

Jukka Korva

## **Developing a web application with Angular 2**

Graphical editor for Happywise's Cove Trainer

## **Developing a web application with Angular 2**

Graphical editor for Happywise's Cove Trainer

Jukka Korva  
Thesis  
Autumn 2016  
Business Information Technology  
Oulu University of Applied Sciences

## ABSTRACT

Oulu University of Applied Sciences  
Business Information Technology

---

Author(s): Jukka Korva

Title of Bachelor's thesis: Developing a web application with Angular 2

Supervisor(s): Jouni Juntunen

Term and year of completion: Autumn 2016

Number of pages: 35

---

The purpose of this thesis is to develop a web application using Angular 2 for Happywise. Happywise has a cloud based training tool for organizations which purpose is to train employees in exceptional situations. These situations can for example be abnormal operating conditions in factories and power plants.

These conditions are simulated in the training tool using scripts, which are partly JSON based. By nature, JSON is not very human friendly, and the creation of these files is tedious. A graphical interface for editing the JSON files would substantially decrease the time needed to create a scenario. It would also remove the possibility of human error regarding errors in JSON syntax.

The theoretical background revolves around Angular 2. Its main features are introduced and discussed. The main source for the theoretical background is Angular 2's own documentation. In the implementation part, a general idea on how an Angular 2 application can be created based on the given theoretical information is given.

The main result of the work done is a working editor application which meets its development targets. Happywise has also stated that they will be using the editor in a future project and will continue developing it. A secondary result of this thesis is a rough guide on the main features of Angular 2 and how an application can be developed with the framework.

---

Keywords: Web application, Angular 2, framework

# CONTENTS

1	INTRODUCTION .....	5
2	THE ANGULAR 2 FRAMEWORK.....	7
2.1	Basic information.....	7
2.2	Basic building blocks .....	8
2.3	Directives and data binding .....	9
2.4	Dependency injection .....	11
2.5	Decorators and lifecycle hooks.....	11
2.6	HTTP library .....	12
3	IMPLEMENTATION PROCESS .....	14
3.1	Setting up the development environment .....	14
3.2	Scenario parsing functionality.....	16
3.3	User interface .....	19
3.4	Editing functionality .....	24
3.5	Communicating with the server .....	29
4	CONCLUSIONS .....	32
5	DISCUSSION .....	33
	REFERENCES .....	35

# 1 INTRODUCTION

Frameworks are used extensively in websites of all sizes nowadays. They can arguably be said to be essential in building a more complicated site, such as a single page application, as HTML was not originally meant to be used for such purposes. The task of writing code to handle just the basic navigation features of a single page application can be bothersome and a waste of good development time. Fortunately, there are many capable JavaScript frameworks in existence that remove the need to write that kind of boilerplate code. One of the more notable ones is AngularJS, which has transitioned to a new version.

The purpose of the thesis is to create a graphical scenario editor for Happywise's Cove Trainer exceptional situation training tool using Angular 2. Cove Trainer is intended for companies to train new employees in exceptional situations which need specific actions. An example training situation could involve a power plant boiler and its malfunction. The system can simulate this and employees would then train to correctly respond to the situation. The editor would be capable of reading scenarios stored as JavaScript Object Notation (JSON), a textual object-oriented data storage format. Happywise is an IT company offering cloud based products as services as well as IT solutions tailored per the customer's wishes.

Currently, the scenarios are created and edited by hand. As the system and the scenarios are quite complex, it is a cumbersome task. It is not made any easier by the fact that JSON is not very easy to read even when formatted in a way to maximize readability. Mistakes are easy to make even when the text editor used can check for incorrect JSON syntax. Given these limitations, a graphical editor would greatly simplify scenario creation and editing by removing the most time-consuming step in the process.

The primary main goal of the thesis is to develop an editor that can read the JSON scenario data, displaying the read data correctly and being able to edit and create new scenarios. The secondary main goal is to design the editor to resemble the actual application output as closely as possible. This is to ensure that what the scenario designer sees is what the user sees. The editor system should also allow one scenario be edited by multiple users, but it is not a main concern for this thesis however.

Considering the project's goals, the development tasks are straightforward. The editor needs a user interface, the actual editor logic which makes everything possible and the server logic which handles the scenario data saving and loading. Happywise has said that the editor should be web based, so the user interface will be built with HTML and CSS. It will make it easier to re-create a similar user interface as the actual application, as Cove Trainer itself is web based. Bootstrap CSS will be used when creating the user interface. This will allow easier prototyping and consistent user experience between different browsers.

As the editor will be created using Angular 2, the development language will be TypeScript. Angular 2 was chosen because it was new territory, and therefore a great opportunity to learn to use it. The previous version of the framework became very popular and made developing web applications easier. Google has claimed the new version will be even better. Another reason for choosing Angular 2 was its DOM manipulation features, which will simplify the interactions between the code and user interface, which are plentiful and potentially complicated considering the application's intended use. Angular 2 is also modular which should code reuse easier.

The server side logic of the editor will be written with PHP. The use of a framework in this case is not strictly necessary, but for convenience's sake Slim will be used. The Slim framework has powerful REST features and can easily work with JSON. The file operations on the other hand will be done with PHP's built-in functionality, as they are not expected to be complicated.

Development work will focus on the editor itself, which means that third-party libraries will be used for features that would otherwise need a more extensive amount of coding. Cases where this might apply are dialog windows and the user interface itself. Bootstrap CSS was mentioned as a useful framework which covers both use cases. The thesis will focus on Angular 2 and its use with the development of the editor. Its main features will be introduced and the implementation process will focus on giving a general idea on how an Angular 2 application can be developed.

## 2 THE ANGULAR 2 FRAMEWORK

Frameworks are intended to simplify the development of an application. They generally implement common features which are integral to an application, like authentication, security and user interface components. When a framework is used properly, the time taken to develop a web site or web application can be reduced, because it is not necessary to re-implement the features a framework offers for every new project. (Srinivasan 2014, chapter 15.5, cited 1.12.2016.)

### 2.1 Basic information

Angular 2 is a framework for building web applications in HTML and a language that compiles into JavaScript such as TypeScript, although a framework version for plain JavaScript also exists. Applications created with Angular 2 are separated into HTML templates, component classes written to manage the templates and services that contain the business logic of the application. (Architecture overview 2016, cited 5.9.2016.) This modular approach, also illustrated in figure 1, enables rapid and logical development of applications and eases their testing. In a properly designed Angular 2 app, the different parts of the application should not be aware of the implementation details of other parts. Should the development guidelines of the framework be followed, it will be noticeably easier to follow this paradigm.

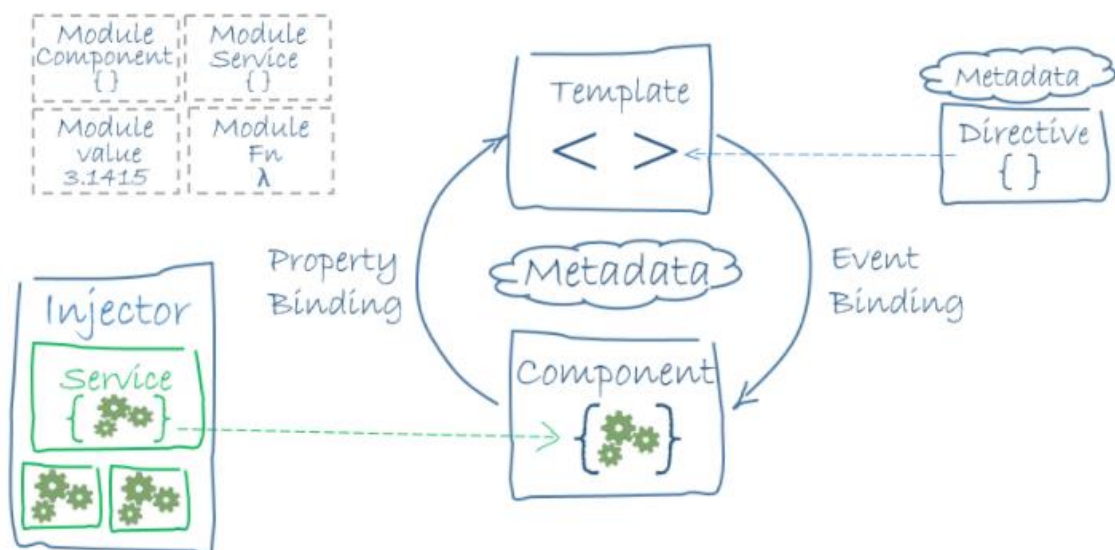


FIGURE 1. Angular 2 application architectural overview (Architecture overview, cited 15.11.2016)

The preferred language for Angular 2 is TypeScript. It is a superset of JavaScript developed by Microsoft to ease the development of large JavaScript projects. One of its features include an optional type system. (TypeScript Language Specification 2016, cited 12.9.2016.) The type system brings the language closer to other typed languages like Java or C# and it is perhaps the biggest advantage TypeScript can offer compared to JavaScript. The ability to define types helps in maintaining the structure of a large codebase and reduces ambiguity of what a function expects as input and what should be expected as output. Code written in TypeScript is transformed into JavaScript that resembles the source as closely as possible to make debugging easier. The compilation is done on every file save to match the usual development cycle of JavaScript. (TypeScript Language Specification 2016, cited 12.9.2016.) In this sense, the speed of JavaScript development is kept.

## **2.2 Basic building blocks**

The most important building block in an Angular 2 app is the module, which are used to group together related component classes and services (Angular modules 2016, cited 5.9.2016). In this sense, they could loosely be compared to namespaces in C# applications, which have the same functionality of grouping together related code elements. Angular modules are still distinctly different from namespaces as they can be divided into two types. Every Angular 2 app has a root module, which is used to launch the app and feature modules which extend the capabilities of the root module (Angular modules 2016, cited 5.9.2016). Namespaces on the other hand cannot be differentiated like this as their distinction comes from the contents of the namespace.

Component classes interact with templates and services, which contain the app business logic. They are bundled together with related services in modules. They have properties and event handlers that the framework binds to the view declared in the component. (Architecture overview 2016, cited 10.9.2016.) The automatic binding done by the framework ensures that little to no time must be wasted writing code to make user interaction possible. Another advantage of this is that services only need to concern themselves with returning output the component expects.

Services are another building block which are needed to build an app with Angular 2. Their function is to provide the application with specific functionality which would not fit into a component class, such as logic to retrieve data from a server or logic to log messages to the browser console.



(Architecture overview 2016, cited 12.9.2016.) Services reinforce Angular's ideology about modularity. Components are not intended to do anything else but enable the interactions between the user and the application logic. While nothing stops the developer from implementing the application without services, it is not recommended.

Component HTML templates tell Angular how to render a specific component's view. In addition to normal HTML markup, a template has Angular template syntax, which the framework parses when the view is rendered. (Architecture overview 2016, cited 10.9.2016.) This compartmentalization means that the user interface of an Angular 2 app can be made up of several different HTML files in addition to the 'index.html' file. It also means that reuse of user interface components is very simple as all that is needed is a single HTML tag which identifies the template and component to insert.

### 2.3 Directives and data binding

The Angular 2 template syntax includes support for directives. These are referred to as structural or attribute directives and their purpose is to transform the HTML document based on user input or supplied code expression. Structural directives for example allow to easily repeat a collection of objects into the document or to hide or show an element based on the value of a component property. (Architecture overview 2016, cited 14.11.2016.) Structural directives make developing the user interface easier and more expressive, when it is obvious where a list of elements for example will be when the application is running and a certain property becomes true. Figure 2 is an example of two structural directives. Attribute directives on the other hand look like normal HTML attributes and only affect the element they are found in. The most notable example of an attribute directive is 'ngModel', which is used for data binding. (Architecture overview 2016, cited 14.11.2016.)

```
<div class="editor-images-overlay" *ngIf="ShowImagesOverlay">
  <div id="editor-images-overlay-entries">
    <div class="editor-images-overlay-entry" *ngFor="let image of EditorData.ScenarioImages.Elements">
```

FIGURE 2. Angular 2 template syntax

Data binding is a central feature of Angular 2 and it is made to be convenient and easy to use. The data binding is done through Angular template syntax. Square brackets mean binding from the component to the document and parentheses mean binding from the document to the component.

The bound properties are updated every time a change is detected in the HTML document. (Architecture overview 2016, cited 14.11.2016.) This means that there is no need to create logic for writing to, and reading values from, the HTML document. The amount of boilerplate code is reduced and the application suffers less potential bugs as well. There is also a third method of data binding, called interpolation. This is identified in Angular template syntax as double curly brackets. (Architecture overview 2016, cited 14.11.2016.) Interpolation is useful for outputting component properties to the document and it is arguably more descriptive than using square brackets.

One feature of the Angular 2 framework ties in to data binding. This feature is a data transformation capability which can be applied to interpolated expressions and directive expressions in a component template. The framework offers built-in pipes for common scenarios, such as dates or changing text to uppercase. It is also possible to create custom pipes by using the '@Pipe' decorator, as illustrated in figure 3. The pipes are used by appending a vertical bar to an expression followed by the wanted transformation. (Pipes 2016, cited 16.11.2016.) This feature makes it very easy to transform data retrieved from the server. The date transform for example removes the need to manually transform a date in its database representation prior to displaying it to the user. The behavior can also be guaranteed to be constant.

```
@Pipe({ name: 'map' })
export class KeysPipe implements PipeTransform {
  /**
   * Interface function of PipeTransform. Transforms input to desired output.
   *
   * @param {Map<string, Variable>} value
   * @returns
   *
   * @memberOf KeysPipe
   */
  transform(value: Map<string, Variable>) {
    let keys: any = [];
    value.forEach((value, index: string) => {
      keys.push({key: index, value: value});
    });
    return keys;
  }
}
```

FIGURE 3. Custom pipe transform example

## 2.4 Dependency injection

Services are inserted into components when they are created through a mechanism called dependency injection. It is a central feature in Angular and designed to make using the component and service pattern as convenient as possible. The services a component needs are declared in its constructor function. (Architecture overview 2016, cited 12.9.2016.) This means that only a glance at the constructor of a component is needed to know which services it needs to function properly. Dependency injection makes testing and development easier, because the component does not care about what the service truly does. This allows the usage of a placeholder function which can return predefined data for the component to process.

Dependency injection also means that the services the Angular framework creates are singletons, meaning that there is always only one instance of them available. This pattern can be broken by using the 'providers' property available in the decorator that declares that a class is a component. If the 'providers' property is found, the framework creates a new instance of the services found in the property. (Architecture overview 2016, cited 15.11.2016.) This is an important feature to take into consideration, as it has implications on how an application should be developed. Components by nature are isolated from each other, and sometimes they must be able to transfer information between each other. Singleton services are one way of achieving this. Injecting the same service instance to two different components means they both have access to the same data.

## 2.5 Decorators and lifecycle hooks

Before Angular can use a component, the framework must be told what to do with it. This is done through a TypeScript feature known as decorators. Angular 2 uses decorators for different purposes and the main one is to specify that a class is a component class. The '@Component' decorator tells Angular which tag the component represents and which template to use among other configuration information. (Architecture overview 2016, cited 10.9.2016.) Separating this configuration from the actual class definition makes reading the code easier and simplifies the implementation process. Like components, services have their own decorator. If a service depends on other services, it must be decorated with '@Injectable'. If the aforementioned is true and the decorator is missing, the framework will throw an error. It is still recommended to apply the decorator to every service that the app uses, even those that do not have dependencies to other

services. (Dependency injection 2016, cited 7.10.2016.) This removes the possibility of the error occurring, as it can potentially be annoying to troubleshoot, especially when the application grows.

The Angular 2 framework also includes hooks into its internal processes. These are so called life-cycle hooks, which can be used to initiate actions during specific points of the framework's execution cycle. These are exposed to the framework's consumer as TypeScript interfaces, and using them is as simple as importing the wanted life-cycle hooks and specifying that a component class uses those interfaces. (Lifecycle hooks 2016, cited 16.11.2016.) The possibility to tap into these events give more power to developers to control the application how they want. Another reason might be to initialize certain data only after a specific point in the framework's execution cycle has passed.

## **2.6 HTTP library**

The Angular 2 HTTP library is built on JavaScript's built-in 'XMLHttpRequest'. The library is included in its own module, which must be imported before the 'Http' service can be used in any component or service. The service itself supports all general HTTP requests, with the most common ones being get and post. (HTTP client 2016, cited 16.11.2016.) The point of building the library on the 'XMLHttpRequest' feature is the ability to make asynchronous requests, which will not lock the application while they are in flight. This results in a more pleasant user experience, when the application will not freeze while it retrieves data from the server, or sends data to it.

Another part of the HTTP library is RXJS observables. RXJS is a 3<sup>rd</sup> party library which the Angular 2 framework makes extensive use of. The observables are used in other parts of Angular 2 in addition to the HTTP library. An observable can be likened to an event handler. For example, the HTTP library request functions all return an observable, which can be subscribed to. This subscription is usually done in component classes, which are generally where the HTTP requests are initiated. Subscribing to an observable is basically the same as attaching a handler function for an event. When a subscription is attached to an HTTP observable, the request is sent. To avoid this behavior, but still attach a handler to the observable, 'map' can be used. (HTTP client 2016, cited 16.11.2016.) This observable chaining can potentially yield great benefit to an application that communicates with a server. Extraneous code in the function that starts the HTTP request can be

avoided by using the 'map' functionality of the observables to do the required work in the service function itself. A simple example of observable chaining is illustrated in figure 4.

```
public OnScenarioSaveClick(): void {
  this.scenarioListData.SaveScenario().subscribe(() => {
    window.alert("Käsikirjoitus tallennettu!");
  }, error => {
    window.alert("Käsikirjoituksen tallennus epäonnistui! Selaimen konsolissa lisätietoja.");
  });
}

public SaveScenario(): Observable<any> {
  return this.scenarioData.SaveScenario()
    .map(response => { return response })
    .catch((error) => { return Observable.throw(error) });
}
```

FIGURE 4. Example of Angular 2's usage of Observable

## 3 IMPLEMENTATION PROCESS

### 3.1 Setting up the development environment

The development process was started with the installation of the Angular 2 environment. The quick start page of the Angular 2 documentation included the necessary information for setting up an Angular 2 application. Node.js and npm were mentioned as requirements for development with the framework, so they had to be installed first. The reason for this is that the recommended way to install Angular 2 is through npm. (2016, cited 1.11.2016.) The easiest way to do this was to use a package manager. In this case, Chocolatey for Windows was installed and used. Alternatively, a separate installer could have been obtained from the Node.js homepage.

As can be seen from figure 5, after executing 'choco install node' in a command prompt the installation process was very straightforward. Npm did not need a separate installation as it is included with Node.js. It was time to create a folder for the editor project and copy configuration information from the Angular 2 quick start page after the prerequisites had been met. It should however be mentioned that at this point there was still no Angular 2 framework anywhere in the project folder. The framework files and their dependencies were obtained through npm, which is the Node.js package manager. One of the configuration files copied from the quick start page was a list of packages for npm to acquire. This list included the Angular 2 framework itself and other required libraries.

```
Administrator: Komentokehote
By installing you accept licenses for the packages.

nodejs.install v6.5.0 [Approved]
The package nodejs.install wants to run 'chocolateyInstall.ps1'.
Note: If you don't run this script, the installation will fail.
Note: To confirm automatically next time, use '-y' or consider setting
'allowGlobalConfirmation'. Run 'choco feature -h' for more details.
Do you want to run the script?([Y]es/[N]o/[P]rint): y

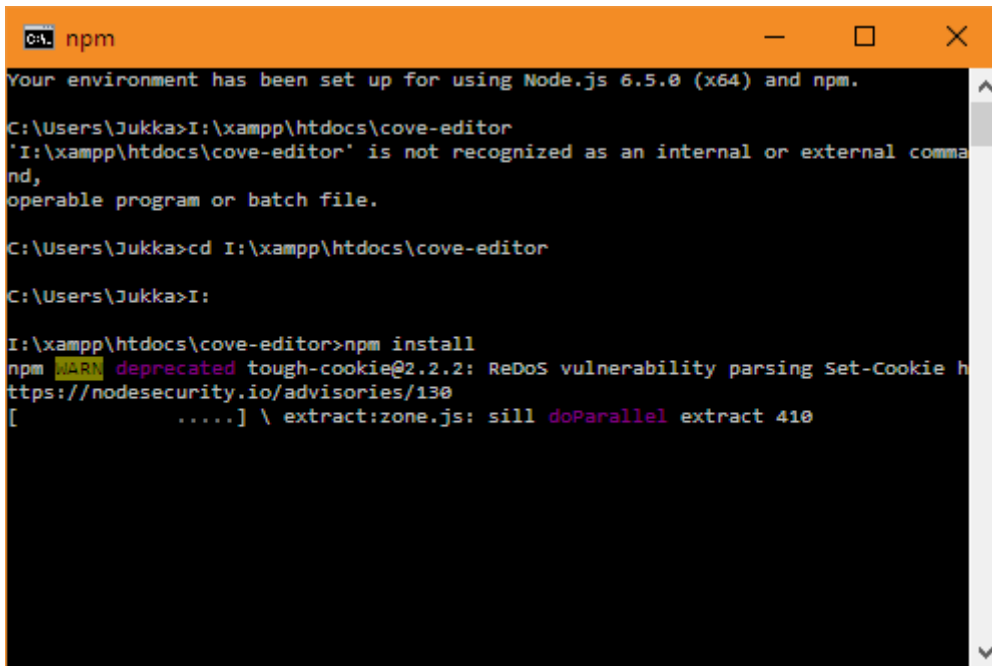
Downloading nodejs.install 64 bit
  From 'https://nodejs.org/dist/v6.5.0/node-v6.5.0-x64.msi'
Progress: 100% - Completed download of C:\Users\Jukka\AppData\Local\Temp\chocola
tey\nodejs.install\6.5.0\node-v6.5.0-x64.msi (12.02 MB).
Download of node-v6.5.0-x64.msi (12.02 MB) completed.
Installing nodejs.install...
nodejs.install has been installed.
Environment Vars (like PATH) have changed. Close/reopen your shell to
see the changes (or in powershell/cmd.exe just type `refreshenv`).
The install of nodejs.install was successful.
  Software installed as 'msi', install location is likely default.

Chocolatey installed 1/1 packages. 0 packages failed.
See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).

C:\WINDOWS\system32>
```

FIGURE 5. Installing Node.js

The installation was once more simple with only one step, which was to execute 'npm install' in the editor's project folder. This installation procedure in figure 6 resulted in a folder called 'node\_modules' inside the project folder which contained all packages defined in the 'packages.json' file. At this point the setup was almost complete. The remaining steps were to create the application's root module and component and copy the 'index.html' file contents from the quick start page so the application could be loaded in a browser. After this was done, the TypeScript compiler and the web server used to serve the application during development were started by executing 'npm start' in the project folder.



```
npm
Your environment has been set up for using Node.js 6.5.0 (x64) and npm.

C:\Users\Jukka>I:\xampp\htdocs\cove-editor
'I:\xampp\htdocs\cove-editor' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Jukka>cd I:\xampp\htdocs\cove-editor

C:\Users\Jukka>I:

I:\xampp\htdocs\cove-editor>npm install
npm WARN deprecated tough-cookie@2.2.2: ReDoS vulnerability parsing Set-Cookie header
https://nodesecurity.io/advisories/130
[.....] \ extract:zone.js: sill doParallel extract 410
```

FIGURE 6. Npm installing packages

### 3.2 Scenario parsing functionality

It was decided that the best way to begin the actual development was to implement JSON file parsing. With a working implementation, it would be easier to design the other parts that were dependent on it. The first step in this was to look at the JSON file itself. It was necessary to know what kind of structure the file had, so that similar data structures could be created for the editor.

The data structures were created as classes like in figure 7, as TypeScript can use them like other typed languages. The 'Scenario' class is akin to the whole JSON file and the properties listed in it are the different parts of the file. This way it was easy to create the functionality that would parse the scenario file.



```

export class Scenario {
  alarms: Alarm[];
  events: Event[];
  scene: Scene;
  /**
   * Contains a name indexed map of the scenario's variables.
   *
   * @type {Map<string, Variable>}
   * @memberOf Scenario
   */
  variables: Map<string, Variable>;
  /**
   * INTERNAL -- Variable list in array form to make certain operations easier. This should not be
   * encoded to JSON.
   *
   * @type {Variable[]}
   * @memberOf Scenario
   */
  variableList: Variable[];

  constructor() {
    this.alarms = new Array<Alarm>();
    this.events = new Array<Event>();
    this.scene = new Scene();
    this.variables = new Map<string, Variable>();
    this.variableList = new Array<Variable>();
  }
}

```

FIGURE 7. Scenario class definition

In line with Angular philosophy, the parsing functionality was implemented as a service. Figure 8 depicts a function inside 'JSONReaderService' service which is called every time JSON data is received. At this point a demo scenario was loaded from a JSON file in the editor's project folder, because there was no server functionality implemented yet. The actual scenario parsing was implemented in their own classes, with one parser class for each property defined in the 'Scenario' class. This was done to make the code more readable and easier to maintain, as there would be no need to look through a monolithic block of code for a single line.

```

private ParseJsonResponse(json: any, requestType: LoadRequestType): Scenario | string[] {
  if (requestType == LoadRequestType.ScenarioFile) {
    var parsed: any = { scenario: Scenario, images: ScenarioImages };
    parsed.scenario = new Scenario();
    parsed.images = new ScenarioImages(json.images);
    console.log(json);
    if (json == null || json == undefined) {
      console.log("Server returned null!");
      return parsed;
    }
    parsed.scenario.alarms = AlarmsDecoder.Decode(json.scenario.alarms);
    parsed.scenario.scene = SceneDecoder.Decode(json.scenario.scene);
    parsed.scenario.events = EventsDecoder.Decode(json.scenario.events);
    parsed.scenario.variables = VariablesDecoder.Decode(json.scenario.variables);
    console.log(parsed);
    return parsed;
  } else {
    return json;
  }
}

```

FIGURE 8. JSON parsing function

Figure 9 is an example of how the parsing was implemented. This decoder is for the 'scene' property in the 'Scenario' class and it contains the interface elements which form the visible graphical user interface of the scenario in Cove Trainer. The decoders for the other properties in the scenario file were created following the same principle; one public and static function which was used to start the parsing, with the parsing performed in private and static functions. This function signature was chosen because it removed the need to instantiate the decoders every time they were needed.

```
public static Decode(sceneData: any): Scene {
  var scene = new Scene();
  if (sceneData.length == 0)
    return scene;

  scene.Description = sceneData.description;
  scene.Name = sceneData.name;

  scene.OperatorDisplays = new Array<OperatorDisplay>();
  for (let i = 0; i < sceneData.operator_displays.length; i++) {
    var current = sceneData.operator_displays[i];
    var operatorDisplay = new OperatorDisplay();
    operatorDisplay.BackgroundImage = current.backgroundImage;
    operatorDisplay.Name = current.name;
    // Check if the scenario contains the item lists and parse them, or just instantiate a new array of list type if it doesn't
    // operatorDisplay.ElementItems = !_has(current, 'elements_name') ? this.PopulateElementsArray(current.elements_name) : new Array<ElementItems>();
    operatorDisplay.BarGaugeItems = !_has(current, 'bar_gauges') ? this.PopulateBarGauges(current.bar_gauges) : new Array<BarGaugeItem>();
    operatorDisplay.SwitchItems = !_has(current, 'switches') ? this.PopulateSwitchItems(current.switches) : new Array<SwitchItem>();
    operatorDisplay.SelectionItems = !_has(current, 'selections') ? this.PopulateSelectionItems(current.selections) : new Array<SelectionItem>();
    operatorDisplay.TextItems = !_has(current, 'texts') ? this.PopulateTextItems(current.texts) : new Array<TextItem>();
    scene.OperatorDisplays.push(operatorDisplay);
  }
  return scene;
}
```

FIGURE 9. Scene parser

While not strictly related to the parsing functionality itself, the service class for the scenario editor was created at this point. The reasoning was that there was not a lot of code written yet, removing the need to refactor the code later down the line. It would also help to solidify the structure of the code. The service's purpose was to provide the scenario editor component with data loading, saving and editing functionality.

With the first implementation of the decoding functionality done, it was time to see if it worked. This was done by importing the 'JSONReaderService' in the scenario editor service class and referring to it in its constructor, like in figure 10. When starting the application, Angular 2 looks at the constructors of component classes and services, and inserts references to the services it finds in them. Performing the load test was as simple as calling the placeholder JSON load function, which tried to parse the demo scenario. On success, it would output what it parsed into the browser developer console. If an error occurred, the whole application would stop as bugs in the scenario parsing could be considered show-stopping errors.

```
import { JSONReaderService, LoadRequestType } from '../io/json-reader.service';
constructor(
  /** Provides JSON download utilities for the app. */
  private JSONReader: JSONReaderService,
```

FIGURE 10. Service import and reference in service class

When it was clear that the scenario parsing worked as expected, the editor still needed a way to encode its data structures back to JSON. This was achieved simply by re-implementing the decoders, but in reverse order. Instead of assigning values from JSON to class properties, the encoders assign class properties to a simple JavaScript object. Another service, 'JSONWriterService' was created for this purpose.

Because there was not any server functionality at this point, some other method was needed to test the encoding functionality. After some googling, one possible solution was found on Stack Overflow. The idea was to create a binary large object from the JSON string and open it in a new browser tab, which is illustrated in figure 11. (2014, cited 12.11.2016.) From the tab, it could be copy-pasted into a source code editor and formatted for an easier inspection experience. A successfully implemented decoder and encoder meant that the scenario could be decoded and encoded without a problem, with the original scenario file contents then replaced with the test's output and the test repeated successfully.

```
public OutputTestJson(scenario: Scenario): void {
  var jsonScenario: any = this.ParseJson(scenario);
  var file = new Blob([JSON.stringify(jsonScenario)], { type: 'json' });
  window.open(URL.createObjectURL(file));
}
```

FIGURE 11. JSON output test function

### 3.3 User interface

The user interface of the editor was tackled next. Because a user interface like in the Cove Trainer player was desired, the first step was to examine its existing user interface HTML. This additionally allowed to get a general idea how the player generated the user interface elements contained in the 'scene' property of the scenario file. Because the player was done with Angular 1, the HTML could easily be translated to Angular 2 for use in the editor. This translation included re-structuring the HTML to make accommodations for differences in Angular 2's component creation and differences in directive names. The finished user interface of the editor is represented in figure 12.

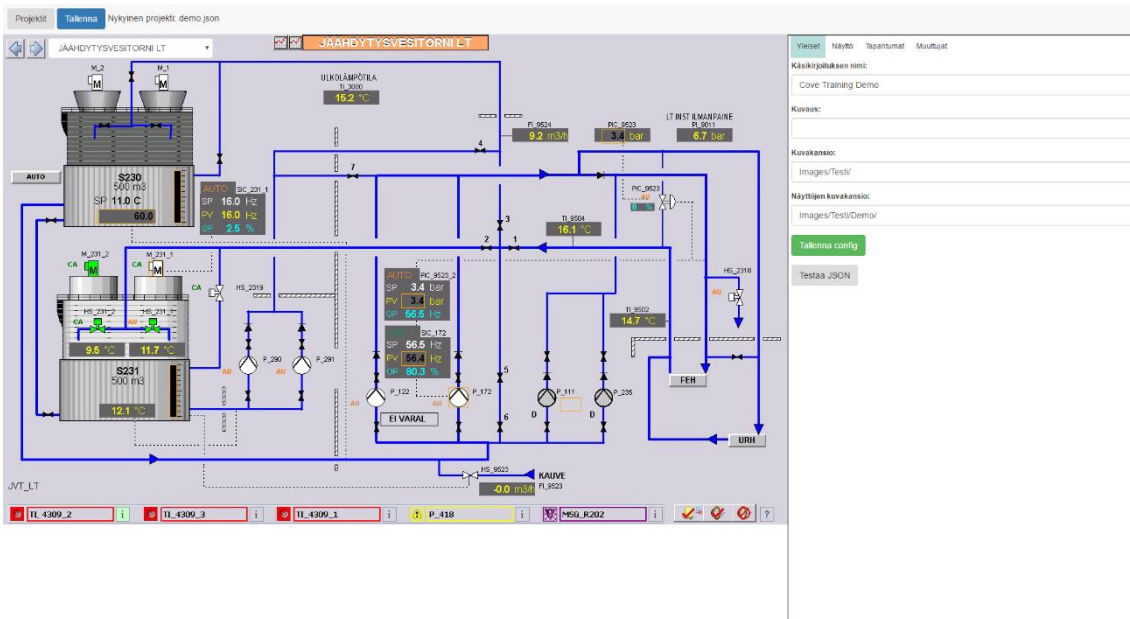


FIGURE 12. Editor user interface

Components in Angular 2 are identified by the framework through a name specified in the '@Component' directive, which it matches with component tags it finds in HTML. Figure 13 includes the HTML tag which the editor's root component is identified with in 'index.html' and the tag the scenario editor canvas is identified with in the root component's HTML file. A slight downside to this approach is that the tags are not block elements by default, which is generally undesirable in HTML container elements. This was fixed with an inline display style definition like in figure 13. The rest of the user interface would be included in the scenario editor's HTML template.

```

<body id="editor-body">
  <editor-app-root style="display: block;"></editor-app-root>
</body>
<scenario-editor style="display: block;"></scenario-editor>

```

FIGURE 13. AppComponent tag and the scenario editor tag in app.component.html

Figure 14 is an example how the user interface elements found in the 'scene' property are generated. This follows the same principle as in the player, which is to use the built-in repeater directive. In Angular 2, this directive is 'ngFor'. The usage of the 'ngFor' directive is simple, because it uses similar syntax as 'foreach' loop constructs found in well-known programming languages.

```

<div *ngFor="let switchItem of LoadedOperatorDisplay.SwitchItems" title="{{switchItem.Comment}}">
  <img [ngStyle]="GetSwitchItemStyles(switchItem)" [ngClass]="IsElementSelected(switchItem)" (click)="OnClickElement(switchItem)"
    [src]="ImageFolder + GetSwitchItemImage(switchItem)" />
</div>

```

FIGURE 14. User interface element repeater

There are also other important Angular 2 features in figure 14, which are the square bracketed HTML tag attributes and the 'click' attribute, which is in parentheses. The square bracketed attributes define one-way binding from the component class to its view, and in this case, they are used to define an element's CSS styling and image source URL. The CSS styles were constructed dynamically because the editor's working area had to resemble the Cove Trainer player as closely as possible. Due to this it was very important for the elements to be precisely positioned. Otherwise the designed scenario could look different in the player, which was undesirable. The parentheses on the other hand define a one-way binding from the component view to the component class. In this case, when a user clicks on the element image, the framework calls the function defined in the 'click' attribute. The function sets the clicked element to be the active element, which could then be edited.

Figure 15 illustrates the principle how the style creation was done. The function creates an empty object, assigns properties to it which are recognized as valid CSS attributes. It then fills them with the proper values from the element that was supplied to it as a function parameter and finally returns the filled object. This follows the way the player itself positions the elements, and allowed the editor to achieve an identical look. Because the style creation is not strictly related to enabling the user experience, services were created for each of the element types which would hold these style creation and other related functions.

```
public GetStyles(switchItem: SwitchItem): any {
  let css: any = {};
  css.left = switchItem.X + "px";
  css.top = switchItem.Y + 50 + "px";
  css.position = "absolute";
  css.width = switchItem.Width + "px";
  css.height = switchItem.Height + "px";
  css.zIndex = 10;
  if (switchItem.OutlinesDisabled == null || switchItem.OutlinesDisabled == false) {
    css.outline = "3px solid #ffccaa";
  }
  if (switchItem.Interact != false) {
    css.cursor = "pointer";
  }
  return css;
}
```

FIGURE 15. Element style creation

As a Cove Trainer scenario very likely has more than one operator display, the editor needed a way to change the active operator display. Two methods were created for navigation, simple buttons which would retrieve the next or previous operator display, and a list which could be used to jump to a particular display. Figure 16 depicts these features. The 'ngModel' and 'change'

attributes inside the 'select' tag together form a two-way binding. This separation allows for more control over the process of changing the value of the property bound to 'ngModel'. Every time a new value is picked from the list, the Angular 2 framework calls the function defined in the 'change' attribute.

```


<select id="systemui-operatordisplaylist" class="form-control" name="systemui-operatordisplays" [ngModel]="LoadedOperatorDisplay.Name"
(change)="OnOperatorDisplaylistPick($event.target.value)">
  <option *ngFor="let display of Scenario.scene.OperatorDisplays" [value]="display.Name">{{display.Name}}</option>
</select>
```

FIGURE 16. Operator display list and buttons to change the current one

Creating the player user interface view in the editor was however not enough. The actual editing interface was still missing. There was a part of the screen free for this purpose, which was normally occupied by a Cove Trainer player feature. The first task was to decide how the editing interface would be best to implement. After some consideration, a tab based system was decided to be best, because the free part was relatively narrow at 570 pixels in width. It also had to be always visible and editing functionality grouping had to be done somehow.

In figure 17 are the buttons used by the editing interface tabs and an example how the active tab is determined. The 'ngIf' directive works like an 'if' construct; if the statement it evaluates is true, it displays the HTML element it is declared in, and all its child elements. If the statement is false, the Angular 2 framework deletes the element and its children from the document until the statement is true again. The statement inside the curly brackets in 'ngClass' evaluates the same 'EditorState' property, and if it is found to be true, the 'tabbuttonselected' class is added to the element's CSS classes. This allowed to highlight the tab that was currently selected.

```
<div class="systemuiitem systemuitabbutton" [ngClass]="{tabbuttonselected: EditorState == 'edit-general'}"
(click)="OnChangeEditState('edit-general')">Yleiset</div>
<div class="systemuiitem systemuitabbutton" [ngClass]="{tabbuttonselected: EditorState == 'edit-operatordisplay'}"
(click)="OnChangeEditState('edit-operatordisplay')">Näyttö</div>
<div class="systemuiitem systemuitabbutton" [ngClass]="{tabbuttonselected: EditorState == 'edit-event'}"
(click)="OnChangeEditState('edit-event')">Tapahtumat</div>
<div class="systemuiitem systemuitabbutton" [ngClass]="{tabbuttonselected: EditorState == 'edit-variable'}"
(click)="OnChangeEditState('edit-variable')">Muuttujat</div>
<div class="systemuitabcontent" *ngIf="EditorState == 'edit-general'">
```

FIGURE 17. Edit interface tabs

Another part of the editing interface was the list of scenarios in existence. However, this functionality could not be added to the scenario editor component without changing the HTML too drastically. The solution was to create an additional component and service to handle the scenario list. This

division did create one issue. Because components are isolated from each other, there was no way for the scenario editor component to know when a scenario had been loaded.

The Angular 2 documentation on component interaction had an answer to the problem, with figure 18 illustrating this solution. By creating a service to handle communication between components and injecting it in the application's root component it was possible to get its child components to share the same service. The 'providers' property of the '@Component' decorator is responsible for this behavior. If used outside the root component, the framework would create a new service instance for that component instead of using the existing one. (2016, cited 12.11.2016.)

```
@Component({
  selector: 'editor-app-root',
  templateUrl: 'app/app.component.html',
  // This is the only place the 'providers' property should be used (excluding AppModule,
  // because the service won't even instantiate without a provider in AppModule),
  // if all of the children of AppComponent having access to the same service instance is desired.
  providers: [ICCSERVICE]
})
export class AppComponent {
  constructor(
    /** Inter Component Communication service. Provides communication services between components. */
    private ICC: ICCSERVICE) { }
}
```

FIGURE 18. Shared service declaration in AppComponent

The service created for component communication required three parts to work properly. A private property to hold the message source, a public property which components could attach to receive a notification when a function that changes the source is called. Figure 19 depicts the properties and function for the message that means a new scenario has been loaded from the server, which at this point were only hard-coded values as there was no back-end work yet done.

```
private newScenarioLoadedSource = new Subject<string>();
NewScenarioLoaded$ = this.newScenarioLoadedSource.asObservable();
public LoadNewScenario(scenario: string) {
  this.newScenarioLoadedSource.next(scenario);
}
```

FIGURE 19. Parts needed for component communication

While the scenario list component could now broadcast a message by calling the 'LoadNewScenario' function, the scenario editor component could not receive it. For that purpose, the component had to 'subscribe' to the message. In other words, an event handler was created for the message, which is illustrated in figure 20. Every time a new scenario would be loaded, the

event handler would be invoked, making the scenario editor component reset its state and refresh its properties from the scenario editor service. This event handler is created in the 'ngOnInit' lifecycle hook. If the hook is found, it is called by the framework every time the particular component is initialized. This ensures that the process happens as it should and it is recommended by developer guidelines.

```
this.ICC.NewScenarioLoaded$.subscribe(newScenario => {
  this.ResetComponent();
  this.Scenario = this.EditorData.Scenario;
  this.OperatorDisplayIndex = this.EditorData.OperatorDisplayIndex;
  this.LoadedOperatorDisplay = this.EditorData.CurrentOperatorDisplay;
  this.SelectedScenario = newScenario;
  this.EditorState = "edit-general";
});
```

FIGURE 20. Attaching an event handler for a component communication message

### 3.4 Editing functionality

Once the editor user interface was in good shape, it was time to implement the editing functionality itself. The process itself was quite straightforward, as there was already a well-defined data structure and an existing location on the interface where the user manipulatable elements would be placed in.

Figure 21 is an example of how the editing functionality was fundamentally implemented. Angular 2's powerful data-binding features required very little actual code for it to be possible to change the properties of editable elements. The changes made could also be instantly seen in the user interface, as changing a value in an input element meant that the document had changed. The procedure for properties which were not a simple primitive value was a bit different however.

```
<div class="form-group">
  <label for="element-height">Korkeus:</label>
  <input class="form-control" id="operatorelementheight" name="element-height"
    [(ngModel)]="ActiveElement.Height" type="number" />
</div>

<div class="form-group">
  <label for="element-width">Leveys:</label>
  <input class="form-control" id="operatorelementwidth" name="element-width"
    [(ngModel)]="ActiveElement.Width" type="number" />
</div>
```

FIGURE 21. Two-way binding of editable properties



In figure 22, the 'ngFor' directive was unable to process the 'variables' property, as it was not a conventional JavaScript array. Instead, it was defined as a more advanced JavaScript object called a 'Map', which has features comparable to a 'Dictionary' in C# for example. This meant that it had built-in functionality to retrieve an object by its key. This was problematic, as changing the data structure at this point would have needed re-writing in many places. It was not an uncommon problem however, and a Stack Overflow answer offered the needed information. A conversion process was needed to transform the 'Map' to a format the directive understood. (2016, cited 15.11.2016.) Figure 22 also illustrates how the transformation was specified through an Angular feature called 'pipes'. The syntax needed a vertical bar followed by the name of the transform identifier. In this case, the used transform was 'map'.

```
<div class="form-group">
  <label for="all-variables">Muuttuja:</label>
  <select class="form-control" name="all-variables" [ngModel]="ActiveElement.Variable"
    (change)="OnElementVariableChange($event.target.value)">
    <option *ngFor="let variable of Scenario.variables | map"
      [value]="variable.key">{{variable.value.Name}}</option>
  </select>
</div>
```

FIGURE 22. Binding object property

The transform function for a pipe is illustrated in figure 23. The class implements the 'PipeTransform' interface, with the actual transformation process happening in the 'transform' function. The answer was adapted for the editor's use by changing the pipe's name to 'map', which is more descriptive considering its use. The 'for' loop was changed to a 'foreach' loop. The 'Map' was passed in as a parameter, and an array of objects is created from the 'Map'. The array was then returned to the 'ngFor' directive, with the key and value pair being available in each 'variable'.

```
import {Pipe, PipeTransform} from '@angular/core';

@Pipe({name: 'keys'})
export class KeysPipe implements PipeTransform {
  transform(value) {
    let keys:any = [];
    for (let key in value) {
      keys.push( {key: key, value: value[key]} );
    }
    return keys;
  }
}
```

FIGURE 23. Transform function (Stack Overflow 2016, cited 15.11.2016)

One desired feature for the editor was the ability to change the position of the editable elements with arrow keys. For this purpose, an event handler for keyboard presses was needed. A standard 'document.getElementById' was used to hook into the key down event which happens every time the browser registers a keyboard press. Like the message event handler, the key down handler was included in the 'ngOnInit' life-cycle hook for the same reason, to keep within development guidelines.

The event handler for the key down event is depicted in figure 24. The handler function calls a separate function defined in the class itself to avoid cluttering the 'ngOnInit' function needlessly. The 'OnKeyDown' function itself first checks if any of the editable element size or position inputs have focus, because in that case the up and down arrows can be used to adjust the number. The key down check itself is a switch, which determines the pressed key by the keycode supplied in the 'e' parameter. It differentiates between the size and position changes by checking whether the shift key is held down.

```
ngOnInit(): void {
  document.getElementById('editor-body').addEventListener('keydown', (e: KeyboardEvent) => {
    this.OnKeyDown(e);
  });
}
```

FIGURE 24. Event hook in life-cycle hook function

While elements could now be selected, and modified, a way to add new elements and operator displays was needed. Buttons were an obvious choice for this. They were specified in the HTML with button tags and a one-way binding was assigned to the 'click' property. The binding would call an event handler which would then handle the element or display creation. There was still the question that how the type of the new element could be specified. The answer was to use a TypeScript feature called 'enum', which is a common feature in object oriented languages, like Java or C#. An 'enum' is basically a list of constant values which can be used to reliably and descriptively control logic flow.

In figure 25 is the 'enum' for the element types and the drop-down list in the scenario editor component template. A property containing the numeric value of the type of the new element is one-way bound to select element. The 'click' handler for the new element button takes the current value of the 'NewElementType' property and creates a new instance of the corresponding element class. It then calls a function in the scenario editor service to add the element into the scenario container, finally setting the new element to be the active element. The handler to add a new

operator display simply created a new 'OperatorDisplay' class instance, called a service function to add it to the operator display list and changed the new display to be the current one.

```
export enum ElementType { Switch = 0, Text = 1, Selection = 2, Bargauge = 3 };  
  
<div class="form-group">  
  <label for="operatordisplay-new-element">Tyyppi:</label>  
  <select class="form-control" name="operatordisplay-new-element" [(ngModel)]="NewElementType">  
    <option [value]="ElementType.Switch">Kytkiyelementti</option>  
    <option [value]="ElementType.Text">Tekstielementti</option>  
    <option [value]="ElementType.Selection">Valintaelementti</option>  
    <option [value]="ElementType.Bargauge">Palkkিয়েlementti</option>  
  </select>  
</div>
```

FIGURE 25. Enum declaration and corresponding HTML

The editing interfaces for the other properties found in a Cove Trainer scenario were created in the same way under their respective tabs, except there was no need to bother with visible interface elements. The remaining properties in the scenario container were 'events' and 'variables'. Of these two, only the 'variables' property had elements in it that would affect the user interface of the editor. Their values would be displayed by the operator display elements. Events on the other hand had no visible impact on the editor user interface. They only needed proper two-way bindings. Adding new events, their sub-objects and variables were implemented in the same way as the operator display elements.

Removing existing operator displays, their elements, events and variables was however a bit trickier. The requirements for the editor stated that events and variables had to have restrictions on their deletion. More specifically, it should not be possible to remove an event or variable if certain conditions were not met. A way to enforce these restraints was needed to implement the deletion correctly. The condition for deleting a variable was that it should not be used by any element capable of having a variable reference. The event constraint on the other hand was that no watcher should refer to the event about to be deleted.

These constraints were enforced by simply going through the editor data structures and checking whether a constraint violation was occurring. An example is illustrated in figure 26, which is the function used to check whether an event is safe to remove. If there was no violation, the function would return a true value. In case one or more violations had occurred, it would prepare an error message to show to the user.

```

public CheckCanDeleteEvent(eventId: number): { canDelete: boolean, errorString: string } {
  let returnObject = { canDelete: true, errorString: "" }
  var canDelete: boolean = true;
  for (let i = 0; i < this.Scenario.events.length; i++) {
    if (this.Scenario.events[i].Watchers != null) {
      for (let j = 0; j < this.Scenario.events[i].Watchers.length; j++) {
        let current = this.Scenario.events[i].Watchers[j];
        if (current.GoTo == eventId) {
          returnObject.canDelete = false;
          returnObject.errorString += current.Name + "\n";
          console.log(current.Name + " is referring to event with id " + eventId);
        }
      }
    }
  }
  if (returnObject.canDelete == false) {
    returnObject.errorString += " viittaa tapahtumaan " + eventId + ". Sitä ei voi poistaa!";
  }
  return returnObject;
}

```

FIGURE 26. Function to check if an event can removed

With the removal constraints in place, it was safe to implement the deletion functionality. For operator display elements, events and variables simple handlers were enough. They only had to deal with the currently active object. Operator displays themselves however required a bit more. Because each of the elements in an operator display had a reference to some variable in the variable list, the element reference counters for those variables had to be decremented somehow. The solution was to create a service function which would use existing element deletion code to remove every element in the supplied operator display. Only after that the function would delete the display itself. This ensured that the reference counters got decremented even when deleting a whole display.

Some of the element types used in the operator displays had images. With the editing interface being quite narrow, it was not possible to include the image selection in it. An alternative way was needed. The solution was to create an overlay that would appear on top of the interface. This was a convenient way because the editor user interface was built on absolute CSS positioning.

The image overlay HTML is depicted in figure 27. The visibility of the overlay is controlled with an 'ngIf' directive, which evaluates the 'ShowImagesOverlay' property. This property in turn is toggled through a button placed in the editing interface. This way, the images could be easily changed and they would not take up precious screen estate when not needed. The same principle was applied to the operator display background images and to an overlay for operator display selection elements. These elements have a drop-down list of possible values from which one could be set as active.

```

<div class="editor-images-overlay" *ngIf="ShowImagesOverlay">
  <div id="editor-images-overlay-entries">
    <div class="editor-images-overlay-entry" *ngFor="let image of EditorData.ScenarioImages.Elements">
      <img class="editor-images-overlay-image" [src]="ImageFolder + image" (click)="OnToggleImage(image)" />
      <label for="image-enabled">Valittu</label>
      <input type="checkbox" name="image-enabled" [checked]="GetImageChecked(image)" (click)="OnToggleImage(image)">
    </div>
  </div>
  <div id="editor-images-overlay-index">
    <h3>Elementin kuvat:</h3>
    <div class="editor-images-overlay-indexentry" *ngFor="let image of ActiveElement.Images; let i = index; let first = first; let last = last;">
      <span>Indeksi {{i}}</span>
      <button class="btn btn-sm" (click)="OnImageIndexChange('up', i)" [disabled]="first">Siirrä ylös</button>
      <button class="btn btn-sm" (click)="OnImageIndexChange('down', i)" [disabled]="last">Siirrä alas</button>
      <img class="editor-images-overlay-image" [src]="ImageFolder + image.source" />
    </div>
  </div>
</div>

```

FIGURE 27. Image overlay HTML

### 3.5 Communicating with the server

The only remaining thing to do was implementing the server communication functionality. The application had been built from the beginning to support this, as all data related functions already existed in service classes. The only thing that needed to be done was to exchange hard-coded values with real data from the server.

All server interactions were started due to user input, with the exclusion of the scenario list retrieval happening on application start. Thus, the HTTP requests were started in component class 'click' handlers by calling the appropriate service functions. An example of how the HTTP requests were done is illustrated in figure 28. This 'click' handler found in the scenario list component class is called when a scenario is selected from the scenario list. The function executes a 'ScenarioListData' service function and subscribes to it. The service function in turn calls another function belonging to the scenario editor's service class, which finally triggers an HTTP get request in the 'JSONReaderService'.

```

public OnScenarioListEntryClick(scenario: string): void {
  this.ScenarioListData.ChangeProjectType(this.CurrentProjectMode);
  this.ScenarioListData.LoadScenario(scenario).subscribe(() => {
    if (this.CurrentProjectMode == ProjectMode.Published.valueOf())
      this.CurrentScenarioIsPublished = true;
    else
      this.CurrentScenarioIsPublished = false;

    this.CurrentProject = scenario;
    this.ICC.LoadNewScenario(scenario);
    this.OnScenarioWindowToggle();
  }, error => {
    window.alert("Käsikirjoituksen lataus epäonnistui! Selaimen konsolissa lisätietoja.");
  });
}

```

FIGURE 28. Click handler which starts an HTTP request

This function chaining had a purpose. For some server interactions, every class involved in the chain needed to do some work after the data had been received from the server. Because the Angular 2 HTTP request functions return an RXJS observable, it was possible to do the required work through the observables themselves. Figure 29 illustrates how this observable chaining was done in practice. The function returns an object that is of type 'Observable' and attaches two handlers to it. Because the handlers were essentially anonymous functions, it was possible to do any kind of work within them. The 'map' function attached a handler that was called when the request was successful. The response parameter had the returned data from the server, which the service used to fill its component class facing properties. Finally, the handler returned the response down the chain to the 'click' handler which started the request. The 'catch' handler on the other hand simply returned the error message if the request failed.

```
public LoadScenario(scenario: string): Observable<any> {
    return this.JSONReader.LoadJson(LoadRequestType.ScenarioFile, scenario)
        .map(response => {
            this.selectedScenario = scenario;
            this.Initialize(response);
            console.log("Server responded to scenario file fetch with", response);
            return response;
        })
        .catch((error) => { return Observable.throw(error) });
}
```

*FIGURE 29. Service function for loading a new scenario*

Figure 30 depicts the function in 'JSONReaderService' which performed the communication with the server, which could take in two parameters. The request type parameter is used to specify which server URL the data should be retrieved from, with the second parameter being optional. It was used to further define the request URL. The retrieved data was transformed to a friendlier format before it was returned down the chain depending on the request type used. If the request was to load a scenario JSON file, the parser function would create a scenario container from the response, but in all other cases it was simply transformed into an array of strings.

```

LoadJson(requestType: LoadRequestType, urlComponent?: string): Observable<any> {
    var url: string = "";

    switch (requestType) {
        case LoadRequestType.ScenarioList:
            // This is different because we can't rely on the service's internal
            // and is just checking the published list, so we'd be loading stuff
            url = `${urlComponent}`;
            break;
        case LoadRequestType.ScenarioFile:
            url = `${this.ProjectBaseUrl}scenario/${urlComponent}/`;
            break;
        case LoadRequestType.ScenarioImages:
            url = `${this.ProjectBaseUrl}scenario/${urlComponent}/images/`;
            break;
    }
    console.log('load JSON:'+EditorSettings.ScenarioRootDirectory + url);
    return this.Http.get(EditorSettings.ScenarioRootDirectory + url)
        .map(response => this.ParseJsonResponse(response.json(), requestType))
        .catch(this.HandleError);
}

```

FIGURE 30. HTTP request function

The same principle was applied to the 'JSONWriterService', which handled sending data to the server. There was however a slight difference, because it was not very easy to write data to the filesystem by using JavaScript. Due to this there was no need to change any function calls in component classes and the data writing functionality could be directly implemented. Like in the service responsible for retrieving data from the server, a function was created to orchestrate the data sending operations. In figure 31 is a part of this function in 'JSONWriterService', which illustrates the principle how the function was created. The requests are differentiated by the request type parameter. The scenario name parameter tells the server which scenario it should work on, and for HTTP requests which needed a request body, the scenario file parameter was used. Depending on the request, a server compatible simple JavaScript object was created on the spot, or parsed into one as had to be done for scenario containers.

```

public WriteJson(requestType: WriteRequestType, scenarioName: string, scenar
var observable: Observable<Response>;
switch (requestType) {
    case WriteRequestType.ScenarioFile:
        observable = this.Http.put(EditorSettings.ScenarioRootDirectory
            + `${this.ProjectBaseUrl}scenario/${scenarioName}/`,
            this.ParseJson(scenarioFile as Scenario), this.Headers)
            .map(response => response.json())
            .catch(this.HandleError);
        break;
}

```

FIGURE 31. Excerpt of data sending function

## 4 CONCLUSIONS

As Angular 2 is a complete rework of the framework, many of the earlier version's principles do not apply anymore. The Cove Trainer editor works as an example of how a more complex application can be designed and implemented with Angular 2. The editor follows the Angular developer guidelines and documentation for best practices to a level it was practical to do so. The code is partitioned as outlined into HTML templates, component classes and service classes. This makes code maintenance and feature implementation easier, as the code is split into well-defined and independent blocks.

The biggest change however is the preferred language for the framework. TypeScript's optional type system can be a welcome addition to any project, as it prevents many bugs which arise from the fact that JavaScript is not a strongly typed language. The use of classes for all parts of the application is also a welcome change compared to the function based system used with Angular 1. Based on this project, TypeScript is certainly a more pleasant language to work with than pure JavaScript. Its main downside is that it needs a source code editor which has support for the language. Microsoft's Visual Studio 2015 and Visual Studio Code are a few examples of editors which can understand TypeScript. The language itself might look intimidating at first to those not familiar with statically typed languages, but even a little experience in Java or C# goes a long way.

It can be concluded that the editor meets the requirements set for it. The application can load a scenario JSON file from the server and interpret it. It can transform the scenario back to the format used by Cove Trainer and send it to the server to be saved. All properties within the scenario are modifiable, it is possible to create new properties and remove existing ones. The editor view is identical to the view presented by the Cove Trainer application itself, which was another important consideration. Most importantly however, the editor removes the need to manually edit JSON files, allowing the scenario designer to focus on the scenario itself. The client has also stated that they will be taking the editor into use in their next project and will continue developing it.



## 5 DISCUSSION

The purpose of the thesis was to create a graphical editor for Happywise's Cove Trainer web application. The main requirement for the editor was that it should be capable of reading and saving scenario files used by the Cove Trainer application. Another equally important target was to present a view that was identical to the application user interface. Because the editor itself was speculated to be quite complex, a framework was considered necessary to speed up development. Angular 2, the newest iteration of the Angular framework, was chosen for this purpose.

The topic itself was quite interesting. Web application development knowledge is arguably quite important nowadays, so it was a good opportunity to get more experience in that field. It was also a very good opportunity to learn how Angular 2 works and how different it was from the earlier version. It could even be said that it was enjoyable, as Angular's documentation was good, especially for a framework which was not fully released when this thesis was started in September. All relevant information could be found from the documentation in a manner that was easily understood. The take-away from this project was quite good; a solid grasp on how to build a fully functioning Angular 2 application which communicates with a server and is able to manipulate data according to user input.

Considering the whole process, the development task itself was quite straightforward. With a good theoretical background, it was only a matter of employing the principles laid out in theory. I suspect that TypeScript could have influenced this as well, as I feel more at home with strongly typed languages. The improved IDE features also helped, as code completion was much more advanced with TypeScript. The type system was an enormous help as well, because the editor relied on it quite heavily.

It can be said that the result was satisfactory, as the editor does not seem to suffer from severe bugs and the development targets were met. It was also a positive thing to know that Happywise intends to use the editor in future projects. There are still some things which could be improved. Currently, all of the scenario editing functionality resides in one component which is not necessarily a good thing if the application grows in size too much. To alleviate this, separate feature components could be created for events, variables and operator display elements. Another thing to consider would be ahead-of-time (AoT) compilation, as the application is currently compiled in

the browser on page load. This is slower than AoT and has a higher request overhead. The downside of this approach is that AoT has a different Node.js setup than the browser based compiler and it can be slightly more complicated to set up.

## REFERENCES

- Angular. 2016. Angular modules. Cited 5.9.2016,  
<https://angular.io/docs/ts/latest/guide/ngmodule.html>.
- Angular. 2016. Architecture overview. Cited 5.9.2016,  
<https://angular.io/docs/ts/latest/guide/architecture.html>.
- Angular. 2016. Component Interaction. Cited 12.11.2016,  
<https://angular.io/docs/ts/latest/cookbook/component-communication.html###bidirectional-service>.
- Angular. 2016. Dependency injection. Cited 7.10.2016,  
<https://angular.io/docs/ts/latest/guide/dependency-injection.html>.
- Angular. 2016. HTTP client. Cited 16.11.2016,  
<https://angular.io/docs/ts/latest/guide/server-communication.html>.
- Angular. 2016. Lifecycle hooks. Cited 16.11.2016,  
<https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html>.
- Angular. 2016. Pipes. Cited 16.11.2016,  
<https://angular.io/docs/ts/latest/guide/pipes.html>.
- Angular. 2016. Quickstart. Cited 1.11.2016,  
<https://angular.io/docs/ts/latest/quickstart.html>.
- Microsoft. 2016. TypeScript Language Specification. Cited 12.9.2016,  
<https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md>.
- Srinivasan, A. 2014. Cloud Computing. Internal reference. Cited 1.2.2016,  
<http://proquest.safaribooksonline.com.ezp.oamk.fi:2048/book/operating-systems-and-server-administration/virtualization/9789332537439/chapter-15-understanding-services-and-applications-by-type/ch15sec5.html?unicode=ouluuas>.
- Stack Overflow. 2016. Access key and value of object using \*ngFor. Cited 15.11.2016,  
<http://stackoverflow.com/a/39180582>.
- Stack Overflow. 2014. Javascript: Create and save a file. Cited 12.11.2016,  
<http://stackoverflow.com/a/29576427>.