VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Kim Schmiedehausen

SINGLE PAGE APPLICATION ARCHITECHTURE WITH ANGULAR

Technology and Communication
2018

ACKNOWLEDGEMENTS

I would like to thank Mr Ville Salomäki and Wapice Ltd. for giving me the opportunity to pick the thesis subject based on my interests and in turn guiding me through the process. I would also like to express my gratitude to Dr. Ghodrat Moghadampour for supervising this thesis.

Special thanks to all the friends, family and colleagues who participated in the work.

VAASAN AMMATTIKORKEAKOULU
Ohjelmistotekniikka

TIIVISTELMÄ

Tekijä	Kim Schmiedehausen
Opinnäytetyön nimi	Single Page Application Architecture with Angular
Vuosi	2018
Kieli	englanti
Sivumäärä	48
Ohjaaja	Ghodrat Moghadampour

Opinnäytetyö toteutettiin tutkimuksena Wapicelle. Työn tarkoituksena oli tutkia keinoja, joilla Angular-ohjelmistokehyksellä toteutettujen ohjelmien rakennetta voidaan parantaa tukemaan projektien muuttuvia vaatimuksia.

Työssä tutkittiin SOLID-periaatteita, joita soveltamalla ohjelmistoista saadaan joustavampia ja ylläpidettävämpiä. SOLID-lyhenne on muistisääntö viidelle suunnitteluperiaatteelle: yhden vastuualueen periaate, avoin/suljettu periaate, Liskovin korvattavuusperiaate, rajapintojen erotteluperiaate ja riippuvuuden kääntöperiaate. SOLID-periaatteita sovellettiin Angular-ohjelmistokehykselle ja TypeScript-ohjelmointikielelle sopivaksi. Angular-sovelluksien rakenteelle etsittiin ratkaisua hyödyntämällä Angular-moduuleita ja komponenttien vastuiden jakamiseen sovellettiin ”fiksuja” ja ”tyhmiä” komponentteja.

Opinnäytetyön tuloksena syntyi yleispätevä tutkimus Angular-ohjelmistokehyksen arkkitehtuurista ja jaottelusta. Tutkimuksen avulla voidaan kehittää Single Page-sovelluksia uusiokäyttämällä vanhoja komponentteja ja parantaa sovelluksen rakennetta ja suorituskykyä jakamalla toiminnallisuutta moduuleihin. Työssä esitelty projekti tarjoaa hyvän esimerkkirakenteen Angular-sovelluksille.

SOLID-periaatteita on mahdollista hyödyntää myös muissa olio-ohjelmointikielissä ja ”fiksu/tyhmä”-komponenttimallia voidaan soveltaa myös muissa komponenttipohjaista arkkitehtuuria tukevissa ohjelmistokehyksissä tai kirjastoissa.

Avainsanat Angular, Single Page-sovellus, SOLID-periaatteet

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Ohjelmistotekniikka

ABSTRACT

Author	Kim Schmiedehausen
Title	Single Page Application Architecture with Angular
Year	2018
Language	English
Pages	48
Name of Supervisor	Ghodrat Moghadampour

This thesis was carried out as a research project for Wapice. The objective of this thesis was to find ways to improve the structure in Angular-applications to become more adaptive to changing requirements.

The thesis examined SOLID-principles, which can be applied to make software more maintainable and adaptive to change. SOLID is a mnemonic acronym for five design principles: Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle and Dependency inversion principle. The work applied SOLID-principles in the Angular-framework with the TypeScript-programming language. The structure for Angular-applications was researched by utilizing Angular-modules and the “smart/dumb”-model was explained to make components reusable inside the application.

As a result, a general research on the architecture and structure of Angular-applications was created. The work can be used as a base for developing Single Page Applications with Angular by reusing components and to improve the structure and performance of the application by dividing functionality into modules. The project included in the thesis provides a good example structure for Angular-projects.

SOLID-principles can also be applied in other object-oriented languages and “smart/dumb”-model can be used in other frameworks or libraries that support component-based architecture.

Keywords Angular, Single Page Application, SOLID-Principles

LIST OF ABBREVIATIONS

SPA	Single Page Application
JS	JavaScript
TS	TypeScript
ES6	ECMAScript 6
HTML	Hypertext Markup Language
DOM	Document Object Model
API	Application Programming Interface
CSS	Cascading Style Sheets
SRP	Single Responsibility Principle
OCP	Open/Closed Principle
LSP	Liskov Substitution Principle
ISP	Interface Segregation Principle
DIP	Dependency Inversion Principle
DI	Dependency Injection

CONTENTS

LIST OF FIGURES AND CODE SNIPPETS

1 INTRODUCTION

Project requirements change constantly, especially when more and more software is being developed with Agile-methodologies. Even when using a framework, careful design choices need to be made to adapt to the changing requirements. The goal of this thesis is to find ways to improve the architecture and design in Angular-projects in general and to avoid common pitfalls when developing applications with it.

Angular is a frontend web application platform designed to build single page applications. It is a complete rewrite from its predecessor AngularJS and it is being developed by Google /1/. Single page applications or SPAs have been gaining popularity due to better user experience and less server load compared to regular multi page applications /2/. SPAs in general are not a new concept but Angular as a framework for building SPAs is relatively new, having its first release in 2016 /1/.

Robert Martin collected a set of design principles to improve object-oriented software called the SOLID-principles /3/. These principles were first mentioned over 20 years ago in 1995 and they have been proven as a good guideline for software design. This thesis aims to find a way to utilize the SOLID-patterns in Angular to make code adaptive to change. All the SOLID-principles apply to almost any object-oriented language, so this thesis is somewhat relevant even when Angular is not used.

In short, this thesis aims to improve structure in Angular-projects. The goal is to find optimal design choices for projects to utilize. The ideal case would be that any requirement change or new feature would only affect the part of software where change is required. SOLID-principles have been a good guideline so far for object-oriented software in producing agile code. This thesis aims to apply them with Angular-framework and find best practices to structure Angular-applications.

2 COMPANY INTRODUCTION, WAPICE LTD.

Wapice is a leading technology partner, which offers high quality software expertise for its customers. The company was established in 1999 and during that time, its primary goal was to develop products for customer needs with the WAP-protocol. The company's head office is based in Vaasa, Finland and they currently employ over 340 people with ten offices also located in Finland /4/. Wapice has been growing steadily and the company's revenue was 21.6 million Euros in 2016 /5/.

Wapice is an IT-company with primary focus in software and hardware development. It has developed its own products for industrial needs such as IoT-TICKET, an IoT-platform, Summium sales configurator and EcoReaction, consumption information management solution for energy companies /6/. Along with their own products, company offers innovative solutions and consulting for customer needs.

3 SINGLE PAGE WEB APPLICATIONS

Single Page Application (SPA) is an application in the web browser which does not reload the page during use. SPA is almost like a native client loaded from the server into the users browser /10/.

Single Page Application is not a new concept, and developers have created SPA's with different techniques such as Java applets or Flash. Both techniques require a third-party plugin or software to function. JavaScript is supported by all the major browsers and most of the time it is the best way to develop SPAs today. Benefits to JavaScript over others include cross-platform compatibility, less bloat (no external plugins) and one client language. Since devices are getting more powerful, more computing tasks can be transferred from the server to the client without sacrificing user experience /10/.

Single Page Applications work by rendering pieces of the user interface based on user action. Instead of loading an entirely new page, only the required section is updated. In contrast, a regular web application redraws the entire application on user action, which causes a "flash" on the page. This loading period might take several seconds if the server is under heavy load. Instead of showing a blank white page for the user, SPA can display a loading animation and keep the other content available for browsing /10/.

Developers can also move business logic from the server towards the client side for faster decision making. For example, forms can be validated before sending them to the server, which allows users to change invalid inputs before sending content to the server. However, all user inputs still need to be validated on the server side and client-side validation can never be trusted to be sufficient for security reasons. From a developer's perspective SPAs are much easier to update compared to regular desktop applications. Developers can just upload new code to the server and all their users will get the newest version of the software. The application does not need any separate installer, navigating and perhaps bookmarking the correct URL is enough /10/.

JavaScript community is making a great effort to make JavaScript available also outside of the web browser. The best-known project is probably Node.js, which allows backend development with JS. Native-like applications are also endorsed with projects like Electron, PhoneGap and other alternatives.

4 OVERVIEW OF THE TECHNOLOGIES

Before looking at TypeScript and what it offers, one should have a general idea of typing in programming languages. In strongly typed languages each type of data is predefined as part of the language and the variables must be declared with one of the data types /8/. Consider the following pseudocode example using a strongly and statically typed language:

```
string a = "1";
int b = 1;
int result = a – b;
```

Code Snippet 1. Subtraction in a strongly typed language

This would result in an error, since the integer type cannot be subtracted from a string type. Since the language in question is statically typed, the error would arise at compile time /9/.

Weakly typed programming languages allow implicit conversion of types. By writing the previous example in JavaScript, which is a weakly and dynamically typed language, it would look like this:

```
var a = "1";
var b = 1;
var result = a – b;
```

Code Snippet 2. JavaScript example of string and integer subtraction

Value for result variable would be 0. However, if the arithmetic operator is changed from subtract to sum, the result will be “11”. The language will treat the variables as strings and concatenate them together. This kind of uncertain behavior will most likely increase the number of bugs in the software. Since JavaScript is also a dynamically typed language, possible problems will only emerge during runtime /9/.

4.1 TypeScript

TypeScript is a programming language designed and developed by Microsoft primarily intended for building large applications with JavaScript. It allows developers to use static typing and common object-oriented programming techniques such as modularity, classes and interfaces. Static type checking allows better support for tooling and Microsoft has already provided great tools for TypeScript development. TypeScript is a superset of JavaScript, which means that all JS code is also valid TS code. This eases the migration from native JS codebase to TypeScript. Developers can gradually migrate towards TS without a complete rewrite of the software. And since TypeScript compiles to JavaScript the code will run on any browser in any host /10/. Angular is written in TypeScript and Angular team suggests developers to use TypeScript when developing Angular applications. Examples in Angular documentation are also written using TypeScript.

4.2 Angular

Angular is a web application framework mainly developed by Google along with multiple other open source contributors. It can be used to build mobile, desktop and web applications for modern browsers using TypeScript or JavaScript. Angular is a complete rewrite from its predecessor AngularJS that has been a relatively popular SPA framework since its release in 2010. AngularJS was pioneering solutions for the web during that time. Today the challenges in web development have changed immensely /1/. Mobile devices are becoming the main platform for web applications and they surpassed desktop devices in internet usage in 2016 when in 2010 the figure was around 5% according to StatCounter /11/.

The main differences between Angular and its predecessor AngularJS is the adoption of TypeScript, component based architecture and differences in template engine in the newest version. It also includes improvements from ECMAScript 6 such as lambda operators and iterators /12/. ECMAScript 6 or ES6 is the 7th edition of ECMAScript standard created by Ecma International /13/. The Angular team is also making constant efforts to provide new versions of the framework to be backwards compatible with the previous version. These versions should officially be referred

as “Angular” without its version number instead of e.g. “Angular 4” /14/. At the time of writing, current major version for Angular is 5.

4.3 Angular CLI

Angular CLI (Command Line Interface) is a tool for initializing, developing and maintaining Angular projects. It is development by the Angular team and multiple other contributors. Angular CLI automates the development workflow by generating necessary boilerplate code for different pieces of Angular application. Boilerplate code refers to pieces of code that need to be included in multiple places with little to no alteration /15/. Angular CLI can also be used to run local development server, unit tests and end-to-end (E2E) tests and automating the build process /16/.

Features generated with the CLI tool follow the best practices from Angular style guide /16/. Angular CLI offers a very powerful local development server, which supports LiveReload. It monitors the changes in the project and modified files are automatically re-compiled and the browser gets refreshed, which eases and speeds up the development process /18/. Angular CLI generated projects include multiple things to be preconfigured: Karma unit test runner and Protractor end-to-end test framework, all required dependencies from the Node Package Manager (npm) are installed such as TypeScript. Karma is a unit test runner for JavaScript that executes given test code in multiple browsers and is mainly used for unit testing Angular applications /19/. Protractor is an end-to-end testing framework for Angular applications that runs tests in a real browser, interacting with the application as a normal user would /20/.

5 SOFTWARE ARCHITECTURE IN GENERAL

Software architecture is a vague term and it has multiple definitions. One definition for it is as follows: *"Software application architecture is the process of defining a structured solution that meets all of the technical and operational requirements, while optimizing common quality attributes such as performance, security, and manageability."* /21/. Software architecture involves multiple decisions based on a wide range of factors and each of these decisions can have a big impact on the maintainability, quality, performance and overall success of the system. Big and complex software must be built on a strong foundation. Architecture focuses on how the major elements and components within the software are used or interact with each other. The implementation details such as data structures or algorithms are design concerns, but very often architecture and design overlap /21/.

A software architect tries to combine business and technical requirements by understanding use cases, and then finding ways to implement them in the software. The goal is to recognize the requirements that affect the structure of the software. A good architecture would be flexible enough to handle the natural drift that occurs over time with technologies and user requirements. Along with usability and functionality, there are often other things to take into consideration, such as security and performance /21/.

The current thinking on architecture assumes that the software will evolve over time and not everything can be known up front to fully design the system. The design will need to evolve during the implementation stages of the application. To get started with the architectural design with limited knowledge, one should identify the foundational parts of the architecture that represent the greatest risk if done wrong, find the parts that are most likely to change, identify key assumptions made in the design and how to test them, and under what conditions refactoring is required /21/.

6 ANGULAR APPLICATION ARCHITECTURE OVERVIEW

Angular framework consists of multiple libraries some of them are considered core functionality, thus required, and some are optional. Applications can be written using HTML templates with Angular-specific markup, writing component classes to manage the templates, include logic in services and wrap these parts to modules. These are the major building blocks for Angular applications. The app is launched by bootstrapping the root module, and Angular will display the application in a browser. Figure 1**.** Angular architecture overview below visualizes the simplified and abstract description what happens inside Angular application /22/.

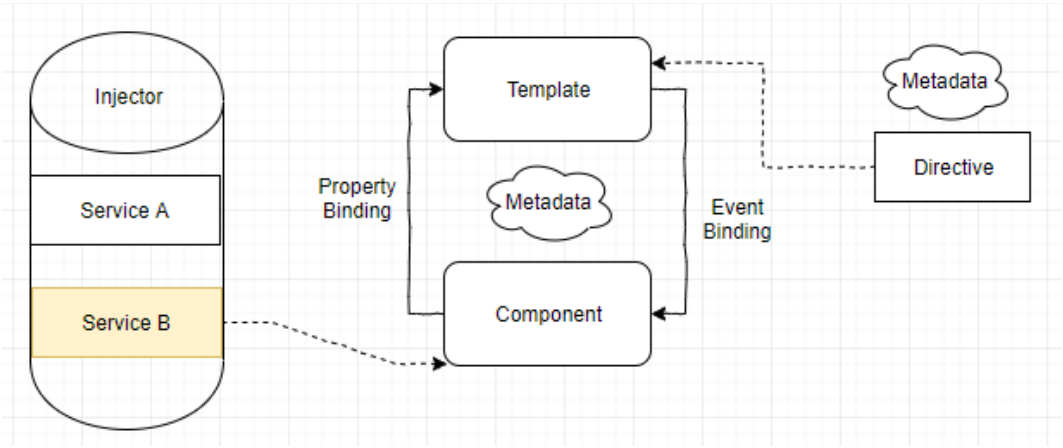

Figure 1. Angular architecture overview

The bootstrapping process creates the components defined in the bootstrap array. Since each bootstrapped component is usually the base of its own tree of components, bootstrapping triggers a cascade of component creations and includes them in the browser Document Object Model (DOM) /23/.

6.1 Modules

Angular modules (NgModules) are used to organize code similarly to JS modules although they differ in some ways. JavaScript modules are mainly used to namespace different scripts to avoid global variable collusions and other unwanted effects and they are exported in a single file. Angular modules can be configured

with metadata to include multiple files, and modules can only export the classes it owns or imports from other modules /24/.

NgModules configure the dependency injector and the compiler inside Angular. NgModules should be used to wrap cohesive blocks of functionality to its own module such as utilities or feature areas. Examples of NgModules are the libraries included in Angular such as the FormsModule, which includes functionality to form handling in Angular applications. These libraries are available for any Angular project to use to ease the development /25/.

6.2 Components and Templates

Components are the main building blocks of Angular applications identified with the @Component decorator. The component is a controller for the template. It includes the application logic to support the template view. The template is just a HTML document with some Angular specific syntax that provides Angular the instructions on how to render the component. The component class then interacts with the template through an application programming interface (API) /26/. The templates can include other components, which means that larger components can be built of smaller components.

Components take care of the data binding visualized in yllä Angular handles the data flow between template and component so that the developer does not have to manually push values into HTML and turn user responses into actions and value updates. Angular supports two-way data binding, which is a mechanism for coordinating parts of a template with parts of a component /26/. This allows seamless data flow from the controller to the DOM and vice versa.

6.3 Metadata

Metadata is used to provide instructions for Angular about a class. Metadata for a component tells Angular where to get the resources needed to present the component and its template. As an example, in the @Component decorator usually is configured the selector used for the component (how it can be referenced in a template

file), location of the template file, location of CSS styles for the template and what services it requires /26/.

6.4 Directives

When Angular renders a template, it transforms the DOM according to the instructions given by directives. Components are technically directives with a template. Directives usually appear in a template within an element, and are divided into components, structural and attribute directives. Structural directives alter the template layout by manipulating the DOM with given instructions. For example, the *ngIf -directive hides or displays elements with given conditions. Attribute directives alter the appearance or behavior of an element /26/. One example of this is the *ngClass-directive which dynamically adds and removes CSS classes to stylize elements.

6.5 Services

Service is an injectable class, which can encompass any value, function or feature that the application needs. Usually services are split into narrow, well-defined classes that do something specific well. In general, view-related functionality should be inside a component, and all other logic should be injected via services. However, Angular does not enforce these principles. User input validation and communicating with the server are examples of responsibilities that should be in a service. One key benefit of this is to provide required services to components through dependency injection (DI) to improve code reusability /27/.

7 SOLID-PRINCIPLES

SOLID is an acronym for a set of practices that, when implemented together, make code adaptive to change. SOLID-principles were introduced by Bob Martin over 15 years ago. SOLID acronym is made from following words:

"**S**" the single responsibility principle

"**O**" the open/closed principle

"**L**" the Liskov substitution principle

"**I**" the interface segregation principle

"**D**" the dependency inversion principle

These principles are valid even if used in isolation. When used in combination, these patterns give the project completely different structure, which is adaptive to change. However, one should not blindly follow these principles. Deciding where and when to use SOLID-principles is part of software development. Excessive use will make the code harder to read and might do more harm than good /28/. All the following code samples in this chapter are written in TypeScript for Angular framework.

7.1 Single Responsibility Principle

Single responsibility principle (SRP) guides developers to write code that has only a single reason to change. Classes with multiple responsibilities should delegate one or more responsibilities to other classes /28/. The example in Code Snippet 3 shows a class, which violates the SRP.

```
@Component({
  selector: 'app-root',
  template: '{{message}}'
})

export class ShowMessageComponent implements OnInit {
  public message: string;
  constructor() { }

  ngOnInit() {
    this.showMessage();
    setTimeout(() => this.showAnotherMessage(), 1500);
  }
  showMessage() {
    this.message = "Message";
    console.log("Message shown was: " + this.message);
  }

  showAnotherMessage() {
    this.message = "Another message";
    console.log("Another message was: " + this.message);
  }
}
```

Code Snippet 3. A class that violates the single responsibility principle

The example in Code Snippet 3 consists of three methods, ngOnInit() which is a lifecycle hook for Angular that triggers when the component is initialized. ShowMessage() and showAnotherMessage()-methods display messages to the user and logs the shown message to the console. Therefore, this class has two reasons to change. Possibly the message logic should be changed to allow setting the message manually, or perhaps all the logs should be saved to a database. To solve this problem the logging functionality should be separated to a service, which is then injected to the component. If the logging functionality needs to be reworked, it will only affect the logging service and thus follows the single responsibility principle. By moving the logging functionality to a service, it can be injected to other components

that might need logging functionality. The example in Code Snippet 4 shows how to implement the same functionality adhering to the single responsibility principle.

```
@Injectable()
export class LoggingService {
  public log(message: string) {
    console.log(message);
  }
}
@Component({
  selector: 'app-root',
  template: '{{message}}',
  providers: [LoggingService]
})

export class AppComponent implements OnInit {
  public message: string;
  constructor(private logger: LoggingService) { }
  ngOnInit() {
    this.showMessage();
    setTimeout(() => this.showAnotherMessage(), 1500);
  }

  showMessage() {
    this.message = "Message";
    this.logger.log("Message shown was: " + this.message);
  }

  showAnotherMessage() {
    this.message = "Another message";
    this.logger.log("Another message was: " + this.message);
  }
}
```

Code Snippet 4. Example of a class that follows the SRP

Code is now a bit more complicated but it follows the single responsibility principle. The example in Code Snippet 4 is obviously more complex than it needs to be. In bigger applications, the logging-service can be shared across multiple components and changes to logging functionality would only be done in the logging service. If the first solution was used across multiple components, changing logging from simple console output to logging every message to a database would require a lot more work.

7.2 Open/Closed Principle

Definition for open/closed principle (OCP) was described in the 1980s by Bertrand Mayer in his book Object-Oriented Software Construction. He defined the OCP as follows:

"Software entities should be open for extension, but closed for modification".

By following the OCP developers can reduce the risk of breaking the system when introducing new functionality by leaving the old implementation intact and extending the existing class with new features /28/. The example in Code Snippet 5 demonstrates a way to extend the LoggingService functionality to support logging to a database.

```
export abstract class ILogger {
    logMessage: (message: string) => void;
}
@Injectable()
export class ConsoleLoggerService implements ILogger {
  logMessage(message: string) {
    console.log(message);
  }
}
@Injectable()
export class DatabaseLoggerService implements ILogger {
  constructor(private http: Http) {  }
  logMessage(message: string) {
    this.http.post('http://localhost:3000/logs/',
{message}).subscribe();
  }
}

@Component({
  selector: 'app-ocp',
  template: '<h1>OCP Example</h1>',
  providers: [{provide: ILogger,
useClass: ConsoleLoggerService}]
})

export class OcpComponent implements OnInit {
  constructor(private logger: ILogger) { }

  ngOnInit() {
    this.logger.logMessage('Logged this');
  }
}
```

Code Snippet 5. Example of the open/closed principle using class-interface

As the example in Code Snippet 5 shows, different implementations for ILogger can be substituted with each other by configuring them in the providers array. Depending on requirements, the ConsoleLoggerService could be used on some components or modules and the DatabaseLoggerService on others. Extension points in TypeScript classes can be created with abstract methods, interfaces or even extending a class and overriding (rewriting) the method. However, TypeScript interfaces are lost after compiling, since JavaScript has no support for interfaces. If the developer tries to give an interface implementation of ILogger instead of an abstract class to providers array, it will result in a compile time error.

The example in Code Snippet 5 demonstrates a way where developer can use the abstract class as an interface aka class-interface. The class name will be used as a dependency injection token in Angular to map the interface to an implementation. Implementing this interface in the service classes is not necessary, but it improves the tooling support and can be considered as a better practice. Inheriting the base class for subclasses (services) and overriding the methods is also possible, but it raises another issue. Since multiple inheritance is not supported in TypeScript, inheriting the ILogger abstract class would prevent inheriting another, more suitable class. There are several other ways to achieve the same functionality. An abstract class could implement multiple interfaces and then the service could extend the base class. Along with the single inheritance issue mentioned previously, it leads to "interface soup" anti-pattern which is explained in the Interface Segregation Principle-section. Another method would be using the @Inject()-decorator in the constructor of the component and inject the concrete implementation to an interface.

7.3 Liskov Substitution Principle

The Liskov substitution principle (LSP) is a guideline for creating inheritance hierarchies for classes. If the LSP is being followed, client can use any class or subclass without compromising the expected behavior. Whenever the LSP is followed, the clients remain unaware of changes to the class hierarchy. When there are no changes to the interface, there should be no reason to change the existing code. The LSP

helps to enforce the open/closed and single responsibility principles /28/. The author of the principle, Barbara Liskov, defined it as follows:

"*If S is a subtype of T, then objects of type T may be replaced with objects of type S, without breaking the program.".*

Consider the example in Code Snippet 6, where RubberDuck class extends Duck class.

```
export class Duck {
    fly(): void { console.log('Flying!'); }
    quack(): void { console.log('Quack!'); }
    swim(): void { console.log('Swimming!'); }
}
export class RealDuck extends Duck {
    dive() { console.log('Diving!'); }
}
export class RubberDuck extends Duck {
    quack(): void { console.log('Squeak!'); }
    fly(): void {
        throw new Error('Rubber duck cannot fly');
    }
}
```

Code Snippet 6. Example of a class that violates the LSP

The rubber duck class in Code Snippet 6 breaks the Liskov substitution principle. If the class names are injected to the definition, it is obvious. "If RubberDuck is a subtype of Duck, then objects of type Duck may be replaced with objects of type RubberDuck, without breaking the program". Since the fly()-method in RubberDuck-class throws an error, it breaks the program and is a violation of the LSP. RealDuck-class does not break the LSP, since any object of type RealDuck could be used as a Duck without problems.

7.4 Interface Segregation Principle

Interfaces are important tools in modern object-oriented programming. They represent the boundaries between the behavior that the client code needs and how that

behavior is implemented. Interface segregation principle (ISP) suggests that interfaces should be small. If a client does not need a member from an interface, it does not make sense to require it to be implemented. All members of an interface must be implemented by the client /28/. Consider the following example in Code Snippet 7 where two ducks implement the IDuck-interface with three common duck behaviors flying, swimming and quacking.

```
export interface IDuck {
    fly();
    swim();
    quack();
}
export class RegularDuck implements IDuck {
    fly() { /* Flying logic */}
    swim() { /* Swimming logic */}
    quack() { /* Quacking logic */ }
}
export class RubberDuck implements IDuck {
    fly() {
        throw new Error('Rubber duck cannot fly');
    }
    swim() { /* Swimming logic */}
    quack() { /* Quacking logic */ }
}
```

Code Snippet 7. Example of a class that violates the ISP

RubberDucks cannot fly, even though they can swim and quack. Instead of a large IDuck-interface, the methods should be split to smaller interfaces to be implemented by the clients that can and should have that behavior. The next example follows the interface segregation principle by dividing the functionality into separate interfaces and the clients only implement the functionality they need.

```
export interface IFlyable {
    fly();
}
export interface ISwimmable {
    swim();
}
export interface IQuackable {
    quack();
}
export class RubberDuck implements ISwimmable, IQuackable {
    swim() { /* Swimming logic */}
    quack() { /* Quacking logic */ }
}
export class RegularDuck implements IFlyable, ISwimmable, IQuackable {
    fly() { /* Flying logic */}
    swim() { /* Swimming logic */}
    quack() { /* Quacking logic */ }
}
```

Code Snippet 8. IDuck interface split into smaller interfaces

Some developers reunify segregated interfaces to avoid multiple interface injection in classes. This could be done in the Code Snippet 8 by combining ISwimmable- and IQuackable-interfaces to e.g. ICommonDuckBehavior. However, this could lead to the interface soup anti-pattern as it removes the benefits that the ISP offers /28/. Anti-patterns are solutions to common problems where the solution is ineffective and may result in undesired consequences /29/. By combining the interfaces, the clients are again required to provide implementations for both methods. Even though the example has only one method per interface, following the ISP principle does not mean that every method should be in a separate interface.

7.5 Dependency Inversion Principle

The last principle and the D of SOLID-principles is called the dependency inversion principle (DIP) and it is defined as follows:

A. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
B. *Abstractions should not depend on details. Details should depend on abstractions.*

In practice, this means that dependency inversion introduces abstractions that are depended on by client code and by the implementers /28/. Following UML (Unified Modeling Language) diagram demonstrates a situation where the OcpComponent uses a separate ConsoleLogger-class for logging and below that is the same thing expressed as code.

Figure 2. OcpComponent with direct dependency to ConsoleLogger

```
export class ConsoleLogger {
    logToConsole(message: string) {
        console.log(message);
    }
}
export class OcpComponent {
    logEvent() {
        const cl = new ConsoleLogger();
        cl.logToConsole('Event');
    }
}
```

Code Snippet 9. Direct dependency to ConsoleLogger

In the OCP section, the requirement was to have a possibility for console and database logging, depending on the situation. To fulfill the same requirements, it would be possible to create a new class with database logging functionality, and use it with the "new" keyword when necessary. However, any changes to these functionalities require a lot of work, especially when these functions would most probably be used

across the application. By following the DIP and changing the classes to depend on an abstraction, the code will look like in Code Snippet 5. Example of the open/closed principle using class-interfacein the OCP section. Below in Figure 3. is an example of the UML-diagram after the high- and low-level modules depend on the same interface.

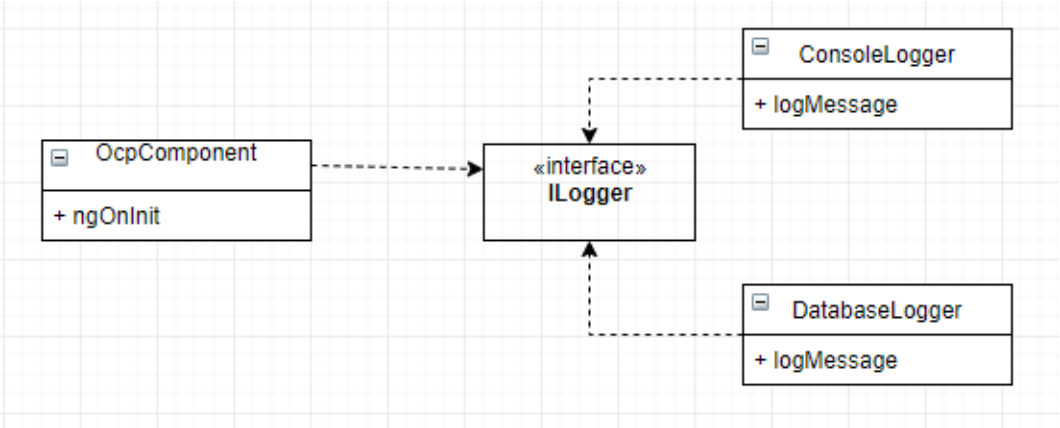

Figure 3. OcpComponent and Logger classes depend on the same interface

One thing to note that an interface is not necessarily a great abstraction. Well-abstracted interfaces can be used in many contexts, when poorly abstracted interfaces can be used only on very specific use-cases. An example of an abstraction that could be improved was shown in the ISP-section in the Code Snippet 8. IDuck interface split into smaller interfaces The IQuackable-interface could be renamed to e.g. ISound or a similar interface, which could then be used on other classes of animals or things that can emit a sound.

7.6 Dependency Injection

Dependency injection (DI) is not the same as DIP, even though they are closely related. Dependency injection is a technique that can be used to achieve the dependency inversion. DI itself can be achieved in multiple ways such as constructor or parameter injection or using an external DI framework. Angular has its own built-in dependency injection framework, which is used in almost every Angular project /30/.

When using dependency injection, the required dependencies are passed into the requiring class, instead of creating them inside the client. This was demonstrated in the previous section about DIP. Code Snippet 9**.** Direct dependency to ConsoleLogger. included an example of a class that did not use dependency injection. The class itself created its own dependency using the “new”-keyword. The example in Code Snippet 5. Example of the open/closed principle using class-interfaceuses dependency injection to pass in the ILogger-interface, and the Angular-frameworks inbuilt Http-class in the DatabaseLoggerService gets injected to the service with DI. These dependencies are defined in the components constructor and the Angular DI framework resolves the dependencies and injects them for the class to use.

DI helps developers to create more reusable and testable code. Dependencies can be configured externally to allow more reusable components and unit testing becomes easier when developers can control and mock the dependencies during testing /31/. Unit tests are used to test parts of the software in isolation so that they function as intended. The objective of mocking is to focus on the actual “unit” or piece of code under test, and provide a simplified and simulated implementation of a required dependency /32/. Since adhering to SOLID-principles generates a lot of dependencies inside the codebase, DI is an important technique for managing them.

8 STRUCTURING ANGULAR APPLICATIONS

The bigger applications built with Angular include multiple components. The components can be split roughly to “smart” and “dumb” components, also known as container and presentational components. Using smart and dumb components helps developers to separate responsibilities (SRP) and improve reusability. The components can then be further bundled into NgModules which can be used to isolate, test and re-use features /33/.

8.1 Dumb Component

A dumb or a presentational component is only aware of itself. It does not know what happens outside of it. It receives input via property bindings, using the @Input decorator, and emits output data as events using the @Output decorator /34/. Dumb components are ideally configurable to maximize reusability. These components are concerned how things look, and do not care how the data is loaded and might contain both smart and dumb components inside /35/.

An example of configurability could be an HTML input component, where the user could type in some text. As for configuration, the component could take in an InputOptions-class with attributes such as “useIcon”, “iconName” and “placeholderText”. These are defined in the parent component and injected to the dumb component. Configuration options would then alter the visual layout of the component by setting or removing the icon and adjusting the placeholder text. As a result, the same HTML input-element component can be used in multiple contexts. As an output, the component can just emit the value it received from the user to the parent component. The parent component will handle the output value as necessary or pass it forward to the smart component.

Dumb components do not necessarily have to be taken to the extreme like in the example. They can also be used to a more specific use-case inside the application e.g. a BirdListComponent, which would take an array of birds as an input. This component could still be reusable across the application, since it does not care where the data comes from. BirdListComponent could display all birds from a data

source or a filtered list of birds, which includes only those entries that start with the letter “b”. The data is handed to it with DI and the BirdListComponents only job is to display the given list of birds it receives.

8.2 Smart Component

Smart components are concerned with how things work. They provide the data and behavior for the dumb components. As for presentation, smart components should only layout the components they hold, not style them /35/. Since the smart component includes application specific dependencies such as services, it makes them a lot less reusable at least across different applications. The top level of a view is usually going to be a smart component /36/. These components are also responsible for working with the data they receive from dumb components.

In Figure 4 there is a mockup of a view. This mockup could be split to three separate components. Two dumb components for search and table controls, and one smart component that acts as a container for them.

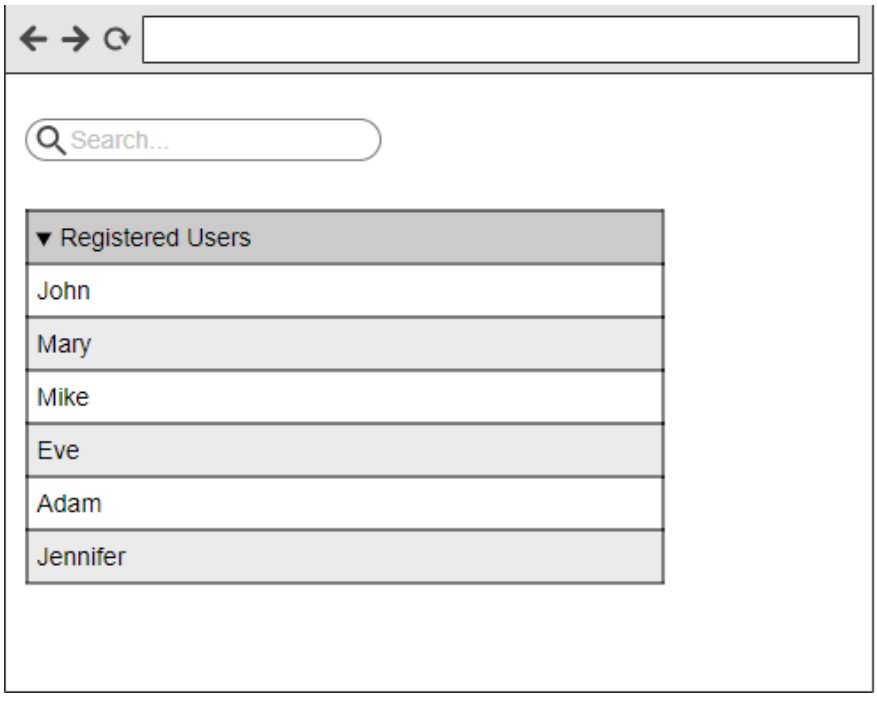

Figure 4. User interface for filtering registered users table

In this hypothetical use-case, an admin can view a list of all registered users. When the view is loaded, the smart component fetches all the users from a database to the table component. When the user types something to the search input, the search component reads and emits it to the smart component. After that, the smart component can filter the users list according to the search term and send the filtered array back to the table component to display.

8.3 Feature Modules

When the application grows, code should be organized relevant for a specific feature. Similar functionality should be included in a NgModule making it a feature module. With feature modules code related to a specific functionality or a feature can be separated from other code /37/. Separated modules allow developers to isolate the modules for testing and modules can be routed to load eagerly and lazily to affect performance /33/. Lazy loading improves application startup time since the application does not need to load everything all at once, it can only load the modules the user is required to use /38/. An example of a lazy loaded feature module could be an admin interface. Most users cannot use the admin page so it would make sense to load it on demand. Eager loading is the default application loading method. All eagerly loaded modules are loaded on startup.

Most feature modules depend on the application in question but the Angular style-guide suggests using core module and shared module in every application that takes advantage of feature modules. The core feature module should include all the singleton services that are shared across the application, modules required by the assets such as FormsModule and application-wide single-use components e.g. NavigationComponent. Core module is then imported only in the base AppModule. If the core module is imported in a lazy loaded module, the lazy loaded module will create its own copy of services, which will likely have undesirable results. Eagerly loaded modules have access to the AppModule's injector, so all the services in core module are available for them /33/.

A shared module should include all the components, directives and pipes that are reused in other feature modules. Providing services from a shared module should

be avoided. Since the shared module includes all the reused components, it should also import the modules required by these components, such as the FormsModule. As a summary, core module provides all the application-wide singleton services and single-use components and the shared module provides reusable components, directives and pipes for other feature modules /33/.

9 EXAMPLE PROJECT: SIMPLE USER MANAGEMENT

Before the application design can be started there must be some context on what should be built. In this case, the example application should include different views for an admin and a regular user and the possibility to login with an admin account. Both roles should have the possibility to view and filter the users table. The admin should be able to add, edit and delete users from the application. A deletion button must be placed next to the individual user records inside a table. These capabilities must not be available for regular users.

9.1 High-level Design of the Example Application

In this small example application, it is possible to design the system completely upfront. The application has high-level specifications for the architect to work with. The implementation details such as algorithms or styling do not have to be taken into consideration in this phase. All the structural specifications are known, but in bigger applications this is not usually the case.

Starting from the top of the requirements, it is specified that admin and user view should be separate, and admin view must not be available for regular users. This implies that the views should be at least different components. In most cases, regular user amount bypasses the amount of admin users in an application. With this knowledge, it makes sense to separate both functionalities into own modules for isolation and enable performance optimizations for regular users by lazy-loading the admin module when needed. Even without the lazy-loading, separating the distinct features to own feature-modules is the correct choice.

The admin and the regular users should both have the possibility to view and search from the users table. This is a clear indicator that the components should be shared across the application and thus, placed into a shared module. However, the admin must be able to delete the users from the table and a regular user can only view the table. The table should then be a configurable dumb-component to adjust it based on user role. It also makes sense to separate the search box from the table component in case the application needs to grow. Creating components with the Angular

CLI is fast and easy, so the separation does not add much overhead but offers great benefits in the long run if the application development is continued.

So far it is known that three modules need to be created; admin, user and a shared module. Search-feature requires a shared service across the application, so that the search-component itself can stay “dumb”. Application-wide services should be placed into a core module, so a fourth module is required for the application.

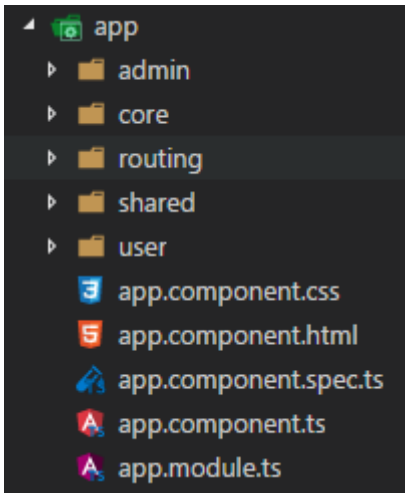

Figure 5. Folder structure of modules for the example application

Routing is added as a fifth module to separate routing logic from other modules, again to adhere to the SRP.

9.2 Component- and Service-level Design

Modules are now decided and more detail can be added. To fulfill the admin-module requirements, six components are needed. A user creation-, a creation dialog- and a login-component (specific for admin-module), user-table and the search component (reusable and shared dumb-components from shared-module) and the smart management component. For the login functionality, the module also includes the authguard-service for restricting access to the admin panel and the login-service. Smart management component needs access to registeredusers- and the search-service, which are in the core-module, to fulfill the requirements.

The shared-module includes search- and usertable-components and the models used in this application. Finally, the user-module requires the shared-module components and the smart details-component, which manages them.

9.3 Implementing the Application

At this point all the required modules, components and services are decided. While developing the application, the knowledge of the application and its limitations increases. During the development, it came apparent that the user experience could be improved greatly by adding a navigation bar. Navigation should include the admin- and user-views and a logout-button if a user is logged in. The navigation-component can be placed into the core-module, since it is shared across the application, but it is not a dumb-component used by other modules.

The admin-module is a lazy-loaded module, so Angular-injector does not have access to it unless the route is activated and the module contents loaded. Therefore, the login-service must be moved to core-module to gain access to logout()-method and isLoggedIn-property inside the navigation-component. However, the authguard-service was kept inside admin-module.

Both authguard-, and login-service could be inside the core-module and be considered as a valid solution, better than the current one if the application development continues. However, the example application is not under constant development, so the authguard-service can reside under admin-module to be lazy-loaded. It could also be argued that the performance gains from lazy-loading this service are non-existent, thus bad practice to place it inside the admin-module instead of the core-module. This example shows that design is not always straightforward. By creating a loosely-coupled architecture like this, adapting to change is easy. Moving the authguard-service from the admin-module to the core-module requires changes only to import statements and providers array inside the respective modules.

9.4 Finished Application

The finished application fulfills all the given requirements. The folder structure is good and the features are isolated into their modules. The folder structure of the finished application can be seen in Figure 6.

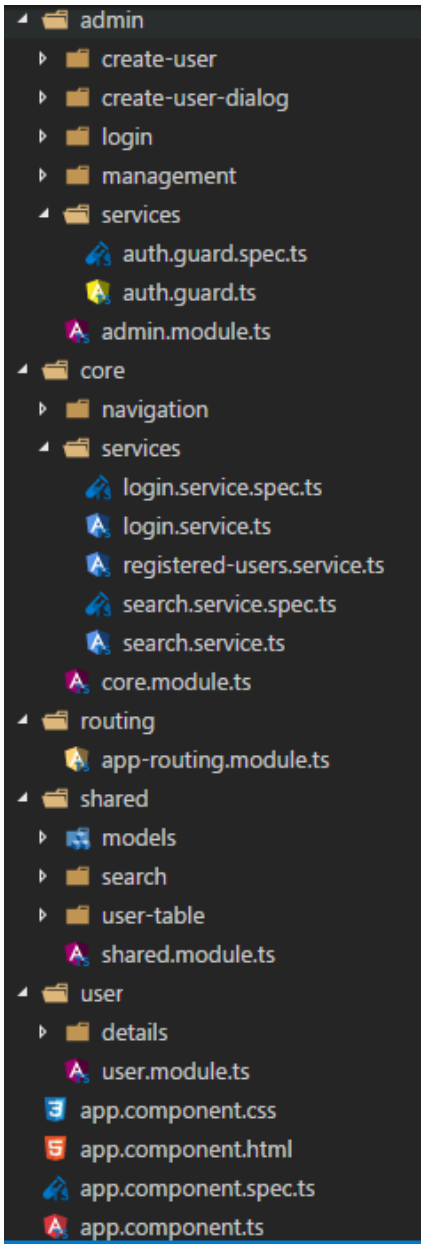

Figure 6. Folder structure for the finished application

Screenshots of the application in Figure 7 display the differences between the admin- and user-views. The admin-view has the added functionality of deletion inside

the table but the component is the same as in the user-view. The table component can receive TableOptions-class, which is used to alter the appearance and functionality by adding an extra column for delete-buttons.

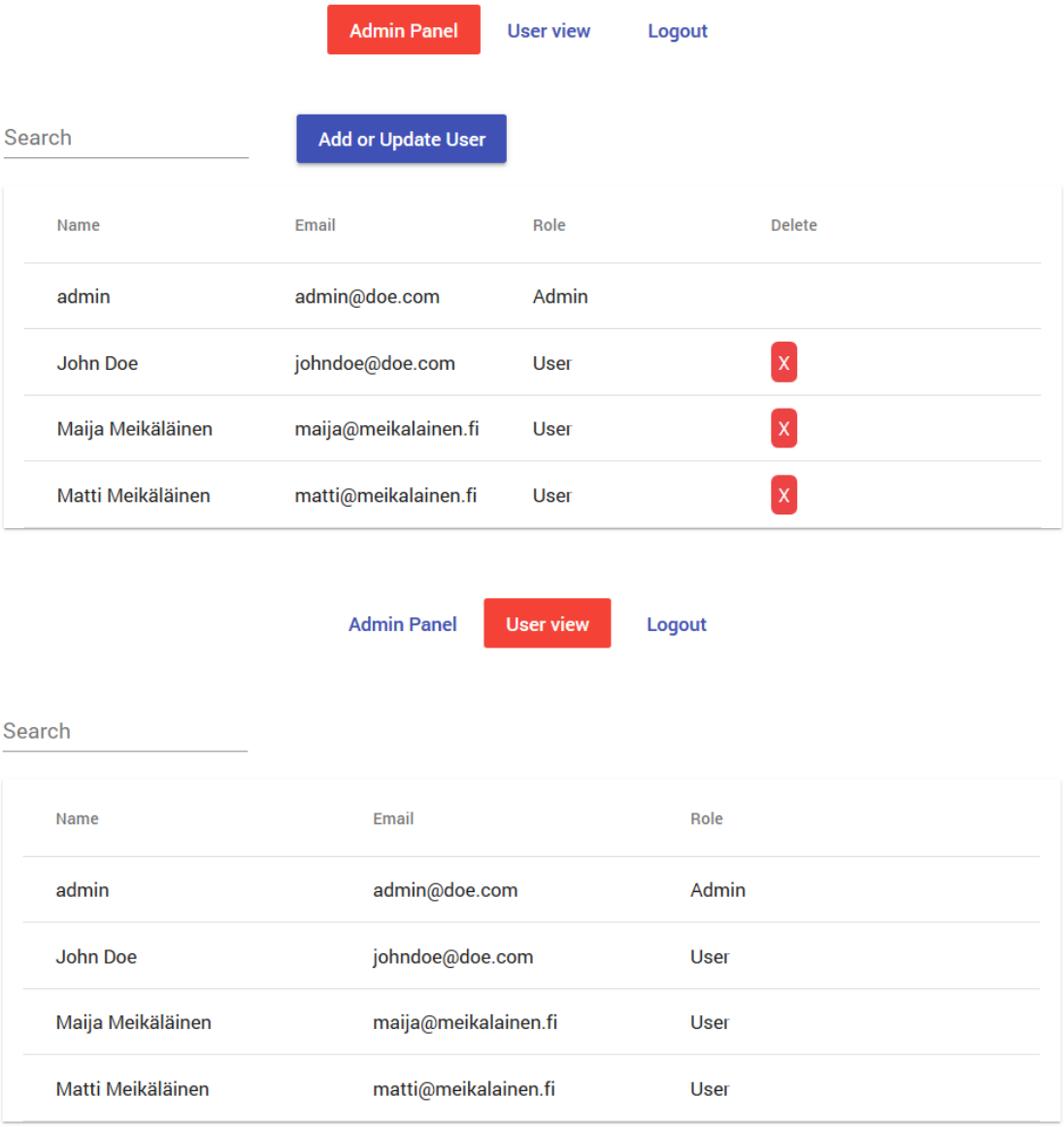

Figure 7. Comparison of the admin- and user-view

Lastly, the difference between dumb- and smart-components can be seen in Code Snippet 10 and Code Snippet 11.

```
@Component({
  selector: 'app-search',
  template: `
  <mat-form-field>
    <input matInput [formControl]="searchField" placeholder="Search"
(blur)="searchField.reset()">
  </mat-form-field>`,
  styleUrls: ['./search.component.css']
})
export class SearchComponent implements OnInit {

  @Output() searchTerm: EventEmitter<string> = new EventEmitter();
  searchField: FormControl = new FormControl();

  ngOnInit() {
    this.searchField.valueChanges
      .subscribe(term => {
          this.searchTerm.emit(term != null ? term : '');
      });
  }
}
```

Code Snippet 10. The “dumb” search-component that emits values

The search-component emits all the values it receives to the input-element. Subscribing to value changes happens on component load, inside the ngOnInit()-life-cycle hook. When the value changes, it is forwarded to the details component in Code Snippet 11.

```
@Component({
  selector: 'app-details',
  template: `
<div class="row">
  <div class="col-md-3 offset-md-3">
    <app-search (searchTerm)="search($event)"></app-search>
  </div>
</div>
<div class="row">
  <div class="col-md-6 offset-md-3">
    <app-user-table [dataSource]="filteredUsers"></app-user-table>
  </div>
</div>
`
})
export class DetailsComponent implements OnInit {
  users: User[];
  filteredUsers: User[];
  constructor(private userService: RegisteredUsersService,  private
searchService: SearchService) { }

  ngOnInit() {
    this.users = this.userService.readAllUsers();
    this.filteredUsers = this.users;
  }
  search(term: string) {
    this.filteredUsers = this.searchService.search(term, this.users);
  }
}
```

Code Snippet 11. "Smart" details-component that processes values from dumb components.

The details-component receives the value from the search-component and calls the injected search-service to perform filtering to the array. The filteredUsers-array contains the records left after search has been performed. The framework notices this change, and passes the changed array to the table component, which displays

the results. The details-component does not have any CSS-styles. It only contains the necessary wrapping divs for laying out the dumb-components.

10 CONCLUSIONS

The goal of this thesis was to find ways to structure Angular-applications so that they can adapt to changing requirements. This thesis was a high-level look to Angular-application structure and SOLID-principles.

Common issues with applications such as performance, testing and collaboration were solved by using feature modules inside Angular-applications. Feature modules allowed testing in isolation and lazy loading. Lazy loading improved performance by loading only the required parts of the application. Application development with multiple developers became easier due to isolated feature modules, compared to a single module.

The smart/dumb-model was introduced to provide a pattern which applies the single-responsibility-, interface segregation-, and dependency inversion principles. This pattern could be used to create reusable components. The Liskov substitution principle should be taken into consideration when creating inheritance hierarchies. Although components and services can be extended, it has not been used a lot to this date so the LSP applies more to regular TypeScript-classes. The open/closed principle was used to add features without altering the existing functionality. In most cases the SOLID-principles are applied by using interfaces. However, since TypeScript is compiled to JavaScript, which has no support for interfaces, it becomes difficult. Alternative solution was introduced by using class-interfaces.

Finally, the example project was provided for bringing theoretical concepts into practice. The thought process behind the decisions were explained. The project follows best practices and should be used as an example for future projects to keep the project as maintainable and scalable as possible.

REFERENCES

/1/ Kremer, J. 2016. Angular, version 2: proprioception-reinforcement. https://blog.angularjs.org/2016/09/angular2-final.html Accessed 24.2.2018.

/2/ Poe, C. 2017. What are the benefits of a Single Page App? https://www.bytelion.com/benefits-of-a-single-page-app/ Accessed 3.2.2018.

/3/ Martin, R. C. 2005. The Principles of OOD. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod Accessed 1.2.2018

/4/ Wapice basic info. 2017. https://www.wapice.com/contact Accessed 6.2.2018.

/5/ Wapice economic information. 2017. https://www.finder.fi/IT-sovelluksia+IT-ohjelmistoja/Wapice+Oy/Vaasa/yhteystiedot/225253 Accessed 6.2.2018.

/6/ Wapice products. 2018. https://www.wapice.com/products Accessed 6.2.2018.

/7/ Mikowski M. S., Powell J. C. 2012. Single Page Web Applications. Manning Publications. 1-5. http://deals.manningpublications.com/spa.pdf Accessed 17.2.2018.

/8/ Harbeck, R. 1999. Definition of strongly typed programming language. http://whatis.techtarget.com/definition/strongly-typed Accessed 13.2.2018.

/9/ Oracle. 2015. Difference between static and dynamic typing. https://docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html Accessed 13.2.2018.

/10/ Somasegar, S. 2012. TypeScript: JavaScript Development at Application Scale. https://blogs.msdn.microsoft.com/somasegar/2012/10/01/typescript-javascript-development-at-application-scale/ Accessed 13.2.2018.

/11/ Heisler, Y. 2016. Mobile internet usage surpasses desktop usage for the first time in history. https://bgr.com/2016/11/02/internet-usage-desktop-vs-mobile/ Accessed 24.2.2018.

/12/ Dziwoki, M. 2017. What's the difference between AngularJS and Angular? https://gorrion.io/blog/angularjs-vs-angular/ Accessed 24.2.2018.

/13/ Ecma International. 2016. ECMAScript® 2016 Language Specification. https://www.ecma-international.org/ecma-262/7.0/ Accessed 24.2.2018.

/14/ Angular press kit, official branding guidelines. https://angular.io/presskit Accessed 24.2.2018.

/15/ Lämmel, R., Jones S. L. P. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. https://www.researchgate.net/publication/221282345_Scrap_Your_Boilerplate_A_Practical_Design_Pattern_for_Generic_Programming Accessed 24.2.2018.

/16/ Multiple authors. 2017. Angular CLI wiki. https://github.com/angular/angular-cli/wiki Accessed 24.2.2018.

/17/ Angular CLI homepage. https://cli.angular.io/ Accessed 24.2.2018.

/18/ Van de Moere, J. 2017. The Ultimate Angular CLI Reference Guide. https://www.sitepoint.com/ultimate-angular-cli-reference/ Accessed 24.2.2018.

/19/ Multiple authors. Official Karma GitHub repository. https://github.com/karma-runner/karma Accessed 24.2.2018.

/20/ Official Protractor homepage. http://www.protractortest.org/#/ Accessed 24.2.2018.

/21/ Microsoft. 2009. Microsoft Application Architecture Guide, 2nd Edition. https://msdn.microsoft.com/en-us/library/ff650706.aspx Accessed 24.2.2018.

/22/ Angular documentation. Architecture overview. https://angular.io/guide/architecture Accessed 13.3.2018.

/23/ Angular Documentation. Bootstrapping. https://angular.io/guide/bootstrapping# Accessed 13.3.2018.

/24/ Angular documentation. JavaScript Modules vs. NgModules. https://angular.io/guide/ngmodule-vs-jsmodule Accessed 13.3.2018.

/25/ Angular Documentation. NgModules. https://angular.io/guide/ngmodules Accessed: 13.3.2018.

/26/ Angular Documentation. Introduction to Components. https://angular.io/guide/architecture-components Accessed: 17.3.2018.

/27/ Angular Documentation. Introduction to Services and Dependency Injection. https://angular.io/guide/architecture-services Accessed: 17.3.2018.

/28/ McLean Hall, G. 2017. Agile coding with design patterns and SOLID principles. Microsoft Press. 213-215, 259, 291, 308-309, 323

/29/ Agile Alliance. Definition of Antipattern. https://www.agilealliance.org/glossary/antipattern/ Accessed 3.4.2018

/30/ Angular documentation. The Dependency Injection pattern. https://angular.io/guide/dependency-injection-pattern Accessed 8.4.2018

/31/ Dependency Injection Benefits. http://tutorials.jenkov.com/dependency-injection/dependency-injection-benefits.html Accessed 9.4.2018

/32/ Telerik. Unit Testing and Mocking Explained. https://www.telerik.com/products/mocking/unit-testing.aspx Accessed 9.4.2018

/33/ Angular documentation. Style guide. https://angular.io/guide/styleguide#application-structure-and-ngmodules Accessed: 21.4.2018

/34/ Van de Moere, J. 2018. Understanding Component Architecture: Refactoring an Angular App. https://www.sitepoint.com/understanding-component-architecture-angular/ Accessed 11.4.2018

/35/ Abramov, D. 2015. Presentational and Container Components https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0 Accessed 11.4.2018

/36/ Angular University. 2017. Angular Architecture - Smart Components vs Presentational Components. https://blog.angular-university.io/angular-2-smart-components-vs-presentation-components-whats-the-difference-when-to-use-each-and-why/ Accessed 11.4.2018

/37/ Angular Documentation. Feature Modules. https://angular.io/guide/feature-modules Accessed 21.4.2018

/38/ Rangle.IO. Lazy Loading a Module. https://angular-2-training-book.rangle.io/handout/modules/lazy-loading-module.html Accessed 21.4.2018