



**BRNO UNIVERSITY OF TECHNOLOGY**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

**FACULTY OF INFORMATION TECHNOLOGY**

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

**DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA**

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

**LIBRARY OF REUSABLE COMPONENTS AND UTILITIES  
FOR THE ANGULAR 2 FRAMEWORK**

KNIHOVNA ZNOVUPOUŽITELNÝCH KOMPONENT A UTILIT PRO FRAMEWORK ANGULAR 2

**MASTER'S THESIS**

DIPLOMOVÁ PRÁCE

**AUTHOR**

AUTOR PRÁCE

**Bc. GABRIEL BRANDERSKÝ**

**SUPERVISOR**

VEDOUCÍ PRÁCE

**prof. Ing. ADAM HEROUT, Ph.D.**

BRNO 2017

**Brno University of Technology - Faculty of Information Technology**

Department of Computer Graphics and Multimedia

Academic year 2016/2017

**Master's Thesis Specification**

For: **Branderský Gabriel, Bc.**

Branch of study: Information Systems

Title: **Library of Reusable Components and Utilities for the Angular 2 Framework**

Category: User Interfaces

Instructions for project work:

1. Get familiar with existing client web technologies and study the Angular 2 framework in detail. Outline the principles for implementing UI components in this framework as well as principles for designing a user interface.
2. Explore and analyze existing libraries of reusable components for Angular.
3. Define requirements on a new library and its components. Design a consistent architecture and principles for the library's components.
4. Design and implement the core of the library, and the user interface.
5. Create an application or applications demonstrating the usage of individual components.
6. Evaluate the characteristics of the library and the created components based on user feedback. Evaluate the achieved results and discuss possible future work; create a poster and a short video for presenting the project.

Basic references:

- consult the supervisor

Requirements for the semestral defense:

Items 1-3, considerable progress on items 4 and 5.

Detailed formal specifications can be found at <http://www.fit.vutbr.cz/info/szz/>

The Master's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

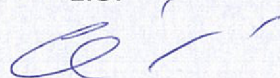
Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Herout Adam, prof. Ing., Ph.D.,** DCGM FIT BUT

Beginning of work: November 1, 2016

Date of delivery: May 24, 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ  
Fakulta informačních technologií  
Ústav počítačové grafiky a multimédií  
612 66 Brno, Božetěchova 2  
L.S.



Jan Černocký

Associate Professor and Head of Department



## Abstract

This thesis is concerned with the creation of a library of reusable components and utilities designed for use in data-intensive applications. One typical component of such applications is a table, which is considered to be the main component of the library. All other components and utilities are closely related to this component in order to ensure high cohesion. The final set of components can be used declaratively in various combinations. The user interface is accustomed for data-intensive applications with various elements.

## Abstrakt

Táto práca sa zaoberá vytvorením knižnice znovapoužiteľných komponent a utilít určené na použitie v dátovo-intenzívnych aplikáciach. Jednou typickou komponentou pre také aplikácie je tabuľka, ktorá je považovaná za hlavnú komponentu knižnice. Pre zaistenie vysokej kohezie sú všetky ostatné komponenty a utility sú s nou úzko prepojené. Výsledná sada komponent je použiteľná deklaratívnym spôsobom a umožňuje rôzne konfigurácie. Užívateľské rozhranie je tiež prispôbené na dátovo-intenzívne aplikácie s rôznymi prvkami.

## Keywords

UI components, utilities, Web technologies, Angular, UX, data-intensive applications, data table, table extensions

## Klíčová slova

UI komponenty, utility, Webové technológie, Angular, UX, dátovo-intenzívne aplikácie, datová tabuľka, rozšírenia tabuľky

## Reference

BRANDERSKÝ, Gabriel. *Library of Reusable Components and Utilities for the Angular 2 Framework*. Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Herout Adam.

# Library of Reusable Components and Utilities for the Angular 2 Framework

## Declaration

With this, I declare that this master's thesis was prepared as an original author's work under the supervision of professor Adam Herout. Additionally, I received the supplementary information in a company interfacewerk GmbH, namely from Kevin Merckx, Moritz C. Türck, and Sebastian Ullherr. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....  
Gabriel Branderský  
May 23, 2017

## Acknowledgements

I would like to express my thanks to the professor Adam Herout for his numerous advice, the guidance, and pointing me in the right direction. I am also very grateful for the opportunity to utilize this library in a real-world project inside interfacewerk GmbH. The support from interfacewerk GmbH was indispensable for the realization of this library.



# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Web Technologies</b>	<b>4</b>
2.1	The Language(s) of the Web . . . . .	4
2.2	Single-Page Applications . . . . .	5
2.3	Angular Framework . . . . .	5
2.4	React Framework . . . . .	8
<b>3</b>	<b>User Experience</b>	<b>9</b>
3.1	Usability Heuristics . . . . .	9
3.2	Techniques for Designing and Testing User Interfaces . . . . .	10
3.3	Developer Experience . . . . .	11
<b>4</b>	<b>Analysis &amp; Design</b>	<b>12</b>
4.1	Overview of Component Libraries . . . . .	12
4.2	Focus . . . . .	13
4.3	Use Case Application . . . . .	14
4.4	Evaluation of Existing Components . . . . .	16
4.5	Design Specification . . . . .	17
4.6	Architecture . . . . .	19
4.7	Public Interface . . . . .	21
<b>5</b>	<b>Implementation</b>	<b>28</b>
5.1	Implementation Language . . . . .	28
5.2	Project Setup & Tools . . . . .	29
5.3	Folder Structure . . . . .	30
5.4	Core Module . . . . .	30
5.5	Performance Optimizations . . . . .	34
<b>6</b>	<b>Demonstration</b>	<b>36</b>
6.1	Basic Example . . . . .	36
6.2	Custom Templates . . . . .	37
6.3	Utilities . . . . .	38
<b>7</b>	<b>Usability Testing &amp; Evaluation</b>	<b>39</b>
7.1	Iterations . . . . .	39
7.2	User Feedback . . . . .	40
7.3	Statistics . . . . .	41

8 Conclusion	43
Bibliography	44

# Chapter 1

## Introduction

The goal of this thesis is to create a client-side library, which is designed especially for data-intensive applications. The library incorporates components and utilities that are typically necessary for such applications, particularly optimized for the data volume and data complexity.

The idea for this library emerged from the need of a real-world application started in the beta version of Angular 2. Existing libraries could not satisfy all project requirements. It was concluded that it should be supported by a library, what consequently gave the birth to this thesis.

The final release of Angular 2 was published only in September 2016. Since it is not backwards-compatible, the lack of libraries is one of the biggest problems when starting with the new version of Angular. Many companies prefer to stay on the old version with the much richer ecosystem. The secondary goal of this work is to contribute to the ecosystem development by exploring patterns for designing third-party libraries.

The second chapter lies the technical foundation. It presents various web technologies, most importantly the Angular framework. Then, user experience (UX) is introduced in the third chapter since it is a necessity for the design and testing of user interfaces.

In the fourth chapter, the analysis of the current state of the art is intertwined with the comprehensive design of the library. Afterward, we transition from the design to implementation in the fifth chapter. The emphasis is put on the core module.

The design of the library is demonstrated on tangible samples in the sixth chapter. The penultimate chapter is concerned with usability testing. It evaluates the library based on the user feedback.

Finally, the last chapter reiterates the important information of the thesis and the results that were achieved.



# Chapter 2

## Web Technologies

This chapter introduces several web technologies with their specific terms and concepts. It is often the case that the same term is used by various technologies to denote (slightly) different meaning. The term “component” is particularly overloaded term, so it is defined in the section about the Angular framework among other things. Several code snippets are presented to enhance the understanding of concepts. The Angular syntax is used also a chapter demonstrating the usage of the library.

Another complementary view on a client-side framework is provided in the section about the React framework. Both Angular and React are intended for single-page applications.

### 2.1 The Language(s) of the Web

If there is a single language of the Web, then it is HyperText Markup Language [2]. It is present on all web pages unlike additional technologies (Cascading Style Sheets [1], JavaScript [4]) that might be skipped for some reason. However, different technologies have their specific purpose in the architecture of web applications. This architecture consists of a client and a server. The client is sending requests to the server and displays responses (in the graphical interface of a browser). At the same time, multiple clients can be served by one server. There might also be multiple servers handling one client request. Typically there is a database server dedicated to storing the data and application server processing the data. In Angular development, no particular server-side technology is assumed; therefore, only relevant client-side technologies are described in this section.

#### **HyperText Markup Language (HTML)**

The purpose of HTML is to define the structure and the content by embedding the markup into the text. This markup can often be found in many online editors allowing users to format the text because of its declarative nature. Even non-technical people can understand it, which is not true for many other programming languages.

#### **Cascading Style Sheets (CSS)**

Although HTML offers style tags, their use is discouraged; separating the structure from the presentation achieved with CSS brings several advantages. It is easier to maintain, and there might be several different styles for the same HTML document. Separated styles are useful for applying only on specific devices, e.g. smartphones.

#### **JavaScript & TypeScript**

In contrast to CSS and HTML that allow only static websites, JavaScript adds in-

teractive elements. JavaScript combines imperative, functional, and object-oriented paradigms. It is one of the most widely used languages. It is utilized not only in client but also the server applications, databases, and even embedded devices.

JavaScript, like any other language in active use, is evolving over time. The actual editions of this language are defined in the ECMAScript standard. The last version ECMAScript 6 was published in the year 2015, which represents the largest update to the language with numerous new features, e.g. classes. Luckily for developers, it follows the philosophy “Don’t break the Web”, so it is backwards-compatible. However, the support of browsers is problematic. To overcome this limitation, JavaScript can be transpiled from version ECMAScript 6 into lower version 5 with excellent support. Transpiling was anticipated, and ECMAScript 6 was designed for it.

Diverse languages can be compiled into JavaScript. One of them is [TypeScript](https://www.typescriptlang.org/)<sup>1</sup>, which is a superset of JavaScript. Although TypeScript was released already in the year 2002 by Microsoft, it considerably gained in popularity in recent years. This growth could be attributed not only to the added support of ECMAScript 6 but also as a result of Angular 2 being implemented in TypeScript. Despite that Angular 2 allows various languages, TypeScript is recommended by the Angular team.

Besides ECMAScript 6 features, TypeScript adds additional features on top of it. As the name implies, TypeScript also enriches JavaScript with optional typing system. If the type is not specified, TypeScript tries to infer the type from the context if possible. Otherwise, implicit type “any” is assumed.

## 2.2 Single-Page Applications

As noted, the foundation of the Web is the client-server architecture. Traditionally, each client request results in a completely new page. If a user just selects a different section in the sidebar, only a small portion of an application is changed. It is a waste to reload everything.

If an application has many interactive elements, it is better suited to be developed as a single-page application [9]. Single-page applications do not need a full page reload. Instead, the data are requested as needed by an application. In representational state transfer (REST), individual resources can be requested. One type of resource is usually associated with one URL. One common response format is (JavaScript Object Notation) JSON.

Figure 2.1 shows a thick client that communicates with a REST application programming interface (API). Firstly, the client loads the resources necessary at startup of an application in two separate requests. Because of the network delay, a response for the first request arrives later than the second. This is not a concern in the traditional model since all resources are always requested at once. Then, a user selects a specific item, which causes another request for that item. This process can continue until the application terminates.

## 2.3 Angular Framework

The Angular framework [3, 10] is a cross-platform solution for creating applications in client-side web technologies. It creates single-page applications, i.e. very interactive applications.

---

<sup>1</sup><https://www.typescriptlang.org/>

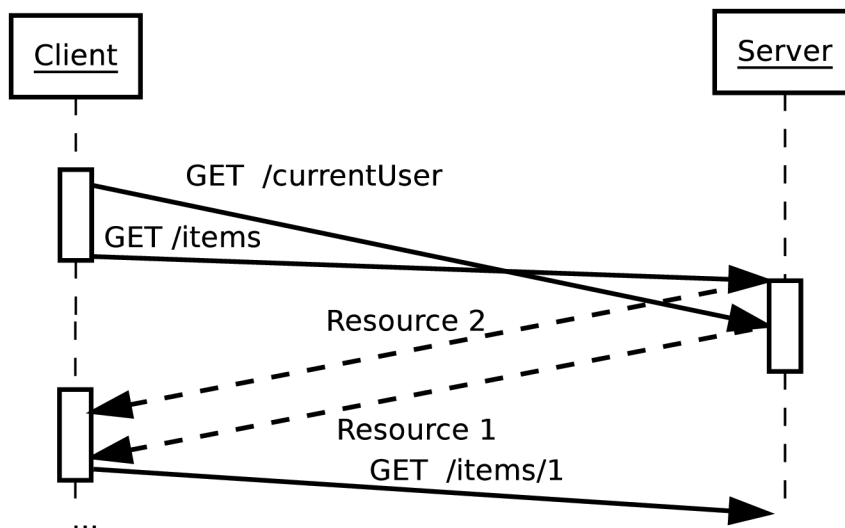


Figure 2.1: In single-page applications, a client requests only certain data throughout the life-cycle of an application.

The first version of Angular appeared in 2009 and over the years gained immense popularity. It proved its qualities for developing interactive user interfaces. At the same time, such wide adoption revealed some limitations (e.g. server-side rendering, lazy loading of modules) that could not be solved without significant architecture changes. On the grounds of these restrictions and advances in the web technology (e.g. ECMAScript 6), the Angular 2 framework was announced. It is a complete rewrite of the framework. Despite keeping some of the previous concepts, Angular 2 changed so dramatically that it could be considered a different framework. These changes inevitably impact libraries, typically resulting in their reimplementations. At this point, it is worth mentioning that an official migration path exists, but it is intended preferably for applications since libraries usually rely on advanced features and low-level interfaces that are more likely to be affected.

Any application of significant size needs to be decomposed into different parts to increase maintainability. For that purpose, Angular<sup>2</sup> possesses several primitives allowing decomposition and then helps with composing them together.

## Modules

Modules not only create a namespace but also structure applications into coherent and independent units. A collection of related primitives can be put into a separate module, then all of them can be included as a whole by importing the module.

## Components

A component is a basic building block of Angular applications. Every component owns a view created by the component's template and its logic. The logic implemented in the component's class is made accessible inside the template, which is an HTML file extended with Angular template syntax. Inside the component's template, other components can be declaratively used according to their defined interface. As

---

<sup>2</sup> By convention, it is preferred to refer to Angular without specifying a version number unless it is necessary. It is for the sake of upcoming major releases. There is already the planned release of Angular 4. This is only a small incremental update in comparison to a huge jump to the second version. This convention is followed. In this text, we always refer to Angular 2 and beyond if no version is specified.



a result, an Angular application is just a tree of components. The parent component communicates with its children by passing data, while a child component can communicate upwards only through events.

The previously described concepts together with template syntax are demonstrated in code listing 2.1 implemented in TypeScript. There is a class `ParentComponent` annotated with `@Component` decorator from Angular, which decorates the class with Angular-specific metadata. There are decorators with different metadata for each primitive. Inside this decorator, CSS selector specifies how the component is used (usually element name like in this case). Additionally, there is the inline template, which uses of another child component by passing one input and expecting one output, named correspondingly.

The square brackets should resemble assigning to a property of an object. Without square brackets, string “answer” would be passed as input into the component, instead of the content of that variable. For the output event, round brackets resemble calling a function. The implementation of a child component is not presented here, but we can assume that it triggers the output event under certain conditions, if a particular value is given to input. After the event is triggered, the expression on the right `onAnswerCorrect()` is executed.

Inside the class, only one property `answer` is defined and also annotated as `@Input` of the component. As a result, this component can be used in HTML as `<parent-component answer="42"></parent-component>`. If the attribute is not passed, then the default value of `answer` “default value” remains.

```
import {Component, Input} from '@angular/core';

@Component({
  selector: 'parent-component',
  template: `
    <child-component
      [input]="answer" (output)="onAnswerCorrect()">
    </child-component>
  `,
})
class ParentComponent {
  @Input() answer = 'default value';

  onAnswerCorrect() {
    alert('Correct. The answer is 42');
  }
}
```

Listing 2.1: Implementation of an Angular component.

### Attribute Directives

Attribute directive, unlike a component, does not need a template because it only augments the behavior of an existing element by applying as an attribute. Despite that, it can still have additional inputs and outputs. The demonstration of the usage (without implementation details) is in the following code listing. The tooltip directive

is applied to the paragraph by adding the attribute “tooltip”. The text attribute “tooltipText” is the input of that directive.

```
<p tooltip tooltipText="This is tooltip text" >
  Hovering over this element will display the tooltip.
</p>
```

Listing 2.2: The tooltip directive is added to a paragraph element.

## Services

It is very common that some logic is shared across multiple components. This is where services come into play. They can be injected into all components and provide higher abstraction. The implementation of data-layer is one common task for the service. A component just calls a method on service to store the record without caring about the implementation details. Data could be saved into local storage or on the database server by executing a server request. This kind of implementation details can change without affecting components as long as the interface remains unchanged.

## Pipes

Pipes in Angular are very similar to pipes in Unix systems. A pipe is a transformation of data implemented in one method. This is frequently needed in the templates for formatting values, but it can be used in the other parts of application too. Angular offers common pipes, e.g. date pipe for formatting date values.

## 2.4 React Framework

**React**<sup>3</sup> is a competitive framework developed by Facebook. In comparison to Angular, it does not try to do as many things. It is often referred as a view layer in the Model-View-Controller (MVC) pattern [9]. In that sense, Angular represents a complete MVC framework that has services as the model layer, component classes as the controller layer, and templates as the view layer.

Some of the concepts adopted in Angular 2 come originally from React, e.g. component architecture. Even concepts that are not in the Angular framework itself can still be present in Angular libraries. One such example is Redux, which defines three principles for state management:

1. An application state is modeled as one object.
2. An application state can be modified as a result of emitting an action. That means that every change is modeled as an object.
3. A state is modified only with a pure function called reducer. It accepts an application state and returns an updated application state.

Keeping a state and its changes in one place decouples it from the rest of the application. There are no duplicated application data, which need to be synchronized. This state object is a single source of truth.

---

<sup>3</sup><https://facebook.github.io/react/>

## Chapter 3

# User Experience

Human-computer interaction plays a key role in the process of creating user interfaces that communicate effectively with a user. The properties of a system are examined from the view of a user, not a system. For example, the response time of an action can be measured in milliseconds. From the purely technical perspective, the faster response means better system. That is not necessarily true for a user if the response is so fast that it is hardly noticeable. What is considered good practice for user experience is described in the usability heuristics section. Creating user interface additionally requires designing and testing. For that purpose, various techniques are presented. At the end of the chapter, the developer experience is discussed.

### 3.1 Usability Heuristics

These heuristics [12] describe general recommendations that should be followed when designing a user interface. These are only heuristics, not rules that should be strictly obeyed.

**Visibility of system status** User interface reflects the current state of a system. If there is an operation in execution or an error occurred, the user is informed about it.

**Match between system and the real world** The system presents information in such manner that is known to the user. It communicates in the user's language (terminology and concepts), not in a system oriented language.

**User control and freedom** The user is free to experiment, try functionalities of the system, and even make mistakes. There is an undo option to correct potential mistakes.

**Consistency and standards** Deviation from standards causes inconsistencies, which are the source of confusion or mistakes.

**Error prevention** Good user interface anticipates user errors and helps to correct them or even better to prevent them. That is far more important than just good error messages.

**Flexibility and efficiency of use** User interface adapts to the user and not vice versa. For first-time users, there is a tutorial with guided steps and for experts keyboard shortcuts.



**Minimalistic design** Any irrelevant information takes space and makes relevant information harder to find.

**Help and documentation** In the case of errors, there should be help or documentation available describing all steps for solving an error.

## 3.2 Techniques for Designing and Testing User Interfaces

So far, only general recommendations were described, which do not answer specific questions in regards to user interface. Luckily, there are some techniques [8, 13] dedicated to that purpose.

### Observation and contextual interview

Observation and contextual interview are techniques for determining the user needs. The basic assumption is that we can not simply ask users because they usually do not know how to solve their problems. They may not even realize what exactly their problems are, but if they do, they usually bury their needs with a multitude of other unimportant requirements.

These techniques are trying to understand the goals and individual tasks of a user and based on that design an appropriate user interface. In the observation session, users are simply watched during their work. In the contextual interview, specific questions about activities in their job are asked. The questions are deliberately targeted on the facts, not subjective opinions.

Combining these techniques might be practical. A few questions during observation can help to understand the user's activity when it is troublesome to deduce. Nevertheless, interrupting the natural flow of work should be avoided.

### Personas

Personas are only hypothetical archetypes of users, not actual users, which represent their characteristics, goals, and needs. Before creating personas, it is necessary to gather information, e.g. through observation or contextual interview. Personas create understanding and empathy for users, which help designers to avoid self-centered design.

### Thinking aloud

It is a simple tool for testing a user interface. The user is asked to say what is on his/her mind while trying the application. It is important to assure the testing users that they should not be afraid to criticize or express their opinion because that is the point of testing. During the testing, a user may have a list of tasks to finish. This process can reveal even small usability problems. A testing session may be recorded or at least notes about the problems are written down by the observer. Finding problems has no value if they are not addressed by proper adjustments.

### A/B testing

During this type of testing, several design proposals are compared. The difference might be subtle, e.g. different position of an element. In the end, only one proposal can be accepted. It may be determined subjectively by a designer or a user. Alternatively, each variant is shown only to certain users while measuring chosen parameters,

e.g. conversion rate, duration of stay, or frequency of use. Even small change in formulation of a sentence can have a substantial effect.

#### **User feedback**

Asking users for feedback can be invaluable despite the systematic mistakes users make, e.g. imagining the value of a future feature. Questions should take these mistakes into consideration to maximize the value; having something tangible (sketch, mockup, user interface) may help. Collecting of feedback can be conducted in any phase with different methods, e.g. in an interview or by submitting a form.

### **3.3 Developer Experience**

A user interface is intended for users; a programmatic interface of a library is designed for developers. In that sense, developers are basically users of the library.

Developer experience can be considered a subset of user experience. The main difference being that a library does not have a graphical user interface, but it is manipulated only with a text interface. Nevertheless, many concepts for designing user interface are still relevant for text interface. For example, the consistency is important regardless of the type of interface.

For the purpose of this text, we loosely define the developer experience by quoting a developer relations expert Pamela Fox [5]: *“Developer experience is the sum of all interactions and events, both positive and negative, between a developer and a library, tool, or API.”*

# Chapter 4

## Analysis & Design

The analysis of existing libraries at the beginning of this chapter is reflected in the focus of the library. Then, a use case application is introduced as a source of requirements for the design and evaluation of the suitability of existing libraries.

The principles affecting both the architecture and interface of the library are defined in the design specification. The details of the architecture and the public interface for individual components are given in the final sections.

### 4.1 Overview of Component Libraries

Most of the existing Angular component libraries are based on some CSS framework such as Bootstrap, Angular Material, or Zurb Foundation. These CSS frameworks ship with several widely used components, more precisely with the styles for components. Then, it is the job of developers to create a corresponding structure in HTML. An Angular library simplifies the usage of these components by providing dedicated HTML tags with appropriate parameters. It is as if the components were supported directly by HTML. It is not rare to see multiple libraries based on the same CSS framework since there can still be differences in a supported version and API.

#### NG Bootstrap<sup>1</sup> & ngx-bootstrap<sup>2</sup>

Both libraries provide seamless integration of Bootstrap 4 components, but only the former is backwards-compatible with Bootstrap 3. In included components, there are tabs, alerts, pagination, progress bars, modal windows, drop-down menus, etc.

A number of team members of NG Bootstrap are core contributors to Angular, and ngx-bootstrap backed up by Valor Software – a company with many open-source contributions in Angular – is not behind at all.

#### Material 2<sup>3</sup>

Partial implementation of Material Design specification in Angular 2, which currently supports around half of the components from the design specification. The idea behind Material design is to make user interface (UI) look and feel similar to the materials in the real world. Since it was one of the first libraries started in Angular 2 by a team

---

<sup>1</sup><https://github.com/ng-bootstrap/ng-bootstrap>

<sup>2</sup><https://github.com/valor-software/ngx-bootstrap>

<sup>3</sup><https://github.com/angular/material2>



in Google, it also states that it tries to be an example for Angular 2 developers by following best practices and implementing high-quality components.

#### Covalent<sup>4</sup>

Covalent is built on top of Material 2. It supplements Angular Material with additional components, various basic layouts, design style guide, and patterns. In addition, Covalent provides services for unit testing and integration testing.

#### Angular UI<sup>5</sup>

Angular UI is another well-established library or “companion suite(s)” as it describes itself. It consists of 26 various modules, each of which has a specific purpose ranging from small validation module to fully-featured UI-router for flexible routing. Angular UI became a brand guaranteeing the quality of individual modules. However, there is almost no support for Angular 2 applications at the moment.

#### Onsen<sup>6</sup>

This component library is based on Polymer library, the platform for extending HTML in compliance with Web Components standard. That makes the library framework agnostic. It can be used with many popular client-side frameworks like React, Vue, Meteor, and Angular. This library is especially focused on mobile applications. The styles of components are highly customizable.

#### PrimeNG<sup>7</sup>

This library contains an overwhelming number of components with own mobile-friendly styles. It also comes with various themes for components. It is developed by PrimeTek Informatics, which is a company focused on providing open source UI components.

## 4.2 Focus

The previous section demonstrates that components must be united in some way in order to form a component library. Otherwise, it is just a collection of random components put together. A uniting factor could be the style, the architecture or a dedicated purpose.

As noted in the introduction, we focus on data-intensive applications. It is an intentionally ambiguous term to avoid being too restrictive. Developers should decide based on their subjective judgment whether this library is suitable for their use case. A developer may decide to use the library for a simple application with a minimum of data if it is expected to grow. Despite the overhead, the simple application still benefits from the library. One representative example of a data-intensive application appropriate for the library is given in the following section 4.3.

The current focus does not imply what components should be included in the library although it reduces the number of possibilities. One component that is closely related to data-intensive applications is a table. It is the most typical component that appears in any substantially complex system. The main concern with data-intensive applications is the amount of data that must be displayed in the user interface. On one side, the data should

---

<sup>4</sup><https://teradata.github.io/covalent/#/>

<sup>5</sup><https://angular-ui.github.io/>

<sup>6</sup><https://onsen.io/angular2/>

<sup>7</sup><http://www.primefaces.org/primeng/#/>

be available to the user. On the other hand, the user should not be overwhelmed with too much data. The table allows to present the data in a condensed form well-known to users. However, the default HTML table is not user-friendly for displaying records with dozens of columns. In this data volume, all shapes of data could be found, figuratively speaking. So data complexity is another concern. It is not enough to consider all field types in the database, because one integer type may represent age, money or grade. The values must be presented in a way that is the most appropriate to the user. Additionally, the user should be able to perform actions that help him to work with the data, e.g. sorting. These features are discussed in detail in the context of the use case application.

The choice of the first component is an important design decision that affects the following components because it is desired to keep the high cohesion among the components. In other words, all components should be related one to another. In that sense, the first component adds additional restrictions. For the sake of very high cohesion, we extend this restriction even further by considering the table to be the main component. Other components must be able to operate in connection with the table component. For example, a hierarchical recursive menu fits into the context of data-intensive applications, but it does not fulfill that requirement. The menu could be used alongside the table, but it is hard to imagine any cooperation with the table component. An example that satisfies this restriction is a search component that filters the table rows.

So new components are added only with this constraint in mind. This effectively limits the possible components. Thereby, the focus is sufficiently narrowed down.

### 4.3 Use Case Application

The purpose of a use case application is to act as a reference point for the design. It allows to address specific issues on a concrete case what makes the design clearer than an abstract interpretation. However, the use case application is just one representative example of a group of applications supported by the library. For that reason, the creation of a library is much harder than building an application. The library must anticipate the needs of different applications independently on its domain. On the contrary, the application is tightly coupled to certain data. So if an application data change tremendously, the application is very likely to need more updates than a library.

The use case application is based on a real project for a university. The primary goal is to manage the information about the former, current and applying students. The users of this system are either professors or officers who need to work with students' information. For example, a professor may want to send an e-mail to students who are enrolled in his/her course. Similarly, an officer can update a student's status if he/she interrupted the study.

Since users work with the system daily, it is desired that the users can perform their tasks as quickly and easily as possible. The first challenge is that only the student entity contains around 50 fields in the database. Not to mention that users may also need to see the associated information from other entities. Displaying the records in the table allows users to look at many students at once, but it quickly becomes overwhelming if there are too many columns. Therefore, the information load on the user should be reduced by hiding it until it is needed. A professor, unlike an officer, is less likely to need the address of a student. So the users should be able to predefine what information is relevant to them, and also quickly change it if a task at hand requires other information.

Overall, there are many features that can save users time when working with a table data. Sorting rows by a column helps to find the student with the highest score quicker

than a user would do manually. Being able to see related information such as studies in the context of other columns is also beneficial. So it should be possible to display them in the table despite that it is another entity. One issue with that is that one person can have many studies (i.e. 1 to N relationship), but practically there will be no more than dozens of studies. So it is unnecessary to display studies in another view, e.g. as a separate table. This applies to many other entities, e.g. an address or a list of addresses.

Here is the summary of required features for the use case application from the user's perspective:

1. Dynamically change which columns are currently visible.
2. Sort columns, either in ascending or descending direction.
3. Change the order of columns.
4. Display complex structured data inside one cell, e.g. an address or several studies.
5. Allow further selection of subfields in the case of complex data.
6. Paginate the records if there are too many of them.
7. Filter on table data.

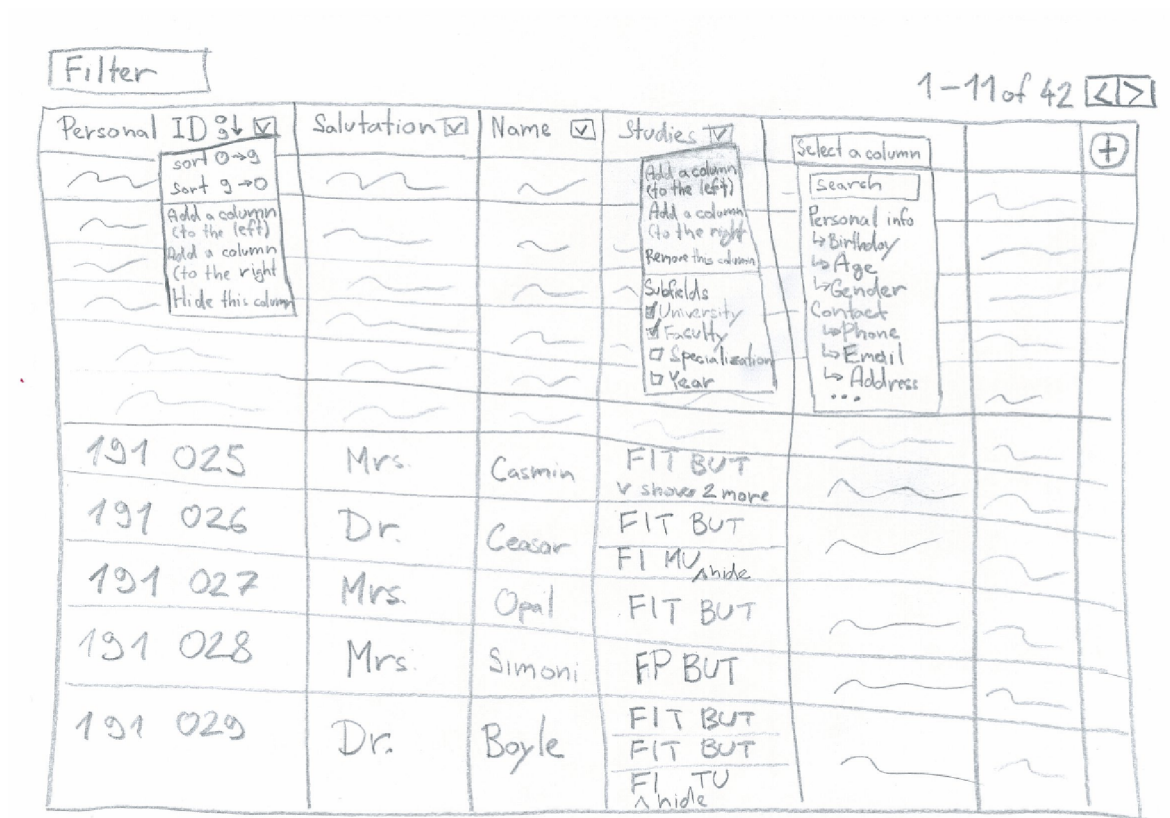


Figure 4.1: A Sketch of the user interface for the use case application.

These requirements bring many new elements that require a proper user interface. For that purpose, we employ a simple, yet powerful technique, i.e. sketching [8]. Figure 4.1

shows a user interface sketch of the table library. Firstly, visible columns are limited according to user preferences. Users can customize them dynamically in various ways. For enabling additional columns, there is a plus sign in the last header cell of the last empty column. After clicking the plus sign, it switches to a select element with a list of available columns grouped into categories.

Additionally, each column has a context menu with an option to add a column at the particular position next to it. In the sketch, the column is being added to the right from “Studies” column. For the reverse effect, there is also an option for removing a column. Two more options are for sorting columns in the desired direction. Column “Personal ID” is sorted from the lowest to the highest values. Since this is very common action, the column name can be clicked to sort or reverse the sorting direction.

For filtering, there is an input element above the table. Typing into it immediately limits the rows depending on the match in any column.

Some menu options are available only for columns with complex data types such as “Subfields” in the “Studies” column. These options modify the display of column cells or more precisely sub-cells. Each study cell contains a list of studies for one person. One study is an object with multiple properties. By checking off a subfield, a corresponding property is displayed in the sub-cell.

Lastly, the records are divided into pages. The currently displayed number of records is displayed in the right corner. A user can use arrows to navigate the pages.

All the actions from a perspective of a library are modeled in the use case diagram in Figure 4.2. The user actions are numbered according to the requirements at the beginning of this section.

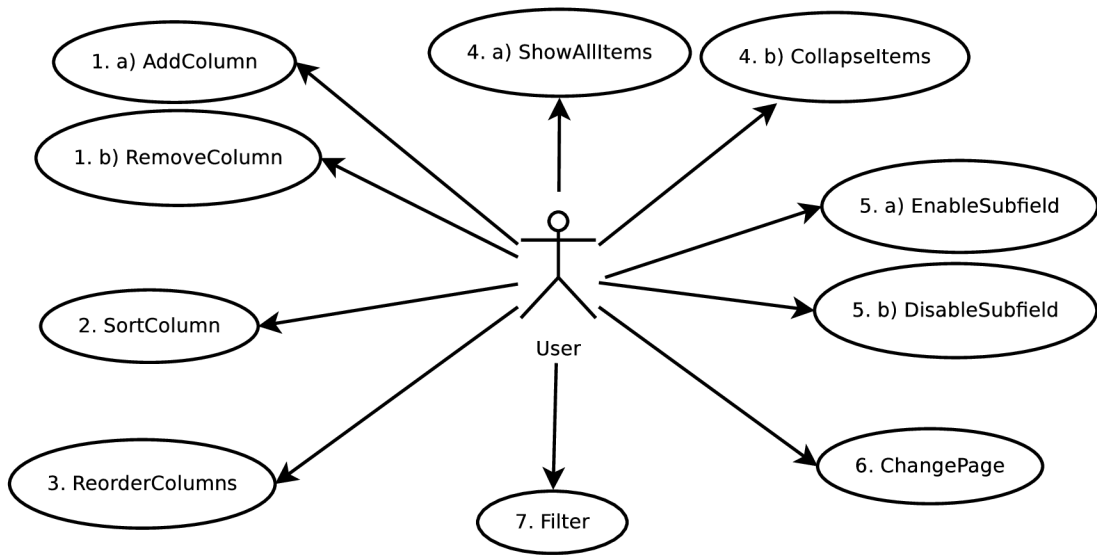


Figure 4.2: The use case diagram for a library based on the requirements of the use case application.

## 4.4 Evaluation of Existing Components

This section examines the suitability of existing libraries for the use case application. At this point, we are concerned with the individual components rather than just a library as

a whole. Firstly, we take a closer look at the components from the previously mentioned libraries in section 4.1 and then introduce additional single-purpose libraries.

	ng(x)-bootstrap	Covalent	Prime NG	ag-grid	ngx-datatable
Column toggling	no	no	yes	yes	no
Sorting	no	yes	yes	yes	yes
Reordering	no	no	yes	yes	yes
Complex types	no	no	no	no	no
Subfield toggling	no	no	no	yes	yes
Pagination	yes	yes	yes	yes	yes
Filtering	no	yes	yes	yes	yes

Table 4.1: The table compares the various libraries in terms of supported features. It does not list all features for each library. The listed features are based on the requirements of the use case application. The first column shows the support for ngx-bootstrap and ng-bootstrap at the same time.

Although Bootstrap provides some styles for tables, there are no components for tables in neither of the two Bootstrap libraries. Only the pagination component is relevant. Covalent has a data table component, but it lacks the complex data types, column toggling, and drag&drop reordering. Moreover, these component libraries are best suited for projects that already use a corresponding CSS framework. The components then fit the style of an application.

The data table of Prime NG meets almost all requirements, except for complex cell types with a subfield selection. Instead, it has many other features like facets, exporting, etc.

One of the outstanding data table solutions is [ag-grid](https://www.ag-grid.com/)<sup>8</sup> where “ag” stands for agnostic. It is compatible with many different technologies, e.g. Angular, React or pure JavaScript. The project strives to be the world’s best JavaScript data grid for Enterprise. The entire ag-Grid company is devoted to this goal. Several license types are offered by the company. Nearly everything is included among the features of ag-grid, e.g. sorting on columns, filtering rows, selection of rows/cells, and grouping by values, except for complex cell types such as an array.

The ag-grid may also be too heavy for smaller business applications. A lighter alternative is [ngx-datatable](https://github.com/swimlane/ngx-datatable)<sup>9</sup>, which is available for Angular 2 and beyond. It creates virtual DOM to handle large data sets. It also has intelligent resizing of columns and vertical and horizontal scrolling. Nevertheless, it does not have column toggling, nor complex cell types.

None of the libraries satisfies all requirements as summarized in table 4.1. However, not only features are important as we briefly touched. The following section discusses other aspects that cannot be so easily quantified.

## 4.5 Design Specification

Each of the existing component libraries from section 4.1 possesses a different set of characteristics. One characteristic can be found in many libraries and is not so interesting by itself. However, each library has a unique combination of these characteristics. In similar

<sup>8</sup><https://www.ag-grid.com/>

<sup>9</sup><https://github.com/swimlane/ngx-datatable>



fashion, this design specification results in a distinct set of characteristics that are especially appropriate for our use as outlined in section 4.3.

### **Declarative**

The simplicity of HTML is a result of its declarative syntax. So learning HTML is much easier than learning JavaScript. On the other hand, JavaScript is more expressive.

The philosophy of Angular is to be as declarative as possible. Angular was originally developed for designers who know HTML but not necessarily JavaScript. However, it turned out that even experienced JavaScript developers can benefit from the declarative syntax.

One benefit of the declarative use of components is that the creator is put into the right mindset. Developers have to think about the interface of a component before they can proceed with its imperative definition. On the contrary, starting with the focus on implementation details may produce a component that is difficult to use.

Developers spent a lot of time using libraries, which requires them to know them in the first place. Using a declarative library is generally easier than an imperative one since it specifies what to do, not how to do it.

For the benefits mentioned above, we inherit the Angular's mission for declarative interface whenever possible. In some cases, it makes more sense to give the developers more power with an imperative code.

### **Convention over Configuration**

Favoring convention over configuration is a well-known concept thanks to a server-side framework Ruby on Rails. It was adapted to many other frameworks and libraries

The library should provide sensible defaults so that the developers can get started quickly. A configuration can be just predefined to a specific value, implied from the name, or computed from the available data.

Preferring convention over configuration helps to reduce a lot of boilerplate code. Naturally, developers can always override the defaults if the default does not fit their needs.

### **Agnostic of the CSS framework**

This library can be used with or without a CSS framework. Different applications may favor various CSS frameworks, but not necessarily components. This decision, being CSS framework agnostic, comes with some trade-offs. Some teams may prefer only a certain CSS framework and components made for it. The integration is then very straightforward. This is particularly important for low abstraction components that appear frequently, e.g. form control. Nevertheless, using a certain CSS framework might not be advantageous nor feasible for all components, especially not for high-abstraction components. Not to mention, it could be even considered the duplication of the effort to implement them in all CSS frameworks. The parts that are framework specific usually constitute only a small fraction. There are alternatives how to avoid them. It comes only at a price of higher implementation cost.

### **Levels of abstraction**

Last but not least, the library exposes APIs with different levels of abstraction. The

idea is to leverage the advantages of a certain abstraction level and mitigate its shortcomings. The high-abstraction API accomplishes a lot with little effort, but it offers less control. The risk that a developer comes to the blind alley is very high if only high-abstraction API is exposed. On the contrary, the low-abstraction API is flexible, but it takes more effort to get the things done. Naturally, it is not just black and white, so a whole range of abstraction levels exists. Finding the right level of the abstraction for components is the design challenge discussed in section 4.7.

### **Fallback**

The developer should be able to gradually migrate out of the library. It may seem counter-intuitive to prepare a fallback solution because the library should try to keep its users, but not at all cost.

Let us consider a typical developer workflow, which starts by picking one library that seems to be the most appropriate. After a successful setup, it continues to live in the application with occasional adjustments. In some cases, it turns out that a library needs to be replaced. After some time, it is either not the best fit for the application anymore or a new requirement requires the library to work differently. Libraries are not general-purpose frameworks, even similar libraries may be optimized for certain situations.

If a library needs to be replaced, a developer should be able to do it quickly. If we imagine that a developer is working on a feature estimated for a few hours, then it turns out that a whole library needs to be replaced. A few hours would become several days. The library has the power to avoid this situation, but it is also a matter of empathy, i.e. being able to put yourself into the shoes of a developer in such situation.

There are many other important characteristics such as performance that are not mentioned here as the main characteristic, but not neglected.

In summary, it all comes down to one thing, which is the adaptability. The library should adapt to its users, not vice versa. Developers who are getting started with the library can leverage declarative components with the high level of abstraction. Experienced developers are able to get more control from the library. This is in correspondence to the heuristic “Flexibility and efficiency of use”.

Similarly, projects can be setup with different CSS frameworks or libraries. Nevertheless, it should be possible for them to use the library regardless of their setup. The only thing that matters is the suitability for their use case.

Projects usually do not have fixed a set of requirements. From the initial requirements, it is pretty easy to know the suitability of a library for current use case; however, requirements may change resulting in different needs for the library. Even in cases like this, the library allows users to react to new situations.

## **4.6 Architecture**

Despite that the library is built for the Angular framework, the design should be independent of the implementation. Therefore, we restrain from using Angular-specific terms and use only the broader terms such as component architecture. This is a foundation of many client-side frameworks, e.g. React. As a result, the architecture of the library can be adapted to other client-side frameworks.

One of the most important tasks in the architecture is to divide the responsibilities and decompose them into coherent units. Since the library adheres to the component-based architecture, the responsibilities are divided among the components. Based on the use case application, several independent components can be identified:

### **Data Table**

As previously mentioned, this is the main component of this library. It renders a complete table with all its data and the functionality.

### **Data Select**

This is an enhancement of the HTML select element for data-intensive applications. If there are too many options, it is not easy to find the one we are looking for. Therefore, the data select element allows the user to search for it.

### **Pagination**

It is advantageous to split the records of a table into several pages to decrease the loading time and increase the responsiveness of an application. This component displays the pagination controls as sketched in Figure 4.1. The controls can be configured to look differently. The actual pagination is executed by utilities that can connect it to the table.

### **Filtering**

Similarly, there is a component that allows users to search. The filtering controls can have different variations. In the simplest case, it is just one filtering input as shown in the sketch.

### **Additional utilities**

Generally, these are either some helper methods or classes implementing some logic. Just to name a few examples in Angular, it can be a service for filtering, a pipe for pagination or a directive for drag&drop reordering.

As shown in diagram 4.3, each of these components is classified into a separate module on the left because developers should be able to pick only one specific module. They may need only a data select from `SelectModule` or just the pagination controls from `PaginationModule`. Naturally, importing a module brings all its dependencies. So importing a data table from `TableModule` also imports the dependent module `SelectModule`.

Unlike other elements, the data table represents a big building block so it should be decomposed further. The main component `TableComponent` consists of two subcomponents, namely `TheaderComponent` and `TbodyComponent`. Each subcomponent is composed of even smaller subcomponents. The subcomponent `TheaderComponent` renders a table header while utilizing `AddColumnComponent` for adding new columns, and `ThComponent` to render a header cell. Similarly, `TbodyComponent` renders a table body while utilizing `TdComponent` to render body cells. Complex cell types are implemented in separate components `ArrayCellComponent` and `ObjectCellComponent`.

The components of the table module take advantage of the service layer. It is a place to put shared state and logic for all table components. Each service has a dedicated purpose:

**TableInitService** The service is responsible for the complete initialization of the table. It sets the default values and automatically detects the configuration from the data if the configuration is missing.

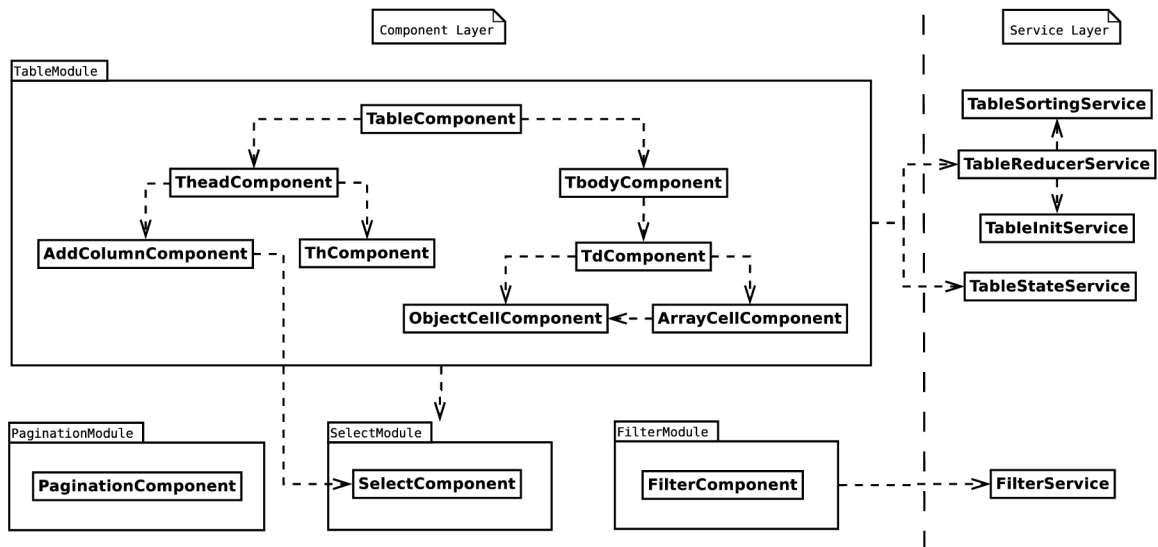


Figure 4.3: The diagram depicts the architecture of the library in two parallel layers. The layer on the left defines a multitude of components divided into four modules. The service layer mostly holds the state and the logic shared for all components of the table module.

**TableSortingService** The sorting functionality is implemented in this service.

**TableStateService** It holds the state of a table, which is accessible to any component.

**TableReducerService** This service acts as a facade because all events are processed here. An event can be delegated to another service.

The design of services is inspired by three principles of Redux as listed in section 2.4:

1. The table state service keeps a state in one central place.
2. All table components only dispatch an event instead of modifying a state directly.
3. An event effectively describes a change that should happen. The state change is then performed by the table reducer service.

The services provide another possibility for the further customization as they can be replaced. To accomplish that, a developer creates an alternative service with the same interface and swaps it with the default service.

## 4.7 Public Interface

At this point, the responsibilities are divided among the components. However, we did not define a public interface that is accessible to the user of the library. Each component defines an interface, but not all components have to be publicly available. If a component is dedicated only for internal use inside a library, then its interface can be changed freely without affecting any users. If a component is exposed, then its interface should be carefully considered in order to avoid breaking changes.

The first step is to decide which components should be publicly accessible for each module in diagram 4.3. There must be at least one component published per module for a developer to be able to use the module at all. That sets it straight for modules with only one component, but what about the table module with a multitude of components?

Let us start by exposing only the table component, so other components are its internal details. Since the component acts as a black box, its internal behavior can be adjusted only through a limited number of inputs. That means that the table component should provide an abstraction that is powerful enough to fulfill the requirements for countless variations of valid data. If we consider the inputs for sorting, the developer should be able to set a column that is sorted by default when a page loads. Additionally, it should be possible to disable the sorting on a specific column or to disable sorting completely. If a column is sorted for the first time, then it makes sense to sort it either in ascending or descending order depending on the column. For instance, column “name” should be sorted ascending and column “updated at” should be sorted in descending direction. So this should be configurable as well. In a similar fashion, we consider various inputs and outputs for other requirements (reordering, complex data, editing, column toggling and filtering), which are listed at the end of this section.

What if the developer wants to slightly adjust the behavior of the table component, but there is no corresponding input for that? For example, there should be a column called “Actions” filled with buttons to execute an action on a row. Since this feature is application-specific, it probably does not make sense to add it to the library. If not, the developer must replace the whole table, either by another library or by a custom implementation. This should not happen. It is very hard to create a good abstraction that anticipates all possible use cases.

What if we expose all components present in the table module? That means we must carefully design inputs and outputs for all components from the table module. This gives developers more flexibility for the price of learning additional application programmatic interfaces (APIs). The upper components have the highest abstraction level. To gain more flexibility, a developer can access building blocks of lower abstractions. They can be combined, complemented or replaced as needed. In that sense, the high abstraction API is a shortcut rather than an abstraction. It just combines the lower-level components in a typical way.

At first, it might seem that publishing all components of the table module is against encapsulation. It is possible to access its internal details, thus a component does not act like a black-box anymore. However, each component still has its private state that can be changed without any breaking changes.

After publishing the components, it is necessary to specify a public interface for every component. The inputs and outputs for each component are listed in the following subsections.

### Pagination Component

The pagination component only displays a user interface for changing pages. It has the following properties to be able to display controls for pagination:

**initialPage** In many cases, initial page value needs to be initialized, e.g. a page is present in URL. The default value is 1.

**itemsPerPage** The number of items per page is necessary to compute the number of pages. If not provided, the default value (10) is used.



**totalItems** The total number of records is also necessary to compute the number of pages. It is a required input.

**paginationLabel** The text to be displayed before pagination controls.

**showInput** The user can change the page by entering a number into an input element. This interface is disabled by default.

**showArrows** The user interface displays arrows to navigate pages. Both mechanisms for changing pages can be active at the same time.

**pageChange** This is an output event notifying about a page change.

## Select Component

This component supports the features of the default select element, e.g. grouping. Despite that it is rather a small component, there are still many properties for its customization:

**items** There are several types of items. Firstly, it can be an array of strings to be displayed as select options. This is just a shorthand for specifying an array of objects which have the same identifier **id** and a user-readable name **text**. Lastly, items can be optionally grouped into categories:

**id** for a unique identifier of a category.

**text** for a user-readable category name.

**children** for nested simple items.

**isOpen** The selection menu can be set to be initially open. The default value is false.

**allowClear** If clearing is not allowed, then a selected value can not be reset. This is useful if there must be at least one selected value. The default value is false.

**isCategorySelectable** Selection of a category can be enabled with this option. The default value is false.

**placeholder** Placeholder value of a select, e.g. please select a value.

**searchPlaceholder** The search input displays a placeholder. The default value is "Type to search".

**focusSearch** The search input box can be focused when the selection menu is opened. The default value is true.

**noOptionsMessage** Another message is displayed if there are no available options.

**ngModel** An initial value can be preset with this input.

**itemTemplate** A custom template that is rendered for each item. Naturally, it can assess the item variable.

**itemSelected** This output event notifies about the selection of an item. The selected item is available in the event data.

**categorySelected** In case that a selection of categories is enabled, this event is triggered when a category is selected. The event data contain the selected category with its items.

**open** The output event notifies that a select state is open. A user can choose a value.

**close** The output event notifies that a select is closed.



## Filtering Component

The filtering is implemented in `FilterService`, which can be controlled with the user interface of this component. The component can be adjusted with these properties:

**records** It accepts records for filtering in this input.

**instantFiltering** The filtering is performed either immediately after typing a text or by clicking on the search button. The default value is true.

**placeholder** The filtering input displays a placeholder text, e.g. “Type to filter”.

**filterText** If the instant filtering is disabled, then the filter button is displayed with the text of this input property.

**filterTerm** The filter can be preset to the filter term.

**filter** If the user filters, then the filtered records are emitted by this output property.

## Table Component

The interface of the table component is mostly just a composition of its direct children. It has the following input properties:

**rows** Input data to be displayed in the table rows. It is an array of objects representing rows. Each row object can contain a nested object or an array of objects with simple data types. In other words, two levels of nesting are allowed. This is the only required input. Other properties can be automatically detected based on it.

**columnsConfig** Input property for a configuration of table columns. It allows to override the values set by default. Each column configuration object contains various properties:

**id** for a unique identifier of a column.

**text** for a user-readable column name to be displayed in a table header.

**sortingDisabled** Since sorting is enabled for all columns by default, this property disables the sorting for a specific column.

**formatters** There are countless possibilities for formatting a cell value since it can represent various things like date, time, etc. The formatters is a list of transformations performed on a cell value to display it in a specific format. It is also useful for compound properties, e.g. concatenating the first and last name. Formatter interface matches an interface of an Angular pipe, which is not a coincidence. It is to allow the usage of pipes of the Angular standard library or other custom pipes.

**subFields** Each subfield of a column can be configured similarly as a column:

**id** for a unique identifier of a subfield.

**text** for a user-readable subfield name displayed in a column context menu.

**isVisible** All subfields are initially visible, but it is possible to specify which subfields should be hidden.

**formatters** A list of formatters as in a column configuration.

**visibleColumns** Identifiers of initially visible columns in a table.

**reorderingEnabled** An input property for enabling/disabling drag&drop reordering of columns. The default value is true.

**changeColumnVisibility** An input property that enables/disables user interface for changing column visibility. The default value is true.

**rowsSortingMode** Table rows are sorted client-side in a default sorting mode. The external mode is intended for server-side sorting. Lastly, the sorting of rows can be disabled completely, i.e. no sorting icons.

**initialSortColumn** One column can be set to be sorted when a page is loaded. An optional plus or minus sign specifies the sort direction.

Apart from the previous input properties, there are some advanced options for the further customization. The developer can provide a custom template to be rendered at a specific part of the table.

Custom templates are a very low-level interface where the developer has the full power of the framework. The developer can leverage the subcomponents and has programmatic access to the component interface. Apart from the subcomponents, the library offers additional utilities that help to join the individual pieces together.

For example, the developer may want to add statistics to a table footer. However, there is no footer component in the library because it is usually application-specific. The footer must be specified in the table to form properly aligned cells, which is not possible without the modifications of the library. If there is no way to specify that, the developer would have to replace the whole table.

The developer should be able to override only a part that needs a modification. It is an overkill to customize a whole table template if only one data cell needs an adjustment. Various templates offer different granularity of customization:

**tableTemplate** It specifies a custom template for the whole table.

**headerTemplate** It specifies a custom template for the table header.

**bodyTemplate** It specifies a custom template for the table body.

**footerTemplate** It specifies a custom template for the table footer.

**bodyRowTemplate** It specifies a custom template that is rendered for each body row. It can access the row variable.

**headerRowTemplate** It specifies a custom template that is rendered for the table header row. It can access the column id in a variable.

Lastly, there are many events happening at the table. Output properties notify about them with all details so the developer can react accordingly:

**addColumn** A column was added by the user. The event data contain the column id.

**removeColumn** A column was removed by the user. Again, the event data contain the column id.

**sortColumn** A column was sorted by the user. The event data contain the column id, direction, and the reference to a column state.

**addingColumn** A column is being added at a specific position. The event data contain the position where a column should be added.

**toggleSubfield** This event is triggered when the visibility for a subfield is changed. The event data contain the column id, the subfield id, and currently visible subfields of the column where the subfield is toggled.

**visibleColumnsChange** This event is triggered whenever a column is added/removed or the order of columns changed. A list of column ids is provided as the event data.

**rowClick** Body row was clicked. The event data contain both row and column information i.e. the column id, the row index, the column index, and the row object.

### Table Header Component

All properties of the table header component are already described for the table component. However, this component has only a subset of them:

- `rows`;
- `columnsConfig`;
- `visibleColumns`;
- `reorderingEnabled`;
- `changeColumnVisibility`;
- `addingColumnIndex`;
- `rowsSortingMode`;
- `initialSortColumn`;
- `headerRowTemplate`;
- `addingColumn`;
- `addColumn`;
- `removeColumn`;
- `sortColumn`;
- `sortColumnInit`;
- `toggleSubfield`, and
- `visibleColumnsChange`.

### Table Body Component

As the table header component, only a subset of properties of the table component is necessary for the table body component:

- `rows`;
- `columnsConfig`;
- `visibleColumns`;
- `changeColumnVisibility`;
- `addingColumnIndex`;
- `bodyRowTemplate`, and
- `rowClick`.

### Table Header Cell Component

The table header cell component has an additional required input `column` for a column state, but it also uses some of the properties of the table component:

- `columnsConfig`;

- `visibleColumns`;
- `changeVisibility`, and
- `rowsSortingMode`.

### Add Column Component

The purpose of this component is to display a user interface for adding columns. It uses a select component internally, which needs to be integrated into a table. This component acts as an adapter. It has one input property from the table component and other properties of the select component:

- `visibleColumns`;
- `open`;
- `selected`, and
- `close`.

### Data Cell Component

This component displays the data of various types. Simple data types are directly displayed, but complex data types are delegated to a corresponding component. A data cell component has two required input properties:

**row** Despite that only one property of a row is usually displayed, an access to a whole row object is advantageous in some cases, e.g. compound properties.

**column** It is a column state, which stores column id, currently visible fields, etc.

### Object Cell Component

This component displays values of an object according to subfields configuration. It extends the table data cell with additional properties:

**row** is the row object.

**column** is the column state.

**object** Values of the input row are displayed beneath each other inside the cell. Either the row or the object must be specified.

**hasPrefix** Displayed values can be prefixed with a corresponding subfield text. The default value is false.

### Array Cell Component

This component uses the object cell type to display object array items. It extends the table data cell with additional properties:

**subrows** It is an array of items to be displayed.

**column** It is a column state.

**showAll** Depending on this input, either just one or all items are shown.

**arrayItemTemplate** It specifies a custom template to be rendered for each array item.

# Chapter 5

## Implementation

At the beginning, the reasoning for the implementation language is given. Before diving into the actual implementation of the library, the project setup and its folder structure are explained. In the implementation, we are focused especially on the core module. Last but not least, the performance optimizations are outlined.

### 5.1 Implementation Language

Not so long ago, the question of implementation language would have not even arisen since JavaScript was the only choice. Nowadays, there are alternative languages for web development; even JavaScript comes in different flavors known as ECMAScript.

New ECMAScript 6 features definitely boost developers' productivity and reduce the clutter. making it clearer. TypeScript has a great support for ECMAScript 6 with additional benefits that are significant for the library:

#### **Type safety**

The TypeScript compiler can reveal a whole range of issues at the compile time. Usually, the earlier an issue is discovered, the faster it is to fix it. Many editors can point out problematic code while typing it.

Developers are more likely to misuse the library since they are less familiar with its code than their own. Additionally, the debugging is also harder. It is simply better if such error can be detected during the compilation, especially for a library.

Let us illustrate the importance of types, a developer mistyped a name of a property on a configuration object. The configuration is then passed to the library written in pure JavaScript.

Usually, libraries check for invalid inputs to warn about the incorrect usage. A warning message indicates the problem and ideally suggests a solution. Unfortunately, the wrong property is silently ignored in this case. A complex configuration object requires many checks. The implementation of these checks is reminiscent of ad-hoc type checking. It is error-prone and carries a run-time overhead. This job is better suited for full-featured type system of TypeScript. It can reliably perform all checks with no run-time overhead.

#### **Tooling**

Since TypeScript has been around for a while, its tooling is very mature, i.e. an

excellent editor support. Any popular editor has various extensions a.k.a. plugins for TypeScript. Developers can benefit from quick refactoring, navigation, auto-corrections and much more.

If we revisit the previous example, a mistake of wrong property name can not only be detected; it could be prevented in the first place if the developer would have auto-completed the property name. Additionally, the developer can see the type information and the documentation of the configuration object directly in the editor.

### Interoperability

The library exists in a large ecosystem. In an application, it has to coexist and integrate with other libraries, often implemented in JavaScript. TypeScript is designed for perfect compatibility with JavaScript, unlike other languages that compile to JavaScript, e.g. [Dart](https://www.dartlang.org/)<sup>1</sup>. Additionally, the Angular framework itself is developed in TypeScript.

Lastly, let us draw a parallel between these advantages and the usability heuristics (section 3.1). The type checking means the “error prevention” and the tooling provides “Help and documentation”. In that sense, these heuristics are applicable to the library as a result of the tools that interact with the library. To summarize this section, there is no doubt that TypeScript is a great choice for the library.

## 5.2 Project Setup & Tools

The setup of the library project was bootstrapped with the Angular CLI (command line interface). It predefines many commands useful for the development such as running tests. Despite that the Angular CLI is primarily intended for applications, it can be adjusted for libraries as well. It has a separate file dedicated for its configuration. Similarly, TypeScript compiler has a dedicated file with many configuration options.

TypeScript compiler is not very strict by default. Thus, it was configured for the library to perform additional checks by enabling some options, e.g. `noImplicitAny` and `strictNullChecks`. The first option states that the compiler throws an error if a type can not be implied from the context. Normally, it would assume the most general type, which is too permissive. The second option forbids to access a possibly null object without a check. Accessing a null object is one of the most common mistakes.

A linter performs additional checks for so-called best practices, which are concerned with various aspects of the source code such as naming convention and style. A linter defines a set of rules and ensures that they are consistently followed. The best practices were introduced for a reason. For example, a rule that if statement must always be braced prevents an error that somebody accidentally adds another statement in the assumption that it will be executed only if the condition holds true.

The library follows the official style guide of the Angular framework. It uses a linter extension that automatically checks whether the style guide rules are followed.

Another form of checks are automated tests. Testing is even more important for the library since many applications may rely on it. The tests should be executed on each commit to a version control system in order to detect a regression. To achieve that, we are using a continuous integration service called [Travis](https://travis-ci.org/)<sup>2</sup>, which is used extensively by open-source

---

<sup>1</sup><https://www.dartlang.org/>

<sup>2</sup><https://travis-ci.org/>



projects. So the tests are executed as a step of the continuous integration process. Other steps are linting and creating a build of the library.

## 5.3 Folder Structure

The source code of the library is located in the `lib` directory. Inside this directory, there is a subdirectory for each module. Apart from the modules from section 4.6, there are a few implementation specific modules `sortable` and `pipes`, which contain utilities. The hierarchy of components of the table module is reflected in the nesting of directories.

The directory structure also shows different types of files for some directories. Overall, there are three types of files based on the extension:

- \***.ts** for a TypeScript file;
- \***.html** for an HTML template, and
- \***.css** for styles.

Additionally, the TypeScript files can contain various content. For each content, there is a corresponding file with tests. To distinguish them at the first glance, they have a suffix according to the Angular style guide:

- \***.spec.ts** with tests;
- \***.module.ts** with a module definition;
- \***.component.ts** with a component class;
- \***.directive.ts** with a directive class, and
- \***.pipe** with a pipe class.

Another directory of the source code is `app/` intended for the demo application. Then, there is `assets` directory with static files. Lastly, `environments` folder contains configuration files for either development or production environment.

## 5.4 Core Module

The table module is the core of this library. It is also the most interesting part from the implementation perspective since it is comprised of many components that need to cooperate together.

Figure 5.3 displays a detailed class diagram for the table module. The relationships among the components are based on the actual properties or the inheritance. In the view, it is still a component tree as shown in diagram 4.3 from the design.

The component tree does not match the inheritance tree despite that it is desired to share properties. For instance, the input property `rows` of a `TableComponent` should be accessible to the lower component `TheadComponent`. However, it is not just a matter of sharing properties or methods.

The component inheritance in the Angular framework<sup>3</sup> also inherits its metadata. That means that all inputs and outputs are also inherited, which is not desired in this case. There is no need for `rows` input property in `TdComponent` because it works only with one row.

---

<sup>3</sup>The component inheritance was introduced in the Angular version 2.3.

```

lib/
|-- dropdown-select/
    |-- dropdown-select.module.ts
    +-- dropdown-select.component.{html,css,ts,spec.ts}
|-- filter/
    |-- filter.module.ts
    |-- filter.service.{ts,spec.ts}
    +-- filter.component.{html,css,ts,spec.ts}
|-- pagination/
    |-- pagination.module.ts
    +-- pagination.component.{html,css,ts,spec.ts}
|-- pipes/
    |-- default-value/
        |-- default-value.pipe.{ts,spec.ts}
    +-- pipes.module.ts
|-- sortable/
    |-- sortable-item.directive.{ts,spec.ts}
    +-- sortable.module.ts
+-- table/
    |-- thead/
        |-- th/
        +-- add-column/
    |-- tbody/
        +-- td/
            |-- array-cell/
            +-- object-cell/
    |--table.module.ts
    +-- ... # table services and types
app/
assets/
environments/

```

Figure 5.1: The folder structure of the source code. It reflects the hierarchy of components and the decomposition to different modules.

Inheritance is meaningful only for data cell types, so `TdComponent` is a parent of `ObjectCellComponent` and `ArrayCellComponent` because they are a specific version of a general data cell. Their interface should be equivalent. However, `TdComponent` is not a more specific version of `TbodyComponent`. These are two different things.

Alternative means of communication are services, which give us more fine-grained control. A service can be shared in a certain subtree of the component tree or a whole application. Shared properties are stored in the service `TableStateService`. The components then define getters to access the service properties, and setters to change them.

Keeping the state in one service also allows to easily initialize its subcomponents. This is automatically performed according to certain conventions. A subcomponent firstly checks whether it is used under a table to initialize itself. As a result, there is no need to duplicate the same input properties even if it is a required input.

Associations are shown in the diagram only for components that share input attributes. A subcomponent can be used independently of the table component. In that case, it initializes a new `TableStateService` for itself and potential children subcomponents.

Diagram 5.2 shows the details of the table services. According to the Redux principles (section 2.4), it models the state as one object `TableStateService`, defines events for user actions, and the state is updated in `TableReducerService` after dispatching of an event. However, the principles are followed only loosely in some cases.

One exception is that initialization directly modifies the state instead of triggering an event. The properties of the state table are not read-only so that the table components can directly assign the user inputs. It would be impractical to trigger an event for the assignment of each input. The overhead associated with that would make the code less readable.

Other state changes are performed as a result of user actions, e.g. sorting. These actions require more complex modifications coordinated by the service `TableReducerService`. All user actions properly trigger an event. Each user event is handled by `reduce` method that accepts the current state and an event corresponding to a user action. As a result, the current table state is updated.

For some functionality, there is a dedicated service for certain changes such as sorting. As specified in the design, services can be replaced with an alternative implementation if it has the same public interface. So a dedicated sorting service can be replaced individually, e.g. a different sorting algorithm. The same applies to `TableInitService` that is responsible for detecting column configuration from the data. The detection does not work reliably if a property contains values of different types. Since the developers can always configure columns themselves, this feature is mainly for developer convenience. Yet, it could be improved with a use of `JSON Schema`<sup>4</sup> detector, which reliably detects an object structure. On the other side, it would add a significant overhead to the core module that should be as lean as possible.

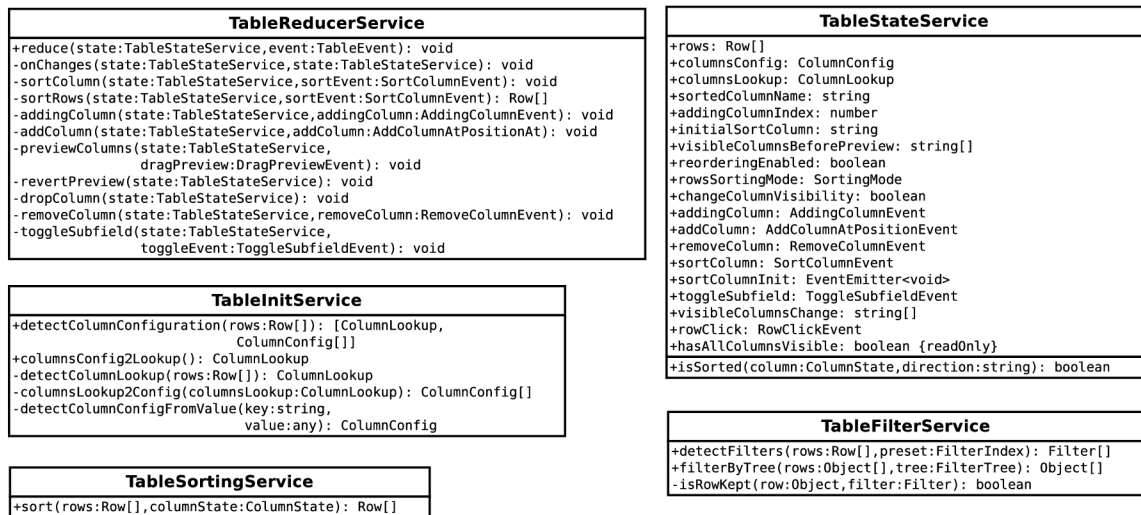


Figure 5.2: The service layer of the core table module.

<sup>4</sup><http://json-schema.org/>

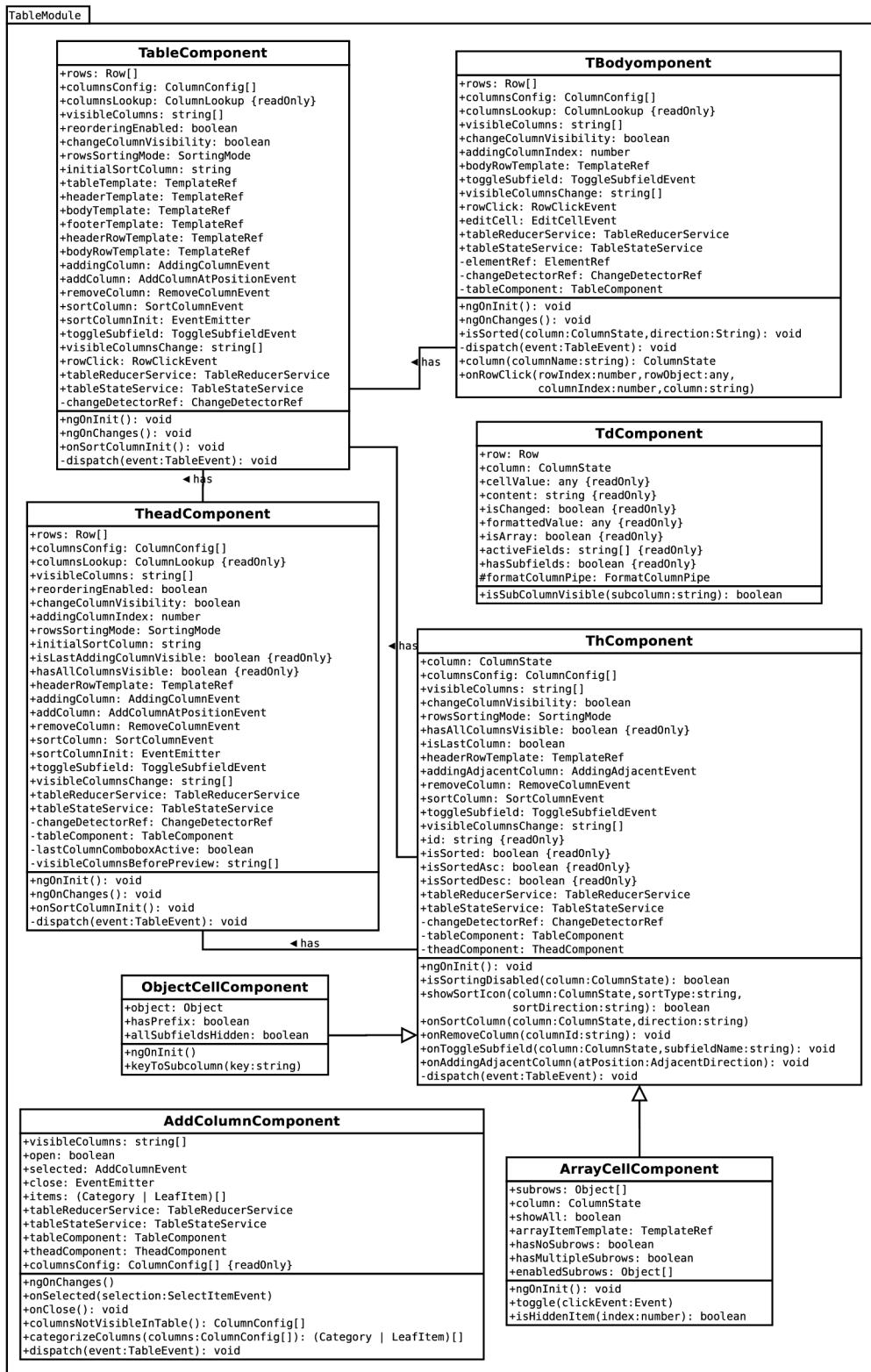


Figure 5.3: The core module of the library.

Records	Rendering before AOT [s]	Rendering after AOT [s]	Difference [%]
10	2.07	1.89	8.7
100	2.92	2.54	13.01
1000	9.12	8.73	4.27

Table 5.1: The AOT compilation improved the rendering time.

## 5.5 Performance Optimizations

The performance is an important factor when dealing with data-intensive applications although user interface usually displays only a part of the data, e.g. thanks to pagination. The Angular framework by itself is very fast, but there were two additional performance optimizations:

1. The first performance optimization is the ahead-of-time (AOT) compilation of the library. It basically means that the library is able to run without the overhead of the framework. This can be achieved by pre-compiling the source code into pure JavaScript.

Generally speaking, frameworks are good for writing the code, but not so good for running it. Firstly, a framework increases the size of an application; it may even be larger than an application itself. More importantly, it adds a run-time overhead associated with interpreting its abstractions. In the Angular framework, components with its extended template syntax have to be processed before they can be rendered a.k.a. just-in-time compilation.

Configuring the AOT compilation for the library requires eliminating certain code constructs, which are not statically analyzable. Despite that, it is still possible to perform dynamic behavior since it is still JavaScript, a very dynamic language, at the end of the compilation.

As a result of static compilation, not only the rendering is faster but also security improved. The static compilation became a part of the continuous integration process to ensure that new commits are statically analyzable.

The impact of this optimization was measured by rendering the table component with a varying number of automatically generated records. Each record contains nested objects, e.g. studies of a random count. The records are generated synchronously to measure the initial page rendering time for all of them. In a typical application, the results would be loaded from a server without blocking.

Table 5.1 shows the average of 5 measurements before and after AOT compilation for each size of records. The rendering time decreased for any count of records.

2. One of the most time-consuming tasks executed by the Angular framework is change detection. It keeps the user interface and underlying data in synchronization. The default detection strategy is performed frequently and assumes that everything can change.

We activated a faster push strategy; thereby, took the change detection into our hands. With this strategy, it is our responsibility to make sure that the changes are correctly detected.

In the library, all events that can cause changes are coordinated by one service, i.e. `TableReducerService`. Thus, this service knows when certain pieces of code can change and ensures that a change detection is triggered accordingly.

One caveat is that a change can be triggered externally in an application. These changes can not be reliably detected. That could cause the confusion if the developer is not familiar with this behavior. In the worst case, it may lead to a bug in the application. Therefore, this optimization is disabled by default. It is meaningful to opt-in in many situations, e.g. if a developer uses immutable operations or data structures.

To test this optimization, we measured the percentage of time devoted to the change detection after performing certain actions, i.e. sorting and reordering a column. The percentage is around 68% for the default strategy in the sample application that renders the table component with 100 records. It is fairly constant for a different count of records.

The percentage decreased by 6.04% after activating the push strategy. This has a significant effect on the responsiveness of an application because if the change detection is running, an application is blocked.



# Chapter 6

## Demonstration

The main goal of this chapter is to demonstrate the characteristics of the library as they are defined in the design specification. It does not try to list all the possible use cases. It only illustrates the various aspects of the library on a few simple examples. More advanced examples of the library usage can be found in the demo application.

The first section starts with a usage of the main table component in a sample application. This component is then adjusted to demonstrate the custom templates. Lastly, we present one example of a utility.

### 6.1 Basic Example

The library is used declaratively with special tags and attributes. The convention is to prefix them to avoid collision with the default HTML tags or other libraries. If something is defined by an application, it typically has a prefix “app”.

Listing 6.1 demonstrates the usage of the main component `<iw-table>` by passing `students` from the sample application `AppComponent` as explained in section 2.3, more specifically Listing 2.1.

For the demonstration purpose, the students are hard-coded in the application. Normally, they would be fetched from a server. Other inputs of the table are automatically detected from the data according to convention over configuration.

Except for the setup necessary for the application, the actual usage of the library is just one line of declarative code in the component’s template. The result is shown in Figure 6.1. The user gets a full-featured table with sorting, drag&drop reordering, dynamically configurable columns, collapsed study items, etc. It can be adjusted with additional attributes, e.g. `reorderingEnabled`. Without this high-level abstraction, it would require hundreds of lines in the application to implement this functionality.

The demo application contains another example where several input attributes are changed, and all output events are printed to the console.

```
@Component ({
  selector: 'app-component',
  template: `
    <h1>Data Table</h1>
    <iw-table [rows]="students"></iw-table>
  `
})
```

```

class AppComponent {
  students = [{
    lastName: 'Zieme', birthday: '1985-10-8', // email
    address: {
      country: 'Lao People\'s Democratic Republic',
      city: 'North Madelynnhaven',
      street: '934 Daniela Crescent'
    },
    studies: [{
      finished: true, degree: 'Master', // university, faculty
    }],
  },
  // other students
  ];
}

```

Listing 6.1: The basic usage of the library in the sample application.

## 6.2 Custom Templates

Custom templates are useful in various situations, e.g. styling. Listing 6.2 assumes the same component class `AppComponent` as in the previous code listing with a different HTML template. The table look is altered by specifying styles from the Bootstrap framework, i.e. `table table-stripped`. The library does not depend on any specific CSS framework, but it does not exclude them. A framework often requires the style to be applied directly to an element, which is not possible if they are hidden under the hood of a component. Luckily, a custom template `ng-template` gives developers control over adding classes to the elements. We can even completely disable the styles of the table component by omitting the class `iw-table`.

Note that the subcomponents `iw-thead` and `iw-tbody` are used without specifying their required attributes since their state is inherited from the parent table. Once again, convention over configuration in action. This code snippet actually illustrates all characteristics from 4.5. Two most important ones are “Levels of abstraction” and “Fallback”.

Custom templates are effectively the fallback solution. Developers get the full power of the framework while leveraging the components of the lower abstraction, utilities, and the public API of components, i.e. the access to properties or calling methods.

For example, an application may suddenly require an uncommon interface for editing that is not supported by the library. The body cells are supposed to be edited inside a dropdown menu. In this case, we can just replace `iw-tbody` with application-specific implementation `app-tbody`. It can be implemented by inheriting from `iw-tbody`. The developer can also use other existing libraries for editing. That is possible in any custom template. Most importantly, an application still benefits from features of `iw-thead`, i.e. sorting and reordering.

The demo application has an additional example for customizing one specific cell, i.e. studies. It display them in the dropdown menu.

```

<iw-table [rows]="students" [tableTemplate]="tt">
  <ng-template #tt>

```

```

<table class="iw-table table table-striped">
  <thead iw-thead></thead>
  <tbody iw-tbody></tbody>
</table>
</ng-template>
</iw-table>

```

Listing 6.2: The table style is altered in the custom table template.

Data Table

LASTNAME	EMAIL	BIRTHDAY	Add a column	ADDRESS	STUDIES
Zieme	Alvis.Sokes26@gmail.com	Oct 8, 1985	Type to search id salutation phone	934 Daniels Crescen North Madselyrhev Lao People's Democ	University of Response Faculty of Interactions Master true
Wolf	Luz_jacobi@gmail.com	Sep 22, 1985		27369 Easton Bypas West Andersensbury Nigeria	University of Group Faculty of Solutions Bachelor false ▼ 1 more
Weissnet	Jovan.Vienow@hotmail.com	Oct 25, 1985		2414 Lyla Crossroad Port Laronview Trinidad and Tobago	University of Metrics Faculty of Functionality Doctor false ▼ 2 more
Slamm	Vivian_Boyer@hotmail.com	Nov 6, 1985		160 Sawayi Station Lebsackborough Lao People's Democratic Republic	University of Program Faculty of Marketing Master true ▼ 1 more
Schroefer	Deonte85@hotmail.com	Sep 11, 1985		5249 Allie Clens Port Peyton Timor Leste	University of Communications Faculty of Usability Master true ▼ 1 more
Setterfield	Kris.Gerlach93@hotmail.com	Mar 13, 1979		108 Ambrose Prairie Siddroville Switzerland	University of Infrastructure Faculty of Mobility Bachelor true ▼ 3 more
Runofsdottir	Amely64@hotmail.com	Jun 28, 1985		84256 Meggie Ways Hannalibary Republic of Korea	University of Infrastructure Faculty of Implementation Doctor true ▼ 3 more
Quigley	Dayna82@hotmail.com	Jan 2, 1979		24677 Winslen Lights Turnershire Sudan	University of Markets Faculty of Mobility Master

Figure 6.1: The final user interface of the table component.

## 6.3 Utilities

The utilities help to put various pieces together, which is handy especially in custom templates. One utility is demonstrated in Listing 6.3. Applying the directive `iwSortableItem` to cell elements allows users to change their position by dragging and dropping them.

All presented examples might seem simple, but the true power of the library is in its ability to put these concepts together. For example, the sorting utility can also be applied to the rows or even the rows of several tables that can exchange their rows.

```

<table>
  <tr>
    <td iwSortableItem>Cell A</td>
    <td iwSortableItem>Cell B</td>
  </tr>
</table>

```

Listing 6.3: The sortable utility adds reordering functionality to the table cells.

## Chapter 7

# Usability Testing & Evaluation

The previous sections describe the current state of the library, which is a result of an iterative process. The first section goes through the history of the project to describe how it was shaped. Then, the library is evaluated based on the feedback of users. At the end of the chapter, the statistics for the library are presented.

### 7.1 Iterations

The user interface was created in iterations. Each iteration ended with usability tests according to the testing protocol from [6] in order to find out what works and what does not.

In the first phase, a prototype was created, which mostly lacks the underlying functionality, e.g. hard-coded output data. Nevertheless, it verifies some of the basic assumptions made during the design without much development effort.

In the testing session, users get a list of tasks to finish:

- Find the youngest person.
- What is his/her address?
- The columns are quite jammed. Can you somehow reduce the information?
- An e-mail column should be at the beginning.

For the first task, it is expected that the users discover the UI for sorting and how it works by themselves. The second task requires adding an address column. Other features are tested by the remaining tasks.

The ideal number of users for usability tests is no more than 5 according to [11]. We chose for 4 testing users. All of them were able to finish their tasks although they were confused sometimes. One user was afraid to remove a column because it might delete the data. To fix this usability issue, UI should use better wording that implies no mutation “Hide this column” instead of “Remove this column”. Even small adjustments can be of great value.

So far, we only mentioned the testing of UI, namely user experience. We also care about good developer experience while working with the library. This is actually even more important since we are primarily concerned with the design of a library. An application can be customized in many ways by the library.

We employ the same testing method, except that the users are developers who have to work with the programmatic interface in order to finish their tasks. Testing users usually get everything prepared, but we would like to see whether they are able to integrate the library in a different environment. So the first task was to install the library and use the table component with sample data.

The first two testing sessions discovered that the installation is overly complicated mainly because of dependencies. The library required a JavaScript component [Select2](#)<sup>1</sup>, which in turn required [jQuery](#)<sup>2</sup>. Additionally, the styles for Select2 and Bootstrap were necessary. There was a mismatch between dependencies in one testing session, so the library could not work properly. This caused that one user was not able to finish his tasks.

Since interoperability with the libraries that are from outside the Angular world is problematic, Select2 and jQuery were removed. There were additional benefits in terms of features and smaller library size. Bootstrap also became optional. Thus, the installation process was simplified to one command to add the library package.

Afterward, no significant issues appeared in two additional testing sessions. Users finished their tasks successfully.

## 7.2 User Feedback

This section focuses on developers as target users of the library. User feedback was gathered throughout the project life-cycle, mainly in the form of discussions. Nevertheless, they were still influential, i.e. contributed to new ideas.

In the final phase, we conducted a survey to evaluate the library. The questions were chosen to address various aspects according to USE (usability, satisfaction, ease of use) questionnaire [7]:

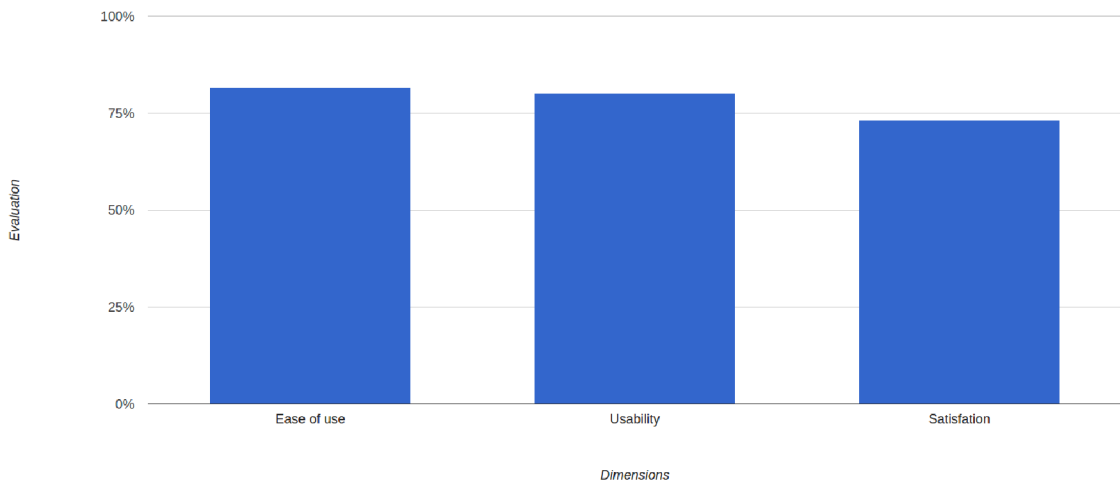


Figure 7.1: The results of a survey showing three dimensions evaluated by users.

- How easy is it to get started with the library?
- How applicable is the library for data-intensive applications?

---

<sup>1</sup><https://select2.github.io/>

<sup>2</sup><https://jquery.com/>



- How do you feel about the library overall?
- Do you have any other feedback?

Users can choose a value on the scale from 1 to 10 for all questions, except the last one. Figure 7.1 displays the summary of results for 6 submitted responses. Especially the ease of use was rated very high with 81.66%, but other aspects are not far behind.

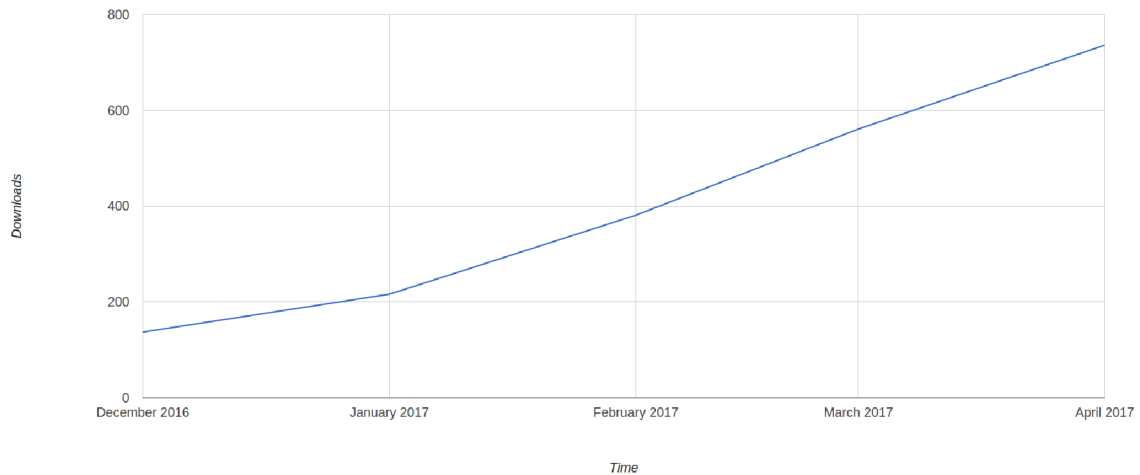


Figure 7.2: The number of downloads for the library grows linearly every month. The download statistics are publicly available by NPM.

The survey contained one open question for writing any kind of feedback:

- *“The table looks visually nice. It is also functional.”*
- *“The first time to see the select component in the Angular way.”*
- *“It’s sometimes a battle to tweak the library to do what you want. It may even be faster to write it from scratch in some cases. Faster means cheaper for companies. Maybe that’s a part of the reason why many companies have their internal libraries for things like a table. Own implementation is easy to modify for their needs. I can imagine that a company would use this library at the beginning. Then as they need to adjust more things, they would gradually replace certain templates until it is replaced completely.”*
- *“Nice library!”*

The final words of feedback belong to Kevin Merckx who is not only an Angular expert but also a product owner of an application in interfacewerk GmbH that uses the library extensively:

*“The library is very flexible. It was able to adapt nicely to various situations.”*

### 7.3 Statistics

Firstly, the download statics show a positive interest in a library despite that there was almost no propagation of the library, except for publishing it on the development platform

“[Github](#)<sup>3</sup>” and the registry of JavaScript packages called “[NPM](#)<sup>4</sup>”. Figure 7.2 shows a number of downloads from NPM over a five-month period. The number of downloads per months is steadily increasing. The total number of downloads is 2031.

Lastly, we would like to mention other library statistics. Overall, there are 4 library modules and 1 module for the demo application. In these modules reside 13 components, 2 directives, 2 pipes, and 8 services. These were created in 166 commits. The first commit performed by Angular CLI generates 571 lines. Other commits contain 23,379 additions and 12,126 deletions.

---

<sup>3</sup>[github.com](https://github.com)

<sup>4</sup><https://www.npmjs.com/>

## Chapter 8

# Conclusion

In the first chapter, we gave an introduction to client-side web technologies, especially these closely related to Angular. Then, we dived into the Angular framework itself. We described its various aspects and principles for the implementation of components. The creation of UI also involves the design and testing which was covered in the second chapter. Afterward, the existing component libraries were analyzed, and the detailed specification was created. It states a set of key characteristics for the library. Then, we designed the architecture and the public interface of components. The implementation was explained with the focus on the library core. The demonstrative examples picture the library in accordance with the characteristics stated in the design specification. The usability tests examined the user interface and the programmatic interface, which led to several improvements. The survey was conducted to evaluate the usability, satisfaction, and ease of the use. The results are satisfactory. The download statistics also indicate positive reactions.

The library and demo application are published on [Github](https://github.com/zorec/ng2-pack)<sup>1</sup> under MIT license. The library offers a collection of related components and utilities for data-intensive applications. Its user interface is accustomed for this use case with certain UI elements such as collapsed list of nested entities. Its programmatic interface contains patterns and conventions relevant for designing other third-party libraries.

Some possibilities for the future work were already discussed at the end of implementation section 5.4. For example, an alternative service for the initialization of the table would detect column configuration more reliably based on the JSON schema. Both the existing solution and the alternative have their pros and cons. Ideally, the developer can choose their preferred implementation.

Naturally, it is always possible to add new components and utilities, e.g. for exporting. This paper is primarily concerned with the core of the library, which proved itself in real-world scenarios. Although it is rather minimalistic, it can be easily adjusted and extended.

---

<sup>1</sup><https://github.com/zorec/ng2-pack>

# Bibliography

- [1] Cascading Style Sheets. [online; visited 2017-05-21]. Retrieved from: <https://www.w3.org/Style/CSS/>
- [2] HTML 5.2. W3C. [online; visited 2017-05-21]. Retrieved from: <https://w3c.github.io/html/>
- [3] *One framework. - Angular.* [online; visited 2016-12-28]. Retrieved from: <https://angular.io/>
- [4] Standard ECMA-262. [online; visited 2017-05-21]. Retrieved from: <http://www.ecma-international.org/publications/standards/Ecma-262.htm>
- [5] Fox, P.: WDCNZ: The Developer Experience [online; visited 2017-05-16]. <http://blog.pamelafox.org/2011/08/wdcnz-developer-experience.html>. 2011.
- [6] Krug, S.: *Don't Make Me Think: A Common Sense Approach to the Web (2Nd Edition)*. Thousand Oaks, CA, USA: New Riders Publishing. 2005. ISBN 0321344758.
- [7] Lund, A.: Measuring Usability with the USE Questionnaire [online; visited 2017-05-15]. 2001. Retrieved from: [https://www.researchgate.net/publication/230786746\\_Measuring\\_usability\\_with\\_the\\_USE\\_questionnaire](https://www.researchgate.net/publication/230786746_Measuring_usability_with_the_USE_questionnaire)
- [8] Mathis, L.: *Designed for Use*. The Pragmatic Programmers LLC. 2011. ISBN 13 978-1-93435-675-3.
- [9] Monteiro, F.: *Learning Single-page Web Application Development*. Packt Publishing. 2014. ISBN 978-1-78355-209-2.
- [10] Murray, N.; Lerner, A.; Coury, F.; et al.: *ng-book 2: The Complete Book on Angular 2 (Volume 2)*. Fullstack.io. 2016. ISBN 0991344618.
- [11] Nielsen, J.: Why You Only Need to Test with 5 Users [online; visited 2014-04-27]. 2000-03-19. Retrieved from: <http://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/>
- [12] Preece, J.: *Human-Computer Interaction*. Addison-Wesley. 1994. ISBN 0-201-62769-8.
- [13] Scott, B.; Neil, T.: *Designing Web Interfaces*. O'Reilly Media. 2009. ISBN 978-0-596-51625-3.