. Most of the information of the model can be extracted in a straightforward way from that JSON file, whose structure must follow the version 2.0

of the OpenAPI Specification (https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md). It would normally contain more information than jsonforms-swagger needs, but using a library like “lodash” we can simply pick the desired properties. For example, this could be the pseudo-code for generating an “API” object:

```
let jsonAPI = getJSONFromUrl('http://petstore.swagger.io/v2/swagger.json');

let api = _.pick(jsonAPI, ['info', 'host', 'basePath']);
```

The process to generate “Operation”, “Parameter” and “APIResponse” objects is very similar.

However, we can also recall from the “requirements analysis” section that this model needs to be extended with some extra information to support the customization of the generated UI. The OpenAPI Specification contains no information about “entity types” or “actions”, so the generation process of these objects is more complicated. We are going to reproduce this process with an example.

The following OpenAPI definition describes the API of a pet store server (only the relevant fields for this example are shown):

```
{
  "info": {
    "description": "This is a sample server Petstore server.",
    "title": "Swagger Petstore"
  },
  "host": "petstore.swagger.io",
  "basePath": "/v2",
  "paths": {
    "/pet": {
      "post": {
        "tags": [
          "pet"
        ],
        "summary": "Add a new pet to the store",
        "description": "",
        "operationId": "addPet",
        "parameters": [
          {
            "in": "body",
            "name": "body",
            "required": true,
            "schema": {
              "$ref": "#/definitions/Pet"
            }
          }
```

```
      ],
      "responses": {
        "405": {
          "description": "Invalid input"
        }
      }
    }
  },
  "/pet/{petId}": {
    "get": {
      "tags": [
        "pet"
      ],
      "summary": "Find pet by ID",
      "description": "Returns a single pet",
      "operationId": "getPetById",
      "parameters": [
        {
          "name": "petId",
          "in": "path",
          "required": true,
          "type": "integer",
        }
      ],
      "responses": {
        "200": {
          "description": "successful operation",
          "schema": {
            "$ref": "#/definitions/Pet"
          }
        },
        "400": {
          "description": "Invalid ID supplied"
        },
        "404": {
          "description": "Pet not found"
        }
      }
    }
  }
},
"definitions": {
```

```
    "Pet": {
      "type": "object",
      "required": [
        "name"
      ],
      "properties": {
        "id": {
          "type": "integer"
        },
        "name": {
          "type": "string"
        },
        "status": {
          "type": "string",
          "enum": [
            "available",
            "pending",
            "sold"
          ]
        }
      }
    }
  }
}
```

This schema contains the API information, two operations and a definition. Operations can be identified by type and path. So the two operations of this schema can be identified as “post - /pet” and “get - /pet/{petId}”. A definition describes a data structure that can be referenced and reused from other parts of the schema. In this case, the two operations have a reference to the definition “Pet”. The operation “post - /pet” uses it as a parameter, and “get - /pet/{petId}” uses it as a response.

We need a method to make these computations programmatically:

```
computeDefinitionsUsages(jsonAPI:{}):{} {
    let definitionsUsages:{} = {};

    _.forEach(jsonAPI['paths'], (jsonPath:{}, path:string) => {
        _.forEach(jsonPath, (jsonOperation:{}, operationType:string) => {

            _.forEach(jsonOperation['parameters'], (jsonParameter:{}) => {
                if (jsonParameter['schema'] && jsonParameter['schema']['$ref']) {
                    let definitionRef:string =
jsonParameter['schema']['$ref'].substring(('#/definitions/').length);
                    if (!definitionsUsages[definitionRef]) {
                        definitionsUsages[definitionRef] = {consumes: [], produces:
[]};
                    }
```

```
                    definitionsUsages[definitionRef]['consumes'].push({path: path,
type: operationType});
                }
            });

            _.forEach(jsonOperation['responses'], (jsonResponse:{}) => {
                let definitionRef:string;
                if (jsonResponse['schema']) {
                    if (jsonResponse['schema']['$ref']) {
                        definitionRef = jsonResponse['schema']['$ref'];
                    } else if (jsonResponse['schema']['items'] &&
jsonResponse['schema']['items']['$ref']) {
                        definitionRef = jsonResponse['schema']['items']['$ref'];
                    }
                }
                if (definitionRef) {
                    definitionRef =
definitionRef.substring(('#/definitions/').length);
                    if (!definitionsUsages[definitionRef]) {
                        definitionsUsages[definitionRef] = {consumes: [], produces:
[]};
                    }
                definitionsUsages[definitionRef]['produces'].push({path: path, type:
operationType});
                }
            });

        });
    });

    return definitionsUsages;
}
```

This method, applied to the example schema, returns the following object:

```
{
  "Pet": {
    "consumes": [
      {
        "path": "/pet",
        "type": "post"
      }
    ],
    "produces": [
      {
        "path": "/pet/{petId}",
        "type": "get"
      }
    ]
  }
}
```


From this object (called “definitionUsages”) we can generate the “entity types” and the “actions” and add them to the model (called “api”). This is done in the following method:

```
generateEntityTypesFromDefinitionsUsages(api:API, definitionsUsages:{}) {
    _.forEach(definitionsUsages, (definitionUsage:{}, definitionName:string) => {
        let entityType:EntityType = new EntityType();
        entityType.name = definitionName;

        let findAction:Action = new Action();
        findAction.name = 'Find ' + definitionName;
        _.forEach(definitionUsage['produces'], (operationPathAndType:{}) => {
            if (operationPathAndType['type'] == 'get') {
                let getOperation =
api.getOperationByPathAndType(operationPathAndType['path'],
operationPathAndType['type']);
                findAction.operations.push(getOperation);
            }
        });
        if (findAction.operations.length > 0) {
            entityType.actions.push(findAction);
        }

        let createAction:Action = new Action();
        createAction.name = 'Create ' + definitionName;
        _.forEach(definitionUsage['consumes'], (operationPathAndType:{}) => {
            if (operationPathAndType['type'] == 'post') {
                let postOperation =
api.getOperationByPathAndType(operationPathAndType['path'],
operationPathAndType['type']);
                createAction.operations.push(postOperation);
            }
        });
        if (createAction.operations.length > 0) {
            entityType.actions.push(createAction);
        }

        api.entityTypes.push(entityType);
    });
}
```

This method creates an “EntityType” object for each “definition usage”. Then, it checks if there are any “get” operations in the “produces” array. If so, it creates a “find” action and adds to it all those “get” operations. The same process is applied to the “consumes” array, adding all its “post” operations to a “create” action.

After applying all this process to the example schema, this would be the result:

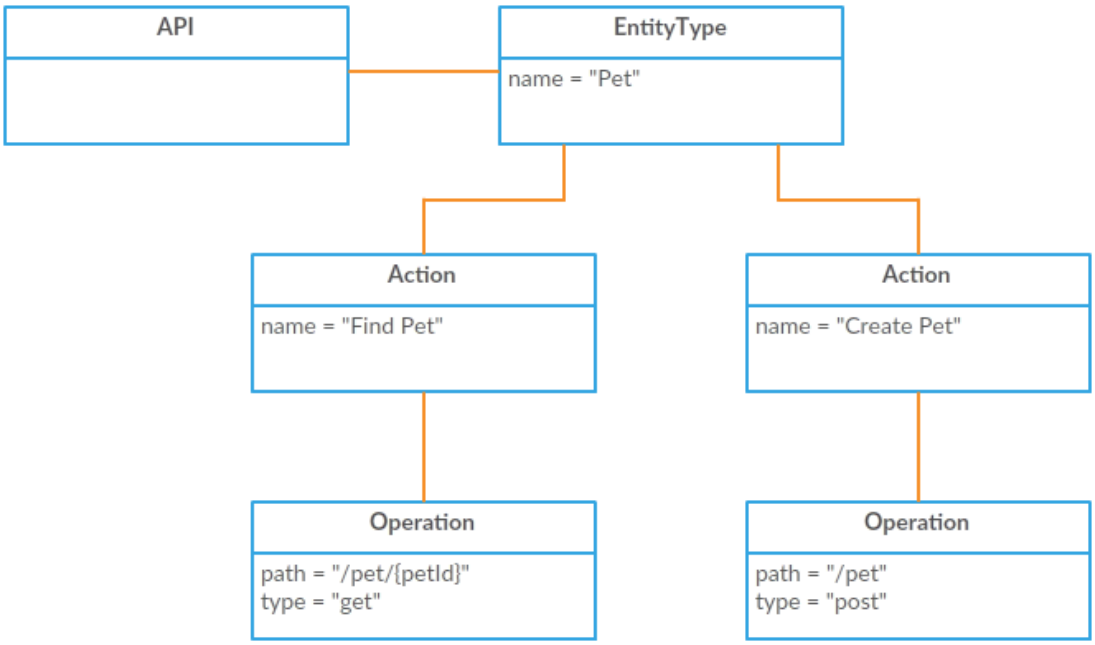

Figure 10: Resulting model

7.2. From the application model to the JSONForms schemas

Until now, we have seen how the model used by jsonforms-swagger is generated from an OpenAPI definition. Now we will focus in how the schemas needed by JSONForms are generated from this model. Jsonforms-swagger uses JSONForms to create the forms of the query and response sections. Basically, the form in the query section allows to input the parameters of the selected operation, while the form in the response section shows the result of the performed operation. We can see these forms in the next mockup:

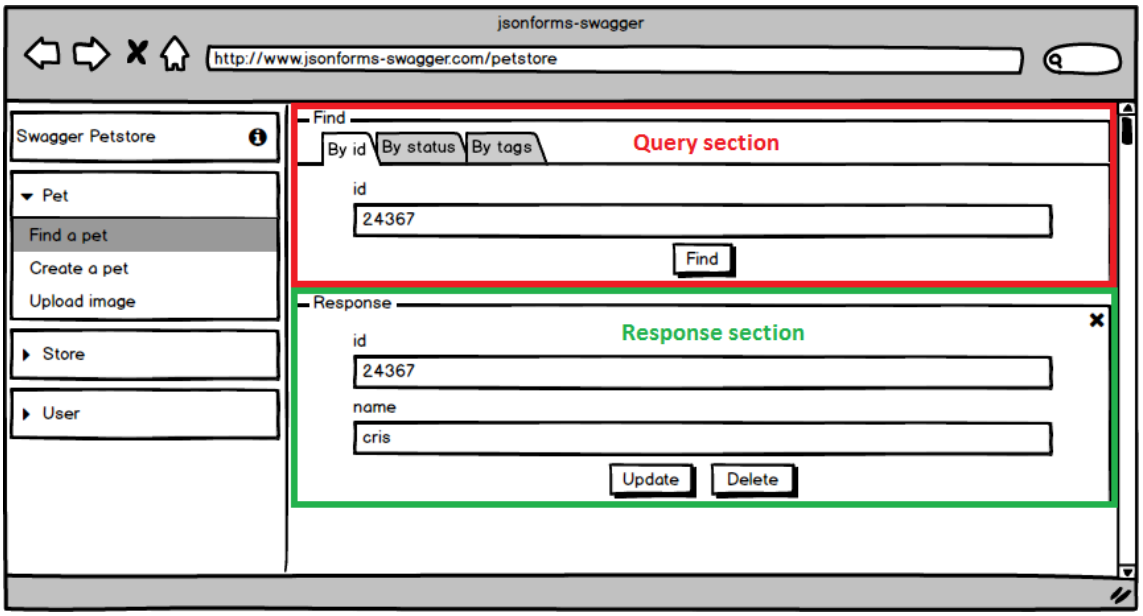

Figure 11: Query and response sections

If we go back to the "Introduction" section, we can recall that JSONForms requires 3 schemas: the Data Schema, which is extracted from the model; the UI Schema, which is generated from the Data Schema, and the actual data provided by the user and/or the server.

Considering the "get - "/pet/{petId}" operation of the previous OpenAPI definition example, we need to extract two Data Schemas: one for the form in the query section, and another for the form in the response section. The model of this operation would look like this:

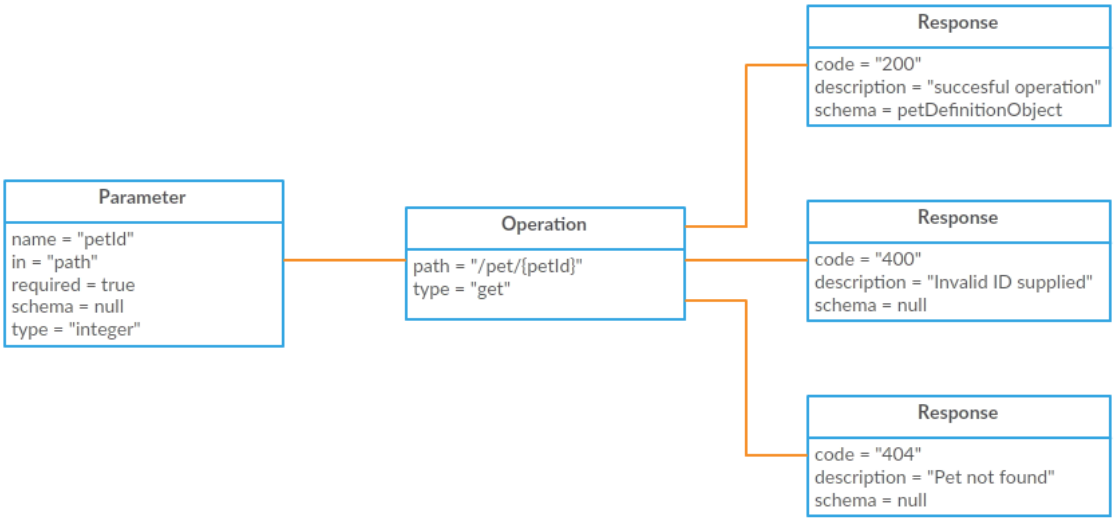

Figure 12: Model of operation "get – /pet/{petId}"

More specifically, the Data Schema for the query section form is generated from the parameters, while the Data Schema for the response section form is extracted from the response with code "200". JSONForms Data Schemas and OpenAPI definitions are all JSON Schemas (the definition of JSON Schema can be found in the "General concepts" section), so the generation process is fairly simple. These would be the resulting Data Schemas of this example:

Query Data Schema	Response Data Schema

<pre>{
"type": "object",
"properties": {
"petId": {
"type": "integer"
}
},
"required": [
"petId"
]
}</pre>
	<pre>{
	"type": "object",
	"properties": {
	"id": {
	"type": "integer"
	},
	"name": {
	"type": "string"
	},
	"status": {
	"type": "string",
	"enum": [
	"available",
	"pending",
	"sold"</pre>

	<pre>{
	  "type": "object",
	  "properties": {
	    "petId": {
	      "type": "integer"
	    }
	  },
	  "required": [
	    "petId"
	  ]
	}</pre>

From these Data Schemas, we can generate the UI Schemas of the forms of the query and the response sections. JSONForms UI Schemas define the layout of a form. Jsonforms-swagger takes the simplest approach for generating these UI Schemas, just including all the form controls in a "VerticalLayout". The resulting UI Schemas would look like this:

Query UI Schema	Response UI Schema

<pre>{
  "type": "object",
  "properties": {
    "petId": {
      "type": "integer"
    }
  },
  "required": [
    "petId"
  ]
}</pre>
	<pre>{
	  "type": "object",
	  "properties": {
	    "id": {
	      "type": "integer"
	    },
	    "name": {
	      "type": "string"
	    },
	    "status": {
	      "type": "string",
	      "enum": [
	        "available",
	        "pending",
	        "sold"
	      ]
	    }
	  },
	  "required": [
	    "name"
	  ]
	}</pre>

After all these transformations, we have finally computed all the needed schemas for JSONForms to create and render the forms of the query and the response sections. To provide a clearer explanation, we have analysed all these processes using a very simple example, but they are also valid for more complex OpenAPI definitions.

8. Acquired competences

In this section I will list the main competences I acquired during the development of jsonforms-swagger.

The area in which I have gained more knowledge is web development. I had very little experience in web development before I started to work in this project. I only knew the basics of HTML and Javascript. Now, I feel that my skills with those programming languages have improved a lot. Especially, I have learnt more about frontend development, and Angular 2 in particular. However, I have also learnt something about other aspects of web development such as: backend development, UI design, system configuration, dependencies management, continuous integration, etc.

As the topic of this thesis suggests, I have also gained knowledge in REST services, the HTTP protocol and client/server architectures in general.

JSONForms and the OpenAPI Specification use JSON structures and JSON Schema, so my skills in modeling data with JSON have improved a lot too.

As a result of the way of work in EclipseSource München, I have become familiar with agile software processes. Jsonforms-swagger was developed in an iterative way, adapting the requirements and planning the next implementation tasks in continuous meetings with my supervisors (every two or three weeks). In these meetings I also learnt how to gather requirements from a client and communicate with my work colleagues, as well as prioritizing and estimating the costs of the different tasks.

Finally, thanks to this project I have been introduced to the Open Source Software (OSS) world, as JSONForms and the OpenAPI Specification are both OSS projects. More specifically, I have learnt how to collaborate in OSS projects using GitHub: submitting issues, opening pull requests, solving conflicts between different branches, etc.

9. Conclusions

After all the development process described in this document, jsonforms-swagger is now in a functional and stable state. It's able to create a web UI which allows a user to communicate with a REST service, given its OpenAPI definition. So, its main objective has been successfully achieved. A demo of the application can be found under this link: https://jsonforms-swagger.firebaseapp.com/.

Besides this main objective, some extra functionalities have been implemented: search/filtering of operations, different methods of authentication, users management system, etc.

The final product covers almost all the requirements analyzed previously. The only use case that still needs to be implemented is "ExportProject". At this moment, jsonforms-swagger doesn't allow a user to download the code of the generated web UI, so that he could include that UI in his web page in a simple way. However, this feature can be added in the future.

Besides this unmet use case, there exist other possible lines of work for the future. For example, extending JSONForms with more custom widgets which could be useful for jsonforms-swagger. The first that comes to my mind is a button able to perform HTTP operations with the data contained in the form. Also, OAuth 2.0 could be added as a possible way of authentication.

Bibliography

[1] M. Castells, The Information Age, Volumes 1-3: Economy, Society and Culture, Malden (Mass.): Wiley-Blackwell, 2000.

[2] C. Babbage, "On the Economy of Machinery and Manufactures," Charles Knight, 1832, p. 114.

[3] "JSONForms," [Online]. Available: http://github.eclipsesource.com/jsonforms/. [Accessed 15 September 2016].

[4] J. Abbate, Inventing the Internet, Cambridge (Mass.): MIT Press, 1999.

[5] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. Thesis, University of California, Irvine, USA, 2000.

[6] L. Richardson and S. Ruby, RESTful Web Services, Sebastopol, CA, USA: O'Reilly Media Inc., 2007.

[7] "Specification | Open API Initiative," [Online]. Available: https://openapis.org/specification. [Accessed 15 September 2016].

[8] "One framework. - Angular 2," [Online]. Available: https://angular.io/. [Accessed 15 September 2016].

[9] "TypeScript - JavaScript that scales.," [Online]. Available: https://www.typescriptlang.org/. [Accessed 15 September 2016].

[10] "JSON Schema and Hyper-Schema," [Online]. Available: http://json-schema.org/. [Accessed 15 September 2016].

[11] "RAML," [Online]. Available: http://raml.org/. [Accessed 15 September 2016].

[12] "API Blueprint," [Online]. Available: https://apiblueprint.org/. [Accessed 15 September 2016].

[13] "EMF Forms," [Online]. Available: http://www.eclipse.org/ecp/emfforms/index.html. [Accessed 15 September 2016].

[14] "joshfire/jsonform," [Online]. Available: https://github.com/joshfire/jsonform. [Accessed 15 September 2016].

[15] "Swagger UI - Swagger," [Online]. Available: http://swagger.io/swagger-ui/. [Accessed 15 September 2016].

[16] "RxJS API Document," [Online]. Available: http://reactivex.io/rxjs/. [Accessed 15 September 2016].