



Hochschule für Angewandte Wissenschaften Hamburg  
*Hamburg University of Applied Sciences*

## **Bachelor Thesis**

Ivan Mihaylov

Comparison of different design patterns for Single  
Page Applications using the example of a Chat and  
Ticket management system

**Ivan Mihaylov**

Comparison of different design patterns for Single  
Page Applications using the example of a Chat and  
Ticket management system

Bachelor Thesis based on the examination and study regulations  
for the Bachelor of Engineering degree programme  
Information Engineering

at the Department of Information and Electrical Engineering  
of the Faculty of Engineering and Computer Science  
of the University of Applied Sciences Hamburg

Supervising examiner : Prof. Dr. Klaus Jünemann  
Second examiner : Prof. Dr. rer. nat. Henning Dierks

Day of delivery March 21st 2019

**Ivan Mihaylov**

**Thema der Bachelorarbeit**

Vergleichen von verschiedenen Entwicklungsmustern für Single-Page-Webanwendungen an dem Beispiel eines Chat und Ticket Management Systems

**Stichworte**

Single-Page-Webanwendung; SPA; Frontend Entwicklung; Zustandsverwaltung; Angular; JavaScript; Redux

**Kurzzusammenfassung**

In dieser Arbeit wird der Entwicklungsprozess einer Single-Page-Webanwendung dargelegt. Dem Verlauf des standardisierten Systementwicklungs-Lebenszyklus folgend, wird ein Chat und Ticket Management Systems erstellt. Dieses Projekt zielt darauf ab, die Schwierigkeiten bei der Zustandsverwaltung von Frontend-Anwendungen durch einen Vergleich zwischen zwei verschiedenen Entwurfsmustern für Anwendungszustände aufzuzeigen und zu beheben. Zu diesem Zweck werden zwei Versionen derselben Anwendung bereitgestellt, in denen die Vor- und Nachteile jeder Implementierung beschrieben werden.

**Ivan Mihaylov**

**Title of the Bachelor Thesis**

Comparison of different design patterns for Single Page Applications using the example of a Chat and Ticket management system

**Keywords**

Single Page Application; SPA; Frontend development; State management; Angular; JavaScript; Redux

**Abstract**

This work presents the process of development of a Single Page Application. It follows the standard application development life-cycle to produce a Chat and Ticket management system. This project aims to address the difficulties involved in the state management of frontend applications by making a comparison between two different application state design patterns. For this purpose, two versions of the same application are provided, outlining the benefits and drawbacks of each implementation.

# Table of content

<b>List of tables .....</b>	<b>7</b>
<b>List of figures .....</b>	<b>8</b>
<b>Acronyms and abbreviations.....</b>	<b>10</b>
<b>1 Introduction .....</b>	<b>11</b>
1.1 Motivation.....	11
1.2 Goals.....	12
1.3 Problem Statement .....	12
1.4 Thesis Overview .....	13
<b>2 Background .....</b>	<b>14</b>
2.1 Angular framework and the SPA design pattern .....	14
2.1.1 Angular and Component-Based Architecture .....	16
2.2 State managing design patterns .....	18
2.3 Firebase .....	19
<b>3 Requirement Analysis.....</b>	<b>21</b>
3.1 Functional Requirements .....	21

Introduction	5
3.1.1 Authentication	21
3.1.2 Ticket board	22
3.1.3 Chat Service	24
3.1.4 Database	24
3.1.5 Summary	25
3.2 Non-functional Requirements	27
3.2.1 High maintainability	27
3.2.2 High performance	27
3.2.3 Excellent user experience	27
<b>4 Software design</b>	<b>28</b>
4.1 Authentication	28
4.1.1 Authentication server-side	28
4.1.2 Authentication client-side	29
4.2 Ticket Management Board	29
4.2.1 Component structure	30
4.2.2 Operational flow	32
4.2.3 Data layer synchronization	33
4.2.4 Summary	36
4.3 Chat Service	36
4.3.1 Additional requirements	37
4.3.2 Structural changes	37
4.3.3 Component Structure	38
4.3.4 Workflow	42
4.3.5 Data structure	45
4.3.6 Summary	46
4.4 Central state management design	47
4.4.1 Additional requirements	47
4.4.2 Workflow	47
4.4.3 Summary	53
<b>5 Implementation</b>	<b>54</b>

Introduction	6
5.1 Authentication .....	54
5.2 Ticket management system .....	55
5.3 Chat Service.....	59
5.4 Central state management implementation .....	61
<b>6 Testing .....</b>	<b>65</b>
6.1 Unit tests .....	65
6.2 Manual testing .....	65
6.2.1 Authentication and board component .....	65
6.2.2 Chat service.....	68
<b>7 Conclusion.....</b>	<b>70</b>
7.1 Future work.....	70
<b>References .....</b>	<b>72</b>
<b>Appendix A.....</b>	<b>74</b>
<b>Appendix B .....</b>	<b>76</b>
<b>Appendix C .....</b>	<b>79</b>

---

# List of tables

<b>Table 1: Comparison of NoSQL database and RDB [6]</b> .....	25
<b>Table 2: Functional requirements</b> .....	27
<b>Table 3: Table displaying the differences between Promises and Observables in JavaScript</b> .....	36
<b>Table 4: Functional requirements tests for the board component</b> .....	68
<b>Table 5 Functional tests for chat service</b> .....	69

# List of figures

Figure 1: Page lifecycle diagram [3] .....	15
Figure 2: Diagram, representing the interaction between Angular's building blocks [5]...	17
Figure 3: Flow chart of Redux design pattern.....	19
Figure 4: Use case diagram of the authentication process. ....	22
Figure 5: Screenshot of a ticket board filled with example tickets. ....	23
Figure 6 : Screenshot, showcasing how tickets will be moved from one column to the next. ....	23
Figure 7: Database user model representation.....	29
Figure 8: Class diagram of the ticket management board .....	31
Figure 9: Activity diagram representing the drag-and-drop action of the tickets .....	32
Figure 10: Use case diagram of ticket service.....	33
Figure 11: Sequence Diagram representing the ticket creation process.....	35
Figure 12: Block list diagram representing the root component structure.....	38
Figure 13: Component diagram of the chat functionality depicting the chat service hierarchy.....	39
Figure 14: Class diagram of the chat functionality.....	41
Figure 15: Activity diagram of the chat functionality flow for displaying messages .....	42
Figure 16: Communication Diagram depicting communication between two components. ....	44
Figure 17: Sequence diagram representing the mechanism for unread messages notification .....	45
Figure 18: Screenshot showing a possibility for a denormalized chat collection model ....	46
Figure 19: A flow chart representing a central state management system design .....	48
Figure 20: Sequence diagram showing a Redux implementation of a chat service function .....	51
Figure 21: Sequence diagram showing an implementation of a chat service function .....	52
Figure 22: Screenshot of the UI representing the Login component view .....	54
Figure 23: Screenshot of the UI representing the Sign Up component view with a wrong validation attempt alert.....	55
Figure 24: Screenshot of the board component UI illustrating the main page view .....	56
Figure 25: Screenshot of the board component's UI illustrating the drag and drop functionality of the tickets.....	57
Figure 26: Screenshot of the ticket editing component's UI illustrating the available editing options.....	58



---

<b>Figure 27: Screenshot of the backlog component's UI.....</b>	<b>58</b>
<b>Figure 28: Screenshot of the chat functionality UI showcasing a chat conversation .....</b>	<b>59</b>
<b>Figure 29: Screenshot of the users list UI illustrating the option for adding a new user to the chat room's list.....</b>	<b>60</b>
<b>Figure 30: Screenshot of the chat service UI illustrating the unread messages notification implementation.....</b>	<b>61</b>
<b>Figure 31: Class diagram of the chat service with Redux implementation .....</b>	<b>62</b>
<b>Figure 32: Screenshot of a code snippet, representing the top level reducer .....</b>	<b>64</b>

---

# Acronyms and abbreviations

API	Application Programming Interface
BaaS	Backend as a Service
CBA	Component-Based Architecture
DOM	Document Object Model
FaaS	Functions as a Service
HTTP	Hyper Text Transfer Protocol
JSON	JavaScript Object Notation
MPA	Multiple Page Application
NoSQL	Not only SQL
SDLC	System development life-cycle
SPA	Single Page Application
UI	User Interface

# 1 Introduction

Nowadays web development has been undergoing rapid changes to meet the modern requirements for faster development time, lower bandwidth restrictions for the 3G and 4G consumers together with multiple platforms compatibility (web, mobile etc.). All these, together with the evolution of technology and drive for automation have forced developers to make a switch from the traditional Multiple Page Applications (MPA) to the more modern web development design pattern of Single Page Applications (SPA).

This work will take a deeper look into SPAs and the different design patterns which could be adopted when developing such web application. Moreover, several frontend frameworks will be introduced to illustrate the different design patterns involved, depending on the type of the application. Using the example of Chat and Ticket management system, developed in the scope of this work, I will outline the benefits of the selected architecture making a comparison between two design patterns – Redux and CBA (Component-Based Architecture).

## 1.1 Motivation

The motivation for this work stems from the desire to explore the newest available web technologies, making a comparison between two cutting edge design patterns and showing the advantages and disadvantages of each implementation. As the web development is rapidly moving away from the Multi Page Application pattern and adopting the newer lighter technology of Single Page Applications, newer frameworks and design patterns are also being developed to utilize the development process. Therefore, even the most traditional “Model-view-controller” design pattern is being replaced in favor of more modern approaches. The value of this work comes from exploring these modern approaches and making a practical and theoretical comparison between two of the alternatives. This work could serve as guiding example for any future client side web development applications.

## 1.2 Goals

The primary goal of this thesis is to design and implement a Single Page Application which would allow the user to interact with an interface, allowing him to create and order tickets, together with an implementation of a chat service to make the management of tasks more fluid and allowing for a higher level of coordination between users. This application will be developed using two different implementations, one using the Redux design pattern and one using the component based architectural pattern. The main purpose for creating this application using two different approaches is to investigate the most important topics when it comes to web development: scalability, maintainability and ease of development. Although the function of the interface should not change the wiring of the components internally together with the state management of the application will differ significantly when inspected in detail. A discussion of this differences in the implementation process is what will lie in the heart of this work.

## 1.3 Problem Statement

The design of a full stack web application comes with a wide range of complexities that need to be handled in the different stages of development. For the development of the current application the following problems were examined:

1. **Project requirements:** requirement gathering and analysis is usually the first and most fundamental step in every Software Development Life Cycle (SDLC). As the requirements cannot be gathered completely at the beginning, close relation with the stakeholder is necessary to gather feedback after every release [1]. In this project however since outsourcing tasks to third parties is not an option, the requirements phase needs to be highly coordinated with the tools and knowledge of the developer. Moreover, as the deadline cannot be postponed, the complete list of requirements, provided in **Table 2**, is agreed upon at the beginning of the project.
2. **Client-side design:** the design of the application should allow for an implementation of a chat and ticket management system which would utilize two different design patterns. The separate parts of the application have to be broken down and specific design be created for each module. The data structures should be designed in a way that allows future changes without the need of restructuring

and consistency needs to be ensured. The system should provide the user with the option to create and assign tickets, as well as move them to the appropriate progress column, while having the opportunity to start a chat with a relevant colleague.

3. **Backend design:** a backend architecture should be designed which would provide a consistent and reliable data storage solution. Since the client-side of the application will provide the user with the ability to interact with an interface, the results of these interactions need to be stored and synchronized across multiple users. Furthermore, the backend solution should be both time efficient and should not require learning a new backend framework, so a serverless solution was chosen. As opposed to the traditional backend development which requires designing and setting up a server that would handle all data requests, Google's latest cloud service, Firebase, allows for coding functions for endpoints directly.

## 1.4 Thesis Overview

Here is a short description of the thesis and an overview of the chapters it contains:

**Chapter 2 - Background:** this chapter describes the building blocks of a Single Page Application. It provides details of the client-side framework Angular and the server-side platform Firebase used in this project.

**Chapter 3 - Requirement Analysis:** This chapter summarizes the function and non-functional requirements for the chat and ticket management system.

**Chapter 4 - Software Design:** This chapter covers the client-side and server-side design of the separate parts of the application, including authentication, chat service, ticket service and backend synchronization.

**Chapter 5 - Implementation:** This chapter shows the implementation decisions taken while following the design steps outlined in **Chapter 4**.

**Chapter 6 - Testing:** This chapter describes the options for testing the correct functionality of the application.

**Chapter 7 - Conclusion:** This chapter presents a final overview of the project and thoughts about the future development of the application.

## 2 Background

This chapter will give a brief introduction into the Angular framework, the Redux state management design pattern and Firebase. These technologies play a vital role in creating the project and therefore understanding the project.

### 2.1 Angular framework and the SPA design pattern

Angular is a JavaScript framework that helps developers build applications. The library provides a number of features that make it trivial to implement the complex requirements of modern applications, such as data binding, routing, and animations. It is a platform that makes it easy to build applications with the web by combining declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or the desktop [2].

The first version of Angular called Angular.js was introduced in 2010 and with it came a revolution to the way web applications were developed. It delivered a client-side framework that would utilize the Model-View-Controller architecture, dependency injection, two-way databinding and it was the first real Single Page Application solution.

However, before SPAs became the standard for developing client-side web applications, Multiple Page Applications (MPAs) were the standard. The largest difference between the two is that while MPAs would load a new page from the back-end every time the user clicks on a link on the page, an SPA would load a single page at the beginning of the process and only re-render specific parts of that page as the user interacts with it.

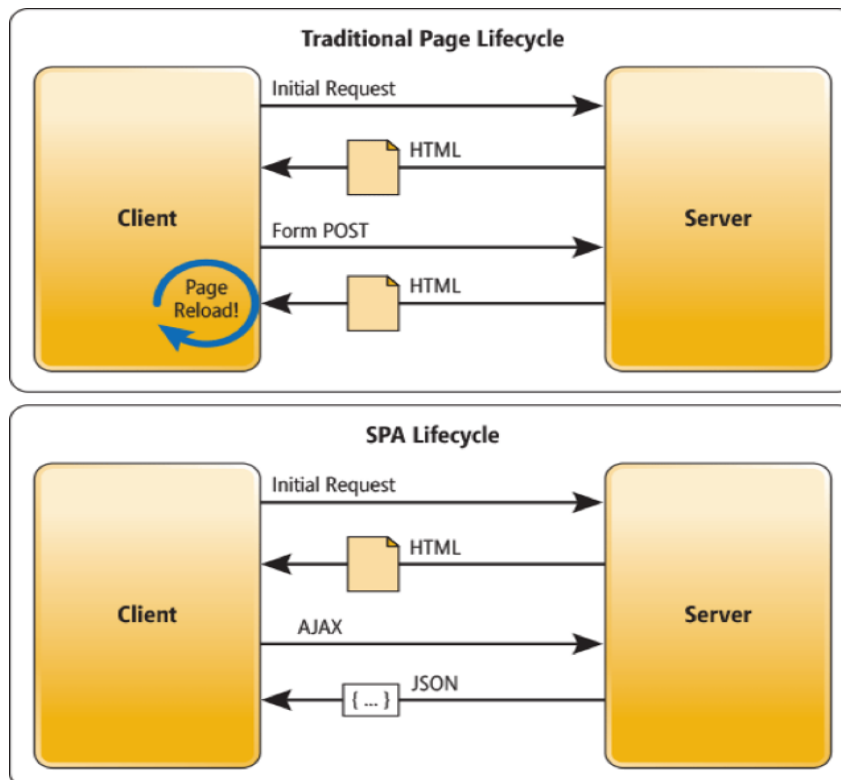


Figure 1: Page lifecycle diagram [3]

As shown in **Figure 1**, in contrast to traditional web applications which constantly request HTML files from the backend to be displayed in the browser, in the lifecycle of the SPAs only data is being transferred. Such data transfer over HTTP request decrease the relevant size of the communication and accordingly the time it takes for the page to load the data and re-render. The advantages of JavaScript SPA design over traditional web pages do not end here:

- No plugin required—Users access the application without concern for plugin installation, maintenance, and OS compatibility. Developers also do not need to worry about a separate security model, which reduces development and maintenance headaches.
- Less bloat—An SPA using JavaScript and HTML should use significantly fewer resources than a plugin that requires an additional run-time environment [4].

- One client language—Web architects and most developers have to know many languages and data formats—HTML, CSS, JSON, XML, JavaScript, SQL, PHP/Java/Ruby/Perl, and so on. Using a single programming language for everything on the client is a great way to reduce complexity [4].
- A more fluid and interactive page— With Flash or Java application on a web page, often the application is displayed in a box somewhere and many details are different than the HTML elements that surround it: the graphical widgets are different, the right-click is different, the sounds are different, and interaction with the rest of the page is limited. With a JavaScript SPA, the entire browser window is the application interface [4].

### 2.1.1 Angular and Component-Based Architecture

Another main feature that Single Page Applications brought to the world of client-side development was the concept of Component-Based Architecture (CBA). In essence, CBA represents a method of encapsulation of large pieces of user Interface that differs from the traditional MVC model. The need for such encapsulation is propagated by the structure of the main building blocks of any SPA – the components.

In the case of the SPA pattern using the Angular framework, the most basic UI building blocks are composed of four separate files:

- Two Typescript files containing the main functionality and the potential tests of a component.
- An HTML file that hold all the mark-up language for the component.
- A CSS or SCSS file that contains all the styling for the component.

```
|-- header
  |-- header.component.ts|html|scss|spec.ts
|-- footer
  |-- footer.component.ts|html|scss|spec.ts
```

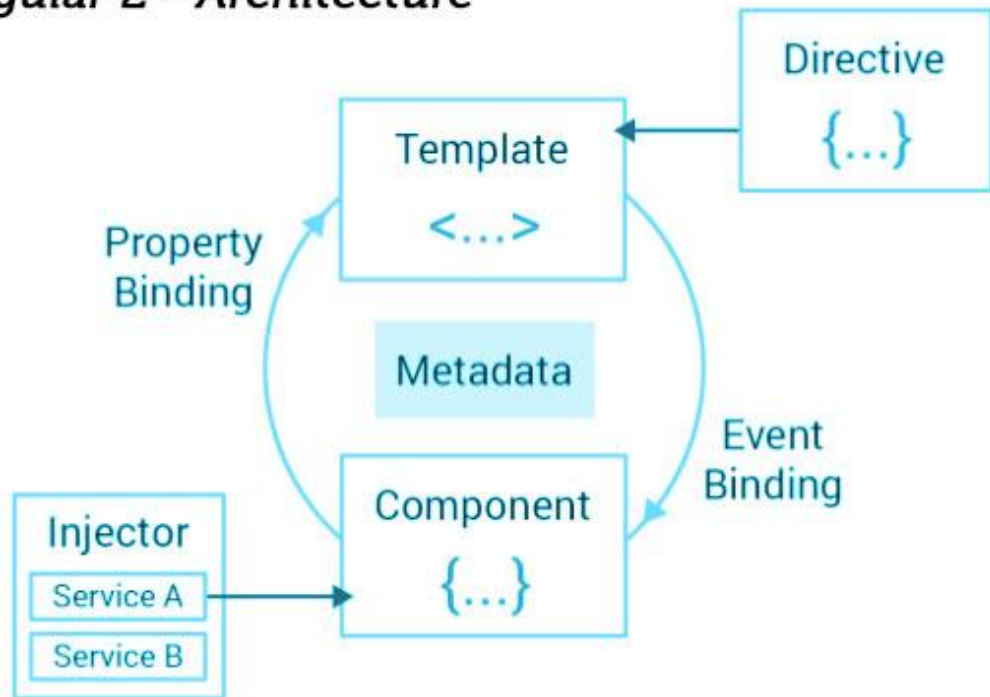
#### Example of component folder scaffold in the Chat and Ticket management system

Together all four files comprise the most basic structure for Single Page Applications called a *Component*. While the traditional MVC approach is designed to separate the responsibilities horizontally, CBA splits them vertically. This means that in the MVC case the UI, business logic and model all live in different levels of the architecture. CBA on the other hand tries to encapsulate all the relevant code that pertains to a given component inside its class as it is displayed in the code snippet above. And the responsibility is split on a



component-by-component basis. Each component has its own function, its own helper methods and routes and they are all present at the same level of the architecture.

## Angular 2 - Architecture



**Figure 2: Diagram, representing the interaction between Angular's building blocks [5]**

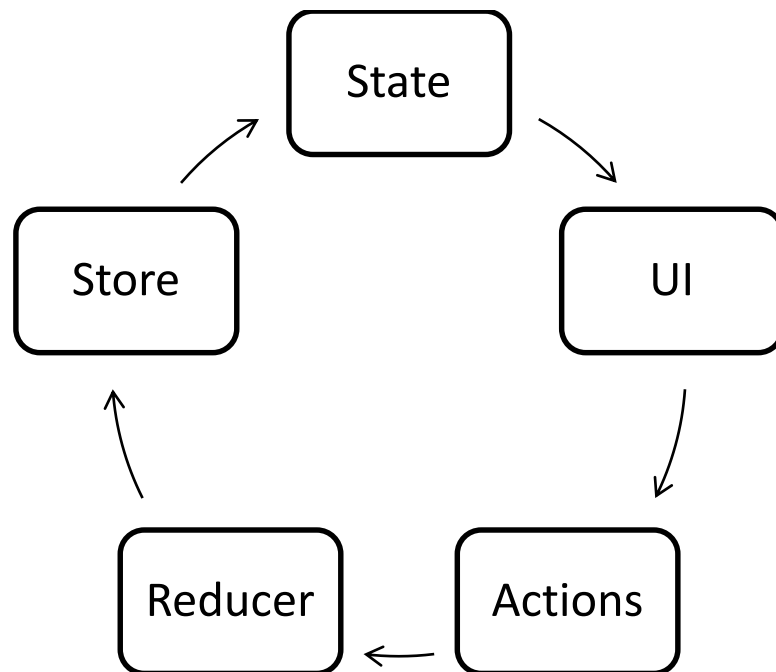
The necessity for components and their integration into the SPA model comes from the need of the SPA to be able to update dynamically the DOM. Since components have their own interfaces that can make calls to the server and update their interfaces, one component can refresh without affecting other components or the UI as a whole [6]. In the case of the Chat and Ticket management system, this allows for the chat messages to appear creating a new chat message component without the need for a page refresh or restructuring other components. The Angular framework is using a diffing algorithm that would detect the change in value of the "messages" array that triggers the DOM to re-render the messages feed component only.

## 2.2 State managing design patterns

One of the most notable features of the CBA architecture, the encapsulation of application state into separate components is however also one of its most prominent weaknesses. Since often components need to communicate with each other or share a piece of the application state, mechanisms for this communication need to be provided. This however poses a real challenge depending on the type and scale of the application. Each of the three large front end development frameworks and libraries today Angular, React.js or Vue.js offer their solution to this challenge, but in all three cases it comes down to the following options:

1. **Inputs and Outputs** – the Angular framework provides a feature to communicate state between nested components and that is through inputs from the parent component to the child component or outputs from the child component back to the parent one. This however limits the communication to only components that have a parent-child relationship. Therefore, if the components need to share state with unrelated components either third party software needs to be involved or one of the other methods mentioned below should be used.
2. **Services** - Angular allows for global variables that could be shared through services. Since services are built on the singleton pattern, they can be injected into any component using a dependency injection. This is what makes them ideal containers for methods that would be shared or reused in multiple components. However, over time the state of any component would be spread this way across services and would be hard for the developer to track the changes of state as the application grows.
3. **Central state management** – Since both of the aforementioned options either spread the component and application state across multiple services or multiple other components it can become a real challenge to find where a change in the state has been triggered from and therefore the cost of maintainability and scalability will increase substantially. Therefore, a design pattern that would alleviate this issue was necessary and the engineering team at Facebook came up with the library called Redux in 2015 that provided a solution. Since then the design pattern has been widely adopted in all major frontend development platforms and has been given the name “Redux”.

The Redux pattern in its core represents a one directional information flow. This means that a central store holding the state of the application will be provided and the communication between this store, the components and the views will follow the same pattern.



**Figure 3: Flow chart of Redux design pattern**

The main concept of the Redux pattern lies in the fact that the whole application state is gathered in one object and this object is a pure function. This means that the state is never mutated, instead a copy of the state is always provided and the changes to this copy are always performed in an immutable manner. In this way the underlying change detection system can pick up the changes and update the view.

## 2.3 Firebase

The traditional way of building a backend architecture for any application involves setting up servers and databases. Nevertheless, for the purpose of providing a comparison of two different frontend state management patterns the backend functionality could be mocked and the clients could be supplied with mocked data. The purpose of this work however, is to show a modern approach to data management architectures and patterns and provide a complete working example including a backend infrastructure.

Since, the way of building a backend has shifted in the last years with the emergence of cloud computing services, when building a serverless backend two particular services are of interest:

1. **Backend as a service (BaaS)** – an approach for providing web and mobile app developers with a way to connect their applications to backend cloud storage and processing while also providing common features such as user management, push notifications, social networking integration and other features that users demand from their apps [7].
2. **Functions as a service (FaaS)** – Firebase Functions, could be given as a good example for what FaaS represents. These are stateless functions, which are written in Node.js as a response to an HTTP call or some other type of cloud service event. With the help of this approach, the main focus remains on the development process and not on the architectural side. The FaaS provider takes care of the maintainability and the scalability of the services.

To make the term *serverless* more clear, this does not mean that there are no servers involved, rather the servers are provided and maintained by a third party that has built the infrastructure for other companies to outsource. Some of the reasons for the success of this approach nowadays include quicker releases of software, smaller operational cost, less software complexity and simplified deployment.

## 3 Requirement Analysis

### 3.1 Functional Requirements

The chat and ticket management application is a visualization tool where agile teams can have an overview which person is working on which task over a given sprint session. The tool provides a ticket generating functionality where a user can create a ticket and assign it to a person that will be responsible for working on it. Afterwards this ticket could move through the separate columns representing its working status. In this way, a stakeholder can have an overview of what task each team member is doing and how far along is the progress on it. Moreover, the tool provides chat functionality so that users can communicate directly through the tool and discuss relevant topics. Lastly the tool provides a backlog functionality where ideas could be saved for future tickets.

#### 3.1.1 Authentication

The first of the many functional requirements that a project like this needs to fulfill comes with the authentication of the users. In order to create separate teams so that the privileged information can only be displayed to the appropriate people the users need to be authenticated. This is a straight forward task where based on the authentication rights certain navigation paths are shown and others are hidden from the user. The following use case diagram in **Figure 4** displays how the authentication should work.

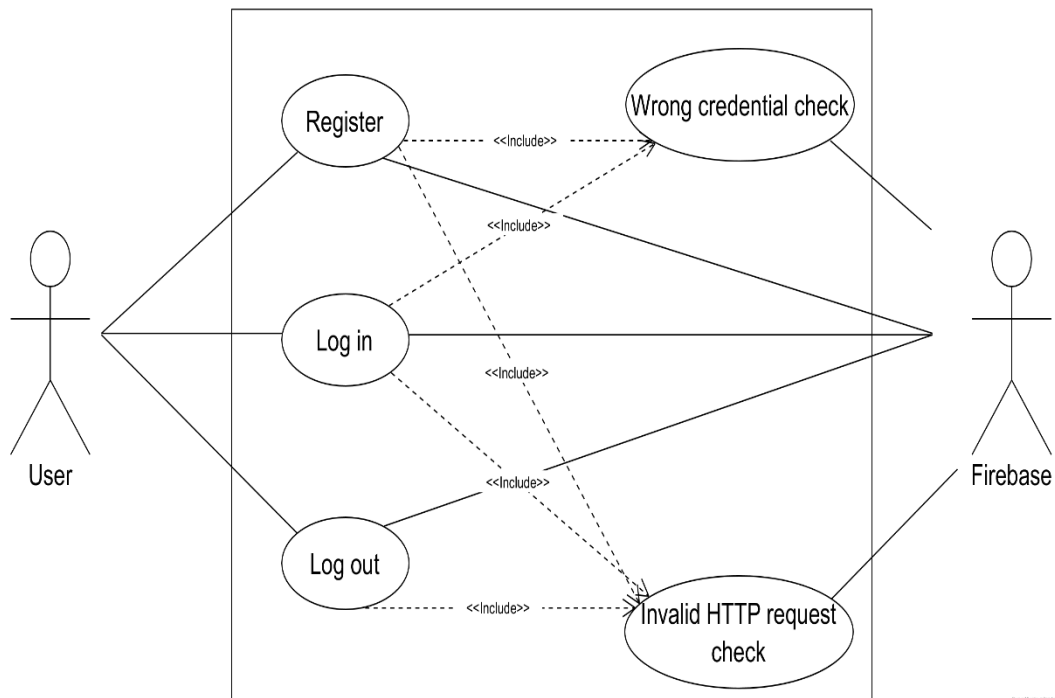
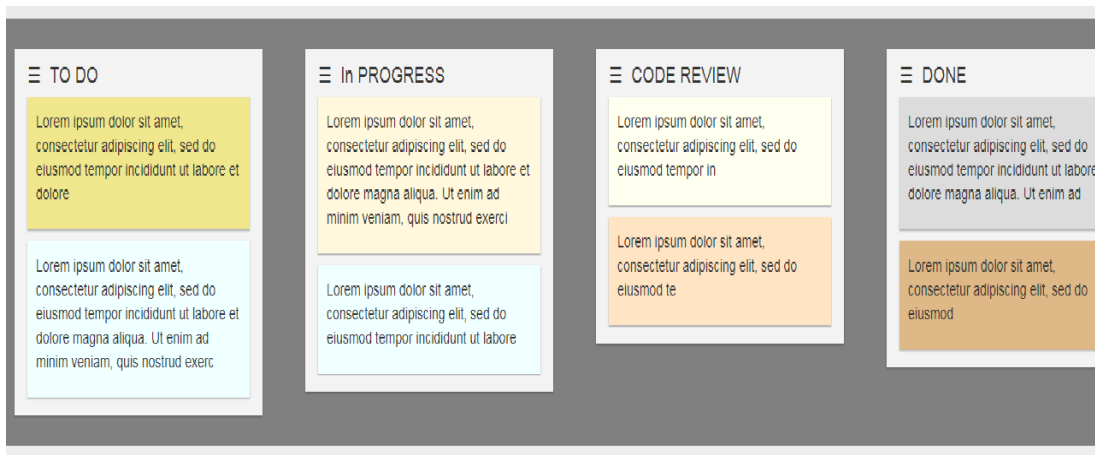


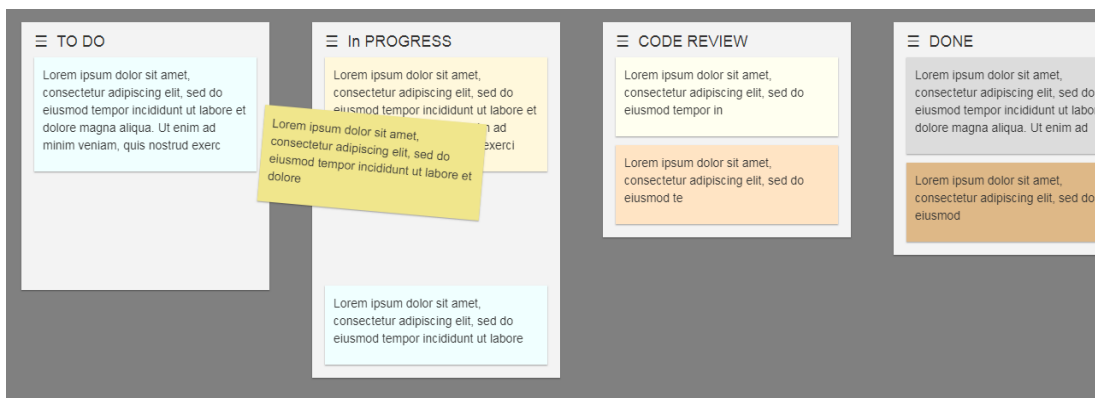
Figure 4: Use case diagram of the authentication process.

### 3.1.2 Ticket board

After authenticating the user, the most important functional requirements that stem from the project description above regarding the ticket board are therefore the creation of tickets and the ability to propagate these tickets along the separate progress columns. This functionality needs to be persisted through the database and all changes need to be displayed to all the users currently connected to this board.



**Figure 5: Screenshot of a ticket board filled with example tickets.**



**Figure 6 : Screenshot, showcasing how tickets will be moved from one column to the next.**

As depicted in **Figure 5** and **Figure 6** tickets should be able to be dragged and dropped from the user from one column to the next, signifying that work has been done on the relevant ticket. In this way the stakeholder can have an overview on the general progress of the sprint session. Since nowadays the development process has been strongly influenced by

this type of agile environments of development where everything happens in high paste such visualization tools are necessary to help cope with the organization of work and management of the working teams.

These columns however signify only the progress of a ticket that is already included in the current sprint. Since the nature of the work for which this board is being developed is prone to rapid changes, some tickets may need to be excluded from the current sprint, so others, more urgent can come in. This necessitates the presence of a backlog where all the tickets that are not relevant for the current sprint can be moved to. Therefore, a functionality, which preserves the state of the ticket but would transfer this ticket from the current board to the backlog and backwards is required.

### 3.1.3 Chat Service

To complement the development process a chat functionality needs to be implemented, that allows users to communicate with each other. This communication needs to be private so a simple single chat room will be insufficient for this task and private chatrooms need to be available for the users. Moreover, a notification system needs to be implemented that would allow a way of showing the user if there are messages in different chatrooms that are awaiting. All this functionality needs to be persisted in an appropriate way through the database.

### 3.1.4 Database

The choice of a particular database is an important moment for any application. This choice will define to a certain extent how the application will be designed and implemented later on. For this project's database a decision between a relational database and a NoSQL database type was of a particular importance.

Relational Databases (RDBs) which are based on the relational model are the best option for storing information that ranges from financial records, personal data and much more. However, as user requirements and hardware characteristics have evolved over time to include data warehouses, text management, and stream processing, these kind of process have very different requirements than traditional business data processing. Therefore, NoSQL is a breed of databases that are appearing in response to the limitations of existing relational databases. They are capable of handling large amounts of structured, unstructured, semi-structured and hybrid data with an amazing performance at reduced complexity and cost [8].

SL	NoSQL Database	Relational Database
1.	Unstructured way of storing the data	Completely structured way storing of data



2.	It can effectively handle million and billions of records	It can effectively handle few thousands of records
3.	It is never advised for transactional management	It is best suited for transactions
4.	Availability is preferred over consistency	Consistency is preferred over availability
5.	It scales horizontally as well as vertically.	It scales better vertically
6.	There is no need of normalization	Tables must be normalized
7.	Most of the NoSQL databases are schemaless	Traditional databases use the strict schema of database design.

**Table 1: Comparison of NoSQL database and RDB [6]**

Since the project is focused much more on chatting activity and other text management functionality, rather than any payment or other types of value exchanging transactions a NoSQL database choice provides a much more suitable solution. Even though this database could be mocked and contained within the client application, a complete working example was the desired outcome of this work. Therefore, appropriate NoSQL databases were examined for compatibility with Angular based Single Page Applications.

The most matured database that has also good integration with Angular and is a NoSQL type of database is MongoDB. However, MongoDB requires the development of a local server and a local backend architecture which would add more complexity and time for development. On the other hand, Firebase, which is a Google product like Angular, offers a real-time NoSQL database that has a native integration with Angular. It does not require additional development of backend since it is a cloud architecture that exposes a fully functional API that provides server services.

### 3.1.5 Summary

After understanding the main functional requirements for this project, Table2 below presents a full list with all the functional requirements originally agreed upon before starting work.

ID	Requirements	Implementation
IM1	The tool should have a home page that lists its purpose and gives brief introduction.	Must

IM2	The tool should have a Sign up/ Register page where the user will be able to create an account with which they will later be able to log in.	Must
IM3	Upon clicking the Sing Up button an account will be created in the database which will persist the user data.	Must
IM4	Minimum data requirements for the Sign up screen will be required. (No empty fields/ password length).	Want
IM5	All other fields of the tool will be hidden since the user is not yet Logged in.	Must
IM6	If an error occurs while the user is trying to register (using an email that is already in use) - an error message will pop up and the user will be denied registering.	Want
IM7	If the registration succeeds the user will be redirected to the main page where he will be able to see the ticket board and start a chat.	Must
IM8	The tool should have a Log in page for users who have already created an account in our database.	Must
IM9	If a log in error occurs it should be displayed and the user should be prompted to try again.	Want
IM10	The tool should have a Log out option on the Navigation bar element which will become active after the user has successfully logged in and upon selecting the option he should be logged out and redirected to the login screen with all the private fields hidden from him again.	Must
IM11	A logged user should be able to see all tickets and create new ones on the main page.	Must
IM12	User should be able to assign a team member or himself to the newly created ticket.	Must
IM13	User should be able to assign a priority to the ticket.	Want
IM14	User should be able to edit a ticket.	Want
IM15	User should be able to delete a ticket.	Want
IM16	User should be able to drag and drop tickets between columns.	Must
IM17	User should be able to send instant messages to other members.	Must

IM18	User should be able to post ideas to the backlog.	Must
IM19	Data should be persisted in the database	Must

**Table 2: Functional requirements**

## 3.2 Non-functional Requirements

### 3.2.1 High maintainability

At the time of writing the current version of Angular is Angular 6 in which the project has been developed. However, Angular is a framework that has the support of Google as the future platform for frontend development and new releases, most of them containing breaking changes come out approximately every eight months, so a good project structure is required in order to keep the updating process possible. On the other hand, JavaScript is a programming language that is notorious for its weak typing and even though TypeScript tends to correct this fault, a lot of the time developers do not use this language property, resulting in a code that is extremely difficult to maintain. Therefore, both of these requirements need to be satisfied in order for this project to remain highly maintainable.

### 3.2.2 High performance

The high performance of any application nowadays is a must. Most of the time this is the deciding factor when choosing a specific app for the task. Therefore, a rapid response is expected when a new chat is being started and all the messages need to be loaded. An appropriate data structure needs to be ensured that would require the least amount of data transfers as well as a short querying time. Since Firebase was chosen as the backend for this project which is a NoSQL database, appropriate flat models for the persisted objects should be provided. Moreover, no leaking subscriptions and on time change detection triggers have to be secured, to further optimize the application performance.

### 3.2.3 Excellent user experience

The high performance of the application is certainly one of the criteria for the good user experience that every application needs to deliver. There are however, other smaller aspects of the application that make it an all-round finished product. This project tries to deliver such service with the implementation of HTTP request interceptors that would invoke a loading spinner showing the user when he needs to wait before he can continue or different pop-ups delivering information that would guide the user through his interaction with the application. Ensuring at every step that the user knows what is happening is a vital concept of frontend development that needs to be sustained.

## 4 Software design

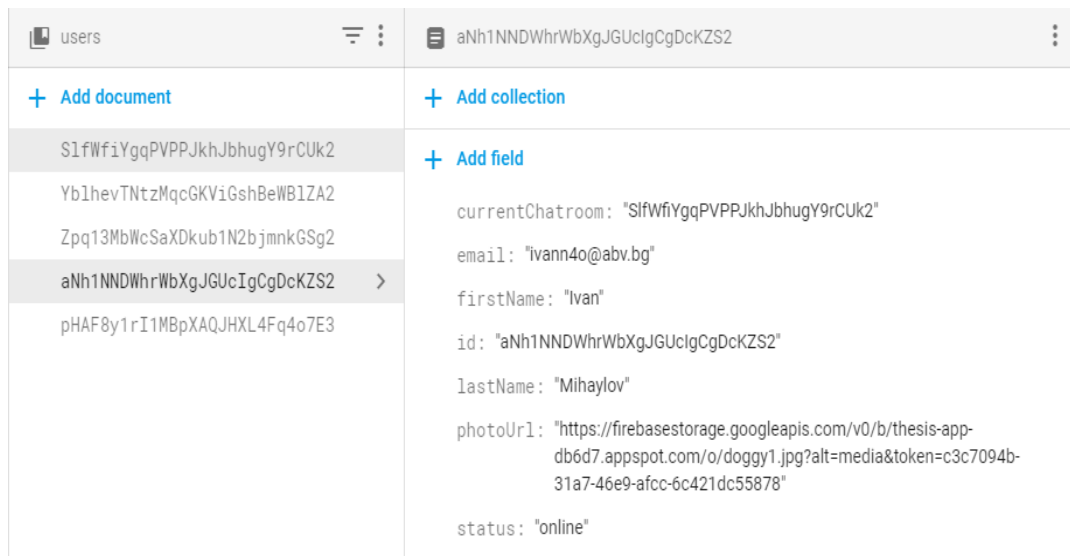
This chapter will introduce the separate steps taken, when designing the different parts of the application.

### 4.1 Authentication

#### 4.1.1 Authentication server-side

For the authentication phase of the application, Firebase provides several methods allowing the developer to choose from the database's user interface. Multiple methods can be selected simultaneously, including a sign in with Google account, Facebook account, phone number and more. For the needs of the project a standard username and password authentication was selected.

Firebase exposes two methods in its API for the successful registration and login using a username and password called *createUserWithEmailAndPassword* and *signInWithEmailAndPassword*. Both methods return a promise with the user credentials which makes them also the perfect place to update the user's profile to show whether he is online or not. Since this is a valuable feature for any chat application it is also added as part of this project.



**Figure 7: Database user model representation**

**Figure 7** presents a snapshot of the users' database entries, together with a view of the user model design. The user model is kept relevant and concise with only two extra properties relevant to the user himself – *status* and *currentChatroom*, both of which are relevant for the chat part of the application.

#### 4.1.2 Authentication client-side

Furthermore, since the authenticated user and his details are relevant throughout the application, the authentication is designed to happen in an Angular service. As mentioned in Chapter 2, services in Angular are one of the three ways to share state in Angular. Since services are built upon the singleton pattern and can be used through dependency injection to be inserted into any component this makes them the optimal solution for the distribution of the authentication state of the user across the components. In this way we subscribe to the authentication state once and then distribute this state internally instead of subscribing to the database from every component that requires the relevant information.

## 4.2 Ticket Management Board

The ticket management system serves the purpose of allowing the user to organize given tasks by providing a suitable structure for the tasks and a set of functionalities for working

with them. The tool is intended to provide a good overview of the workload in more agile working environments. A description of the structure and the functionalities of the tool are presented in the sections below.

#### 4.2.1 Component structure

The ticket management board is the main page of the application. It consists of three separate components extracted in a separate module that together comprise the view. Being able to wrap separate components, pipes and services into a module is a feature that is specific to the Angular framework. It helps not only with the organization of a project, but it also gives the developers the ability to split a larger application into smaller bundles, that would each be loaded at different times. This is an important feature since one of the main drawbacks of Single Page Applications is that they would try to download the whole application at the initialization of the app and afterwards only swap the data layer into the views. That behavior normally leads to much larger loading times at the start of the application. Angular therefore, gives the developers the ability to *lazy load* different modules on demand.

The term *lazy loading* is reserved for modules that are not being downloaded as part of the initialization of the application, instead the loading occurs when a specific route is being requested. Once the module has been download it would become a part of the Document Object Model (DOM) and Angular can preserve the fluidity of the application and the navigation to this module. For the needs of the current application however, lazy loading of the board module was considered, but since it would not at this point add benefit as the application is still reasonably small, adding that complexity was decided against.

The three components comprising the ticket management board can be separated into two groups. One purely presentational component, otherwise known as *dumb* component and two *smart* components. While the smart components contain a part of the logic that happens in the particular view, the *dumb* components are there usually for reusability purposes as it will be demonstrated in **Section 4.3** or as in this case for encapsulation of the view.

The depth of the components constructing a view can be as large as the number of HTML elements that are included since in practice every single HTML tag can be extracted into a separate component that has some additional functionality to it. It is therefore a task for the application designer to determine the necessary component depth of each view.

As shown in the class diagram of the ticket management board in **Figure 8**, the *BodyContentsComponent* class serves the single purpose of encapsulating the view. It sets a frame where each of the two other components should be fitted which provides an easier setup for styling the whole view as intended. This type of structure where similar components are grouped together into a larger presentational component brings another benefit in the Angular framework since it provides a parent-child relationship between the separate components, allowing them to pass data between one another. In comparison,

the component structure in the React framework for comprising a similar view would most likely look much more granularized since components in React are much simpler entities which do not need separate files or classes as in the case of Angular components.

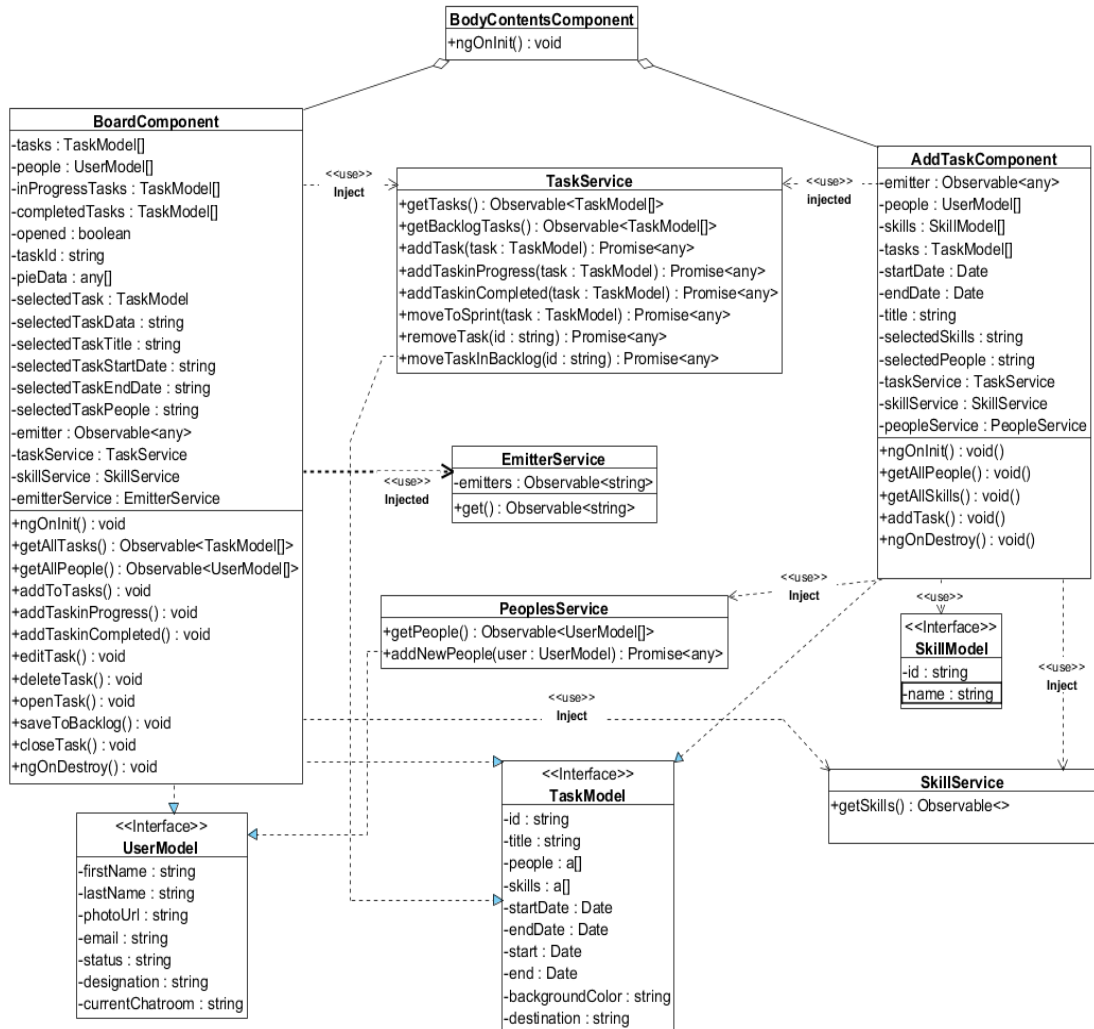


Figure 8: Class diagram of the ticket management board

The *BoardComponent* class provides the drag and drop functionality for the tickets, together with the ticket editing view. The drag and drop functionality of the component is achieved through an Angular library that provides a special directive.

There are three types of directives in Angular:

1. Components: directives with a template.

2. Structural directives: changes the DOM layout by adding and removing DOM elements.
3. Attribute directives: change the appearance or behavior of an element, component or another directive.

#### 4.2.2 Operational flow

In the case of the board component functionality, a structural directive is used that allows the rearrangement of DOM elements by providing handles to detect the changes to a certain element. In this way every time a ticket is drag into a certain column, a function is triggered that provides the object that has been dragged in its initial state, before the dragging and the target element's *id* in which it was dropped. In that way we can programmatically update the relevant columns' content while providing a nice animation that the user can understand.

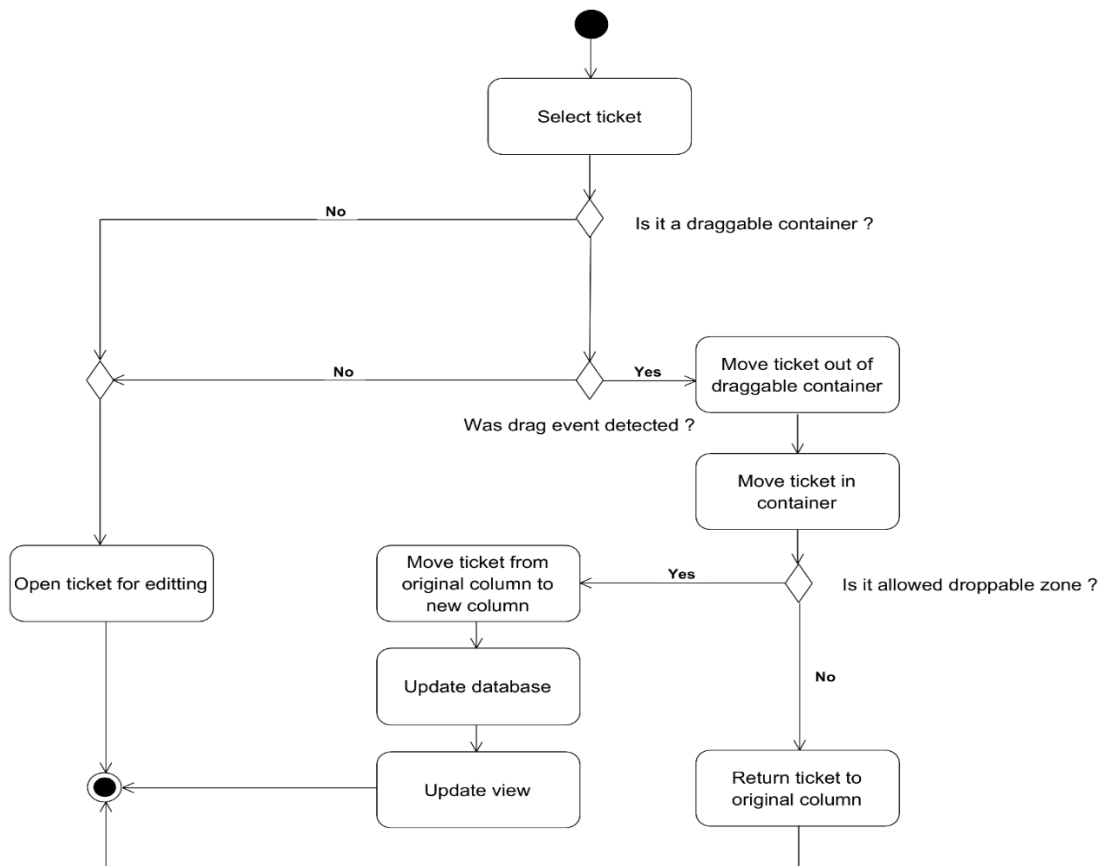
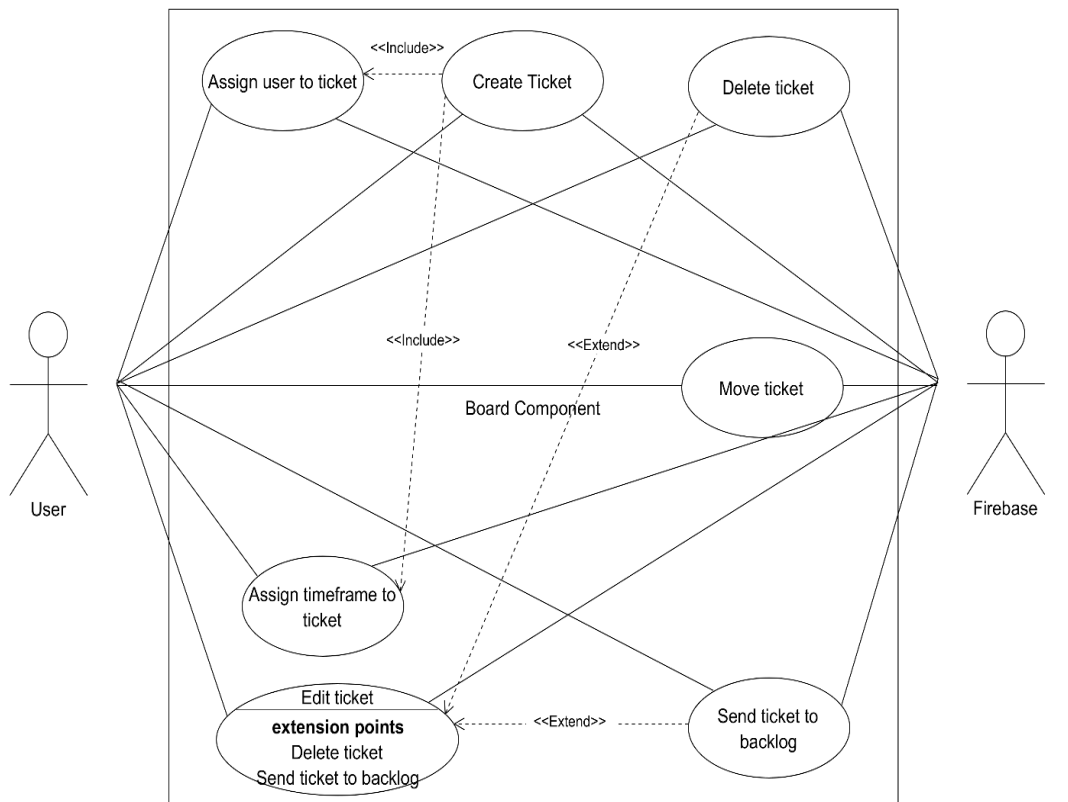


Figure 9: Activity diagram representing the drag-and-drop action of the tickets



**Figure 9** presents an activity diagram with all possibilities that the user has after selecting a certain ticket. In this use case, selecting is represented by a *mousedown* event which is a native Angular mouse event. The drag-and-drop directive then listens for the other two mouse events *drag* and *mouseup*. If a *mouseup* event is registered without the dragging event, then the ticket will be opened for editing and the process will terminate there. However, if the ticket is being dragged across columns, a check is made first whether or not the selected column over which the element is dragged over, is an allowed zone for dropping. If an attempt is made to drop the ticket over a no-drop zone, then the ticket will appear in the column from which it originated and no further actions will be performed.



**Figure 10: Use case diagram of ticket service**

#### 4.2.3 Data layer synchronization

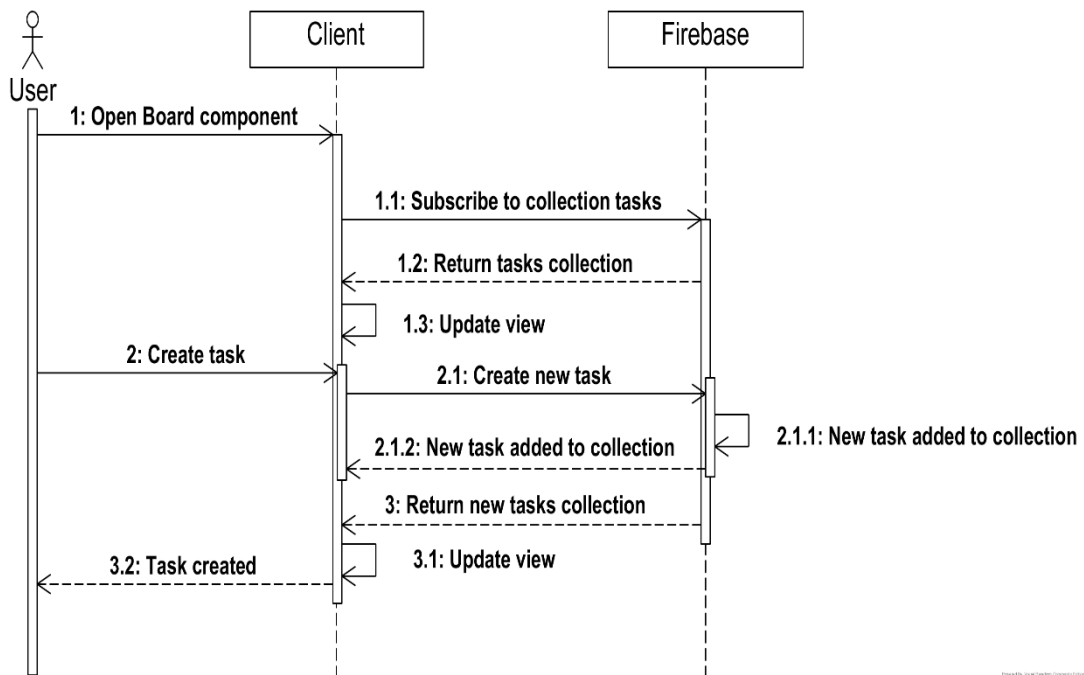
The synchronization of the tickets is made possible through Firebase, which is a real-time database. A real-time database system is a classic database system which is providing real-time constraints and ensures reliability on system's timing requirements. Timing constraints are not required to be extremely short but the database needs to manage explicit time

constraints in a predictable way using time-cognizant methods. This kind of database combines multiple features facilitation [9]:

- Description of data
- Maintenance of correctness and integrity of data
- Efficient access to the data
- Correct execution of query and transaction execution in spite of concurrency and failures

The most valuable feature of Firebase however is that it actively notifies all clients subscribed to the data of any changes. This means that all local variables that have a subscription to Firebase will be updated automatically upon an update to the database. Furthermore, this update of the local variables will cause Angular to refresh its views and in this way we can achieve a real-time synchronization between the data changing events and the particular view being displayed.

This sort of synchronization will not be possible out of the box if MongoDB was chosen as the database client for this project. Since MongoDB does not have any real-time features, every time we want to get an update of the data state in the database, a *GET* or a *PUT* request needs to be performed. This indeed could be an acceptable solution for the ticket management board, since we have to deal with only a few changes at a time. The chat part of the application however will suffer unacceptable lagging and it will be hard to synchronize the chat messages as they come. There are solutions through third party software like Socket.io that can enable databinding between the database and the client similar to what Firebase provides, but this would add a lot of complexity to the model as well.



**Figure 11: Sequence Diagram representing the ticket creation process**

The sequence diagram in **Figure 11** shows how the view ‘knows’ when to update itself when a ticket is created. Since the client binds a variable to the *tasks* collection in the database through a subscription in the *ngOnInit()* lifecycle hook of the component, this variable gets updated every time the collection is also updated. Even though the lifecycle hook is called only once at the initialization of the component, the variable continuously gets update because of its subscription. This subscription persists indefinitely or until the *unsubscribe()* method is called on it. This is the main difference between Observables and Promises in JavaScript. While Promises and AJAX calls are used for asynchronous operations, they are only able to handle single events. As soon as the operation completes or fails the promise will be exhausted and will not produce other values. On the other hand, an Observable acts as a steam of data.

	Observables	Promises
1.	Are cancellable	Not easy to cancel
2.	Emits multiple values over time	Emits only a single value
3.	Have a large amount of methods (mergeMap, flatMap, etc.)	Have only a few methods available

4.	Can be created from multiple sources, including events	Usually used only with asynchronous data return.
5.	Observable are lazy and they do not stream until they have been subscribed to.	Promise executes always.

**Table 3: Table displaying the differences between Promises and Observables in JavaScript**

The ability of the Observable to emit a stream of data over time with only a single subscription to it is what allows this active notification that Firebase provides. Therefore, the client is being updated as a last action in the sequence diagram in **Figure 11** with the new state of the *tasks* collection without an explicit request to the backend.

Since the subscription was created with the initialization of the component it will persist until the observable has been unsubscribed to. Best practices dictate, the unsubscribing to happen in the *ngOnDestroy()* lifecycle hook which is called when the component is being removed from the DOM.

#### 4.2.4 Summary

The ticket management board allows the user to see the current state of the tasks in the sprint session without the need to refresh the page in order to get the latest updates. With the observable implementation of the backend and the leveraged capabilities of the Angular framework, a real-time synchronization has been achieved which would help significantly to improve the user experience when using the ticket board. The data structure of the tasks collection allows for a single property to be updated in order for the task to be moved from one column to the next which ensures the efficiency and high performance of this approach. Furthermore, the extraction of the functionalities into a service allows for a high maintainability of the code base. In this way all three non-functional requirements listed in **Section 3.2** are satisfied.

### 4.3 Chat Service

The main purpose of the chat functionality is to give the users of this application the ability to exchange instant messages. In the requirement's phase of this project no other specifications for this functionality were requested. In the process of developing however, it was established that a common chatroom where everyone is automatically included and can post messages would not fit the organizational flow and needs of this application.

The need for private conversations however, brought a significant amount of structural changes to the chat functionality. Not only the models at the backend needed to be adjusted but also the component depth and structure on the client side.

#### 4.3.1 Additional requirements

The introduction of a chat functionality in this project led to uncertainties about the structure and the organization of the project's components. This section will address this issues.

Since the main purpose of the application is to provide a ticket management system and the chat functionality is a supplementary service, an appropriate place for the chat needs to be allocated. The main structural requirement for the chat is that it needs to be active on all pages after the login of the user. This however, dictates that it cannot be assign to its own page, but instead it would need to be sharing the view of all other pages. Therefore, finding an appropriate way to show the chat while making it an active background service on all views poses a challenge for the whole structure of the project.

Furthermore, the chat's rich functionalities demand that the chat would have to take a substantial part of the view. This is less relevant for the project's structure, but it requires an adequate solution that would solve those issues.

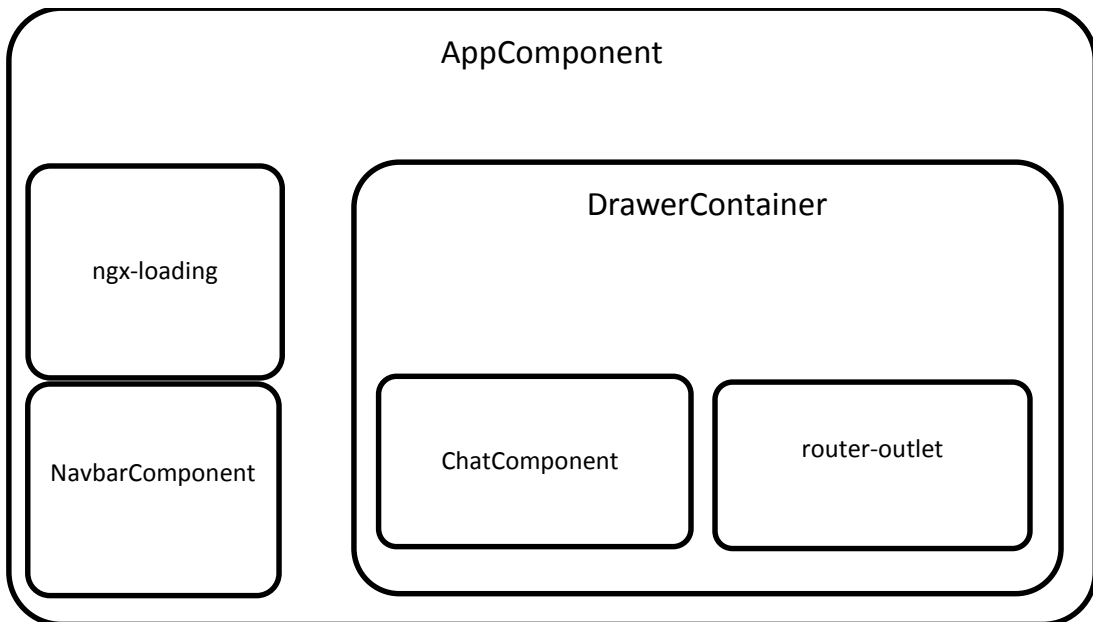
#### 4.3.2 Structural changes

The structural design issues that were presented in the previous section were resolved by placing the chat functionality into a side navigation component called a *drawer* component. This drawer component is designed to add side content to a small section of the application. It is designed to add collapsible side content alongside some primary content to a full screen app [10].

Placing the chat functionality into a collapsible container allows it to be attached to all pages without making in the primary focus of the view. In this way, the chat remains active throughout the application but also does not overpopulate the view. Moreover, since it is attached only to the main app component before the router-outlet which enables the navigation, it is only loaded once regardless of the in-app navigation requests. In this way, this approach for displaying the chat functionality provides multiple advantages:

1. **Efficiency** – it is loaded once after the user login. It is not reloaded with navigating between routes.
2. **User-friendliness** – it is allowed to take as much space as needed for styling without compromises and without taking any of the view space on the main page. It does not require to navigate away from the current screen.

3. **Maintenance** – It requires only one extra component, on top of the chat service implementation, that enables this functionality.
4. **Implementation speed** – it does not require any changes to the component structure of the current views to be implemented.



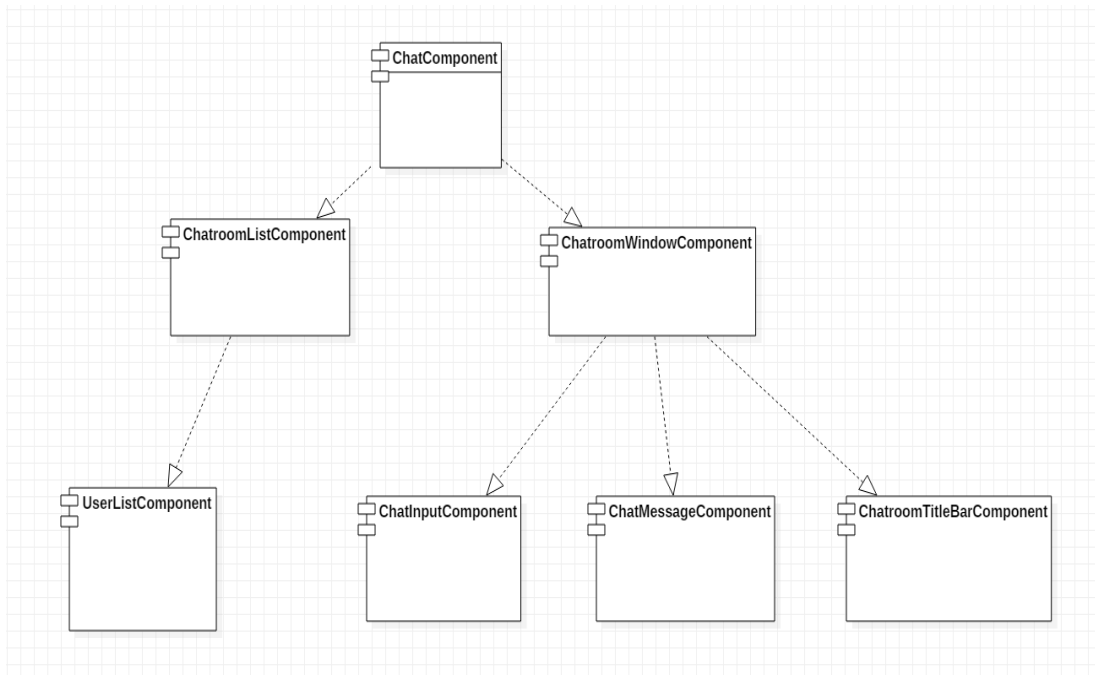
**Figure 12: Block list diagram representing the root component structure**

**Figure 12** represents the hierarchical structure of the root component of the application. It shows how the chat component is nested in the top most level outside of the navigation module. This allows the user to navigate to any available route in the application which will be shown in the router-outlet window, while keeping the chat active through the process.

### 4.3.3 Component Structure

The chat functionality is composed of seven components. The component diagram in **Figure 13** shows that there are two branches each with two levels of nested components inside the *ChatComponent* class. The relatively flat structure is a design decision since the most bottom components of both branches need to communicate with each other and passing variables along the components' tree needs to be facilitated. Since one of the two ways of

passing state between components is through the parent-child relationship of the components, the flat structure allows for a lot less tree nodes which need to pass the same information until the state gets passed from the *UserListComponent* to the *ChatMessageComponent* class.



**Figure 13: Component diagram of the chat functionality depicting the chat service hierarchy**

The *ChatComponent* class serves as a wrapper class that provides the layout for the two branches. It is a purely presentational component that has no other functions but passing state between the two branches. All the functionality of the chat service is spread through the rest of the components on **Figure 13**.

A brief description of the components functions can be found below:

- **ChatroomListComponent** – provides the user with the ability to select a chat room from a list of chat rooms or create a new chat room from a list of users.
- **UserListComponent** – provides the user with a list of usernames from which he can select one to be added to the shortlisted chat room names.

- **ChatroomWindowComponent** – provides a frame for the messages and the message input field. It sorts and displays the messages exchanged by the user.
- **ChatInputComponent** – serves for the creation of the messages.
- **ChatMessageComponent** – A presentational component used for displaying and styling each message individually, while having the same frame. It is created for reusability purposes.
- **ChatroomTitleBarComponent** - A presentational component used for styling the name of the active chatroom conversation.



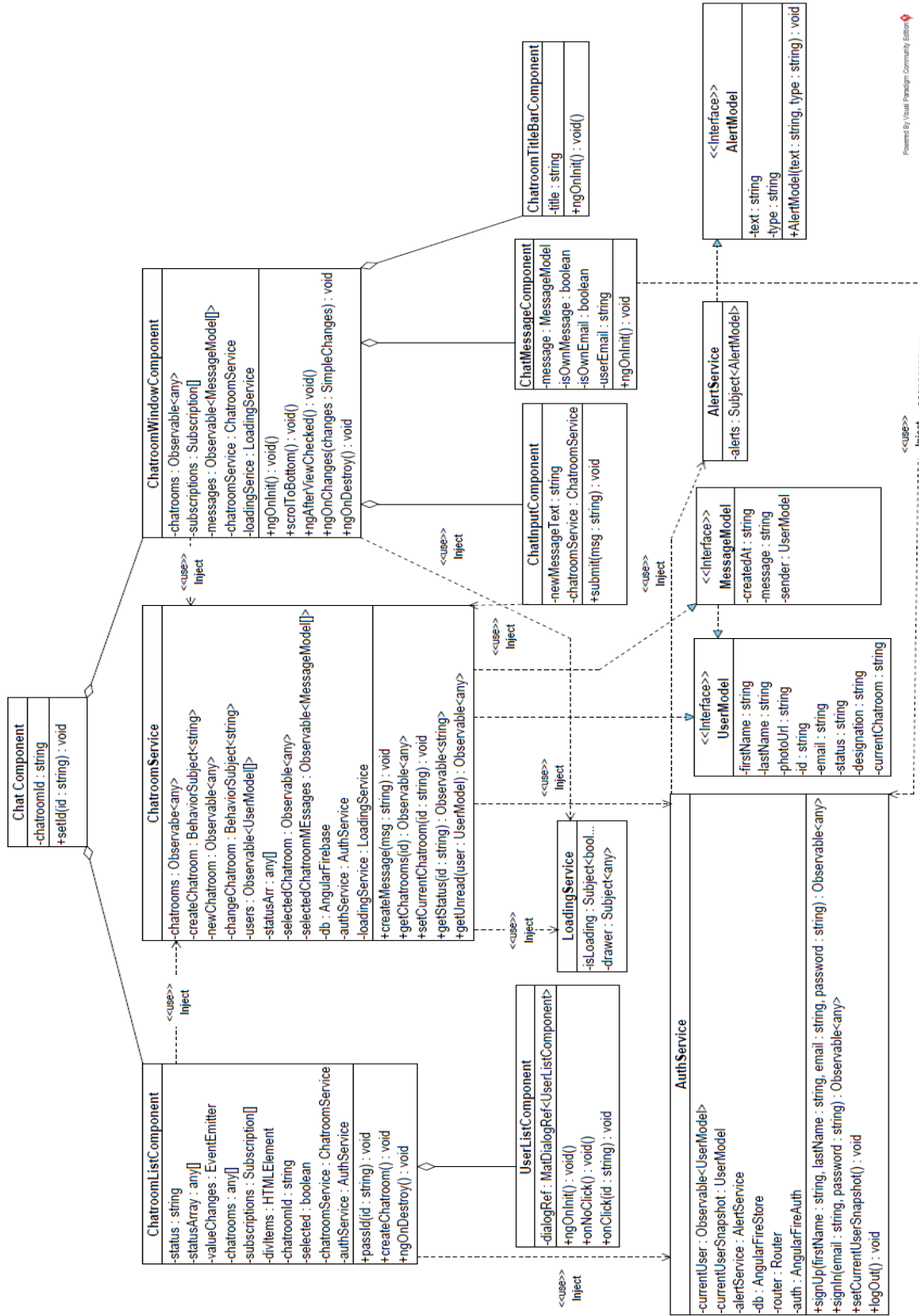


Figure 14: Class diagram of the chat functionality

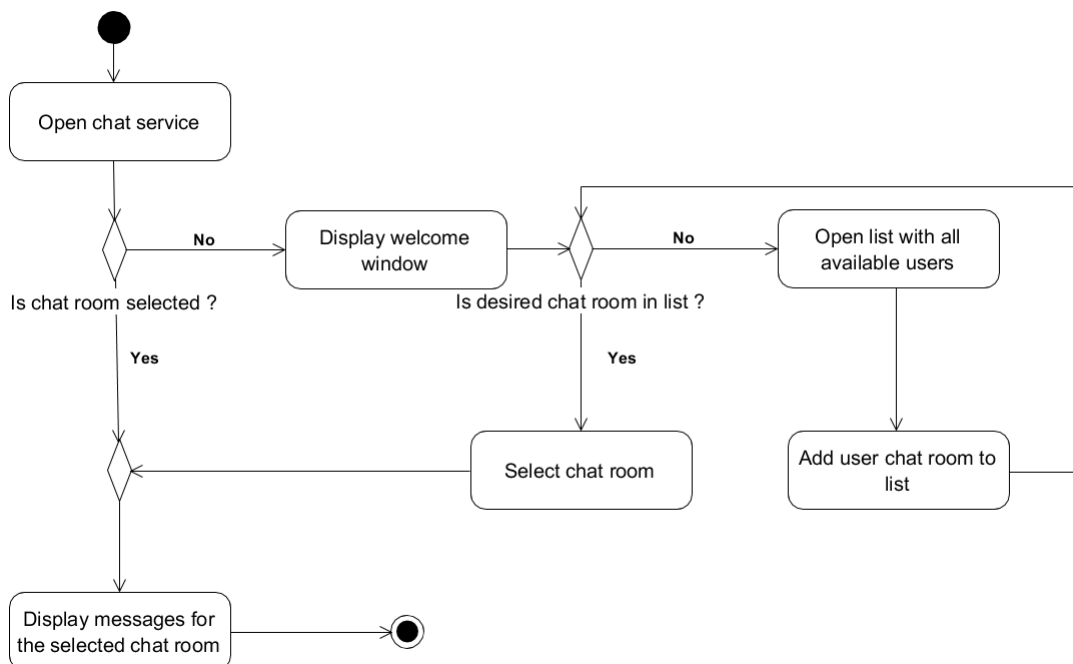
The class diagram displayed in **Figure 14** shows a more detailed representation of the component structure and the links between the components and services used for the creation of the chat service.

The *ChatroomService* class provides the main functionality for the chat since all of the HTTP requests that go out of this module are extracted here. Following the *Façade* design pattern which dictates that complex sub-systems need to be encapsulated into a single interface, the main functionality of this feature is encapsulated in this class.

Using the *Façade* design pattern for encapsulating the main functionality into a single interface per feature for this application reduces the learning curve necessary to successfully leverage the subsystem and it promotes decoupling the subsystem from its potentially many clients [11]. Moreover, since services in Angular are built on the singleton pattern and are initialized only once, they can preserve the state of all the local attributes helping to reduce the amount of HTTP requests and improve performance as the user navigates between routes in the application.

#### 4.3.4 Workflow

The chat functionality is a typical case in an application where the user interacts frequently with the program. Moreover, most of the parts of the chat need to be made functional and not just presentational since they are all inter-related and the behavior of the chat depends on all parts working together.



**Figure 15: Activity diagram of the chat functionality flow for displaying messages**

The activity diagram in **Figure 15** shows the possibilities of user interaction with the chat service to start a private conversation with another user. Because of the high level of interconnectivity between the components in the chat service, each interaction of the user with one component, leads to the update and re-rendering of a different component.

To utilize the full reactive potential of the framework the chat functionality was built using the Observer design pattern. Several of the advantages of this approach include:

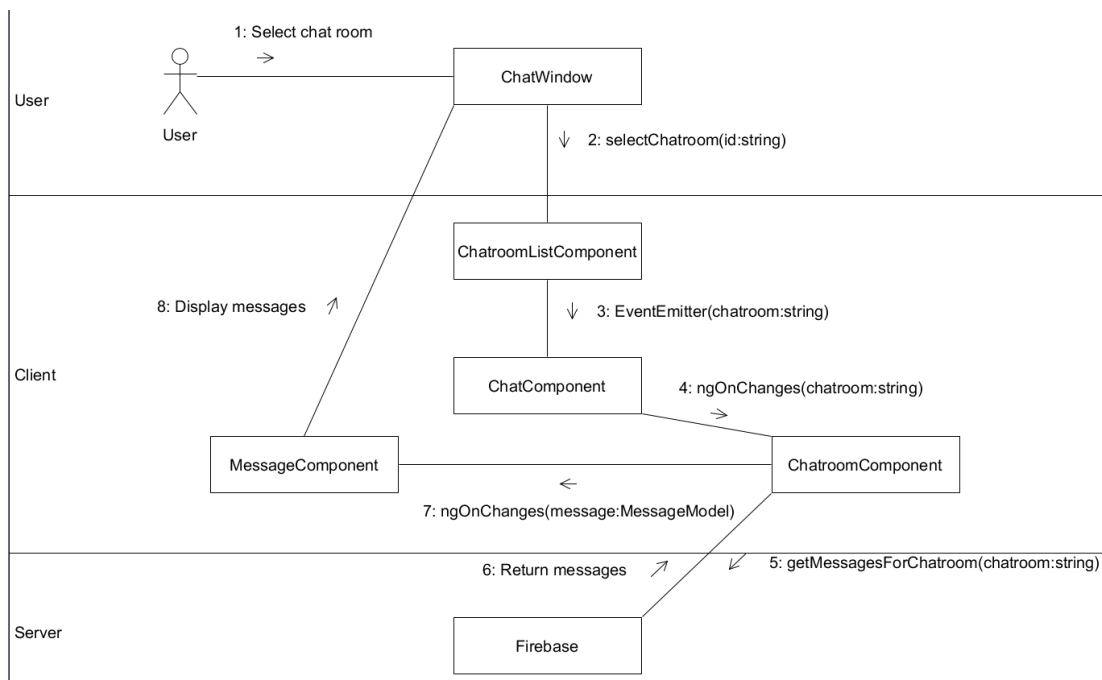
1. Support for broadcast communication – unlike an ordinary request, the notification that a subject sends need not specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject does not care how many interested objects exist, its only responsibility is to notify its observers. This provides the freedom to add and remove observers at any time. It is up to the observer to handle or ignore a notification [12].
2. Unexpected updates - because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects [12].
3. Fluid framework integration – Angular uses at its core the Observer pattern to provide most of its functionality, including data binding, event handlers, forms, router events, etc. Developing a chat functionality involving predominantly user interactions and database communication in this way preserves the integrity of the code base and leverages fully the methods and utilities of the framework.
4. Reusability – This project requires two implementations with two different state management patterns. The second implementation involves the redux state management patter which has a completely reactive implementation in Angular. Building the chat functionality in this manner would allow for reusing and simpler rewriting of the existing code.

**Appendix A** shows the implementation of a single variable controlling the current active chat room using the Observer pattern. It demonstrates how the component creates a subscription to the database through the service.

Firstly, the component creates a subscription to a variable in the service, which in turn creates a subscription to any changes in the database concerning the selected chat room

object. In this way the component stays updated with the current state of the object without the need for explicit follow up requests.

As described in the previous Section 4.3.3 the flat structure of the service is such by design in order to minimize the upstream and downstream communication paths between components. A small example of this type of communication would be the process of loading messages for a particular chat room. In this case the *ChatroomListComponent* and the *MessageComponent* can be found on two different branches in **Figure 13** sharing one common parent component. Therefore, all of the data that needs to be exchanged between those components should flow through the common parent component.

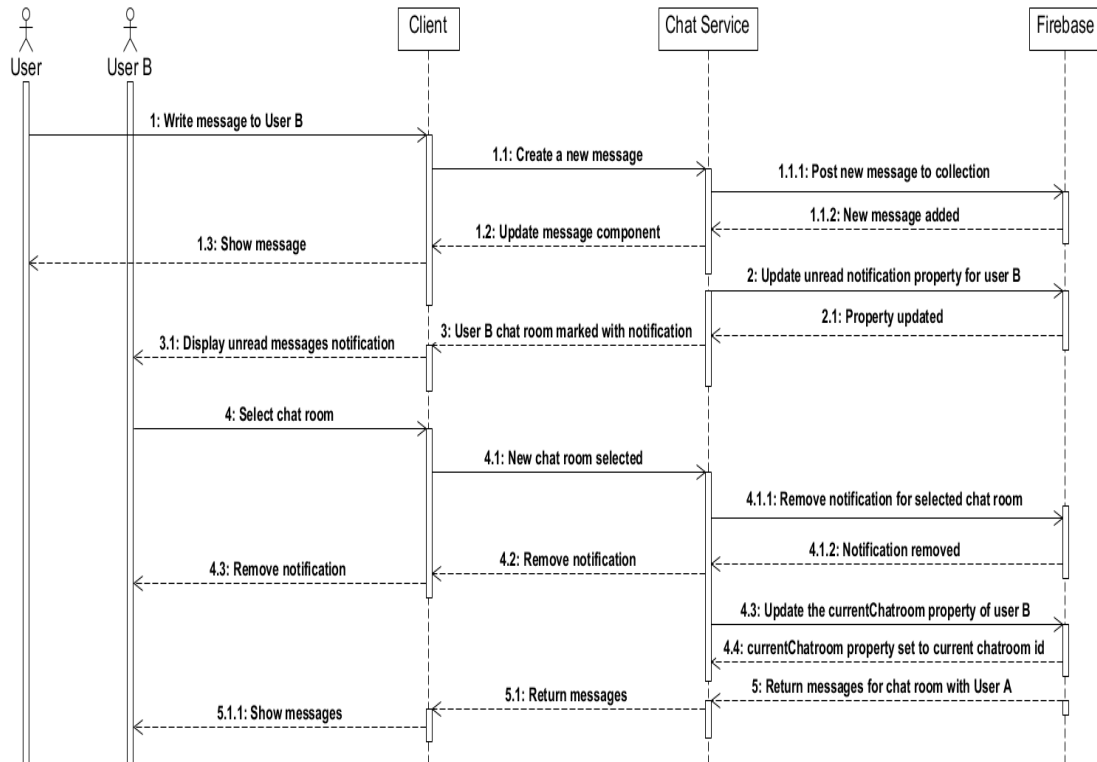


**Figure 16: Communication Diagram depicting communication between two components.**

The common-parent communication strategy is a typical example how most SPA frameworks exchange data between components. Since creating singleton services in React framework is not easy and the framework does not provide such functionality natively, data is exclusively shared between components with common parents through a variable named *'props'* or a third party software is used that offers some central state management pattern.

Another feature built into the chat service is the notification for unread messages. It serves to show the user if he has received messages in any other chat room than the currently active one. An active subscription to all the available chat rooms in the user's list takes care

of this functionality. Every time a user sends a message to a second user, a check is made to see the last active chat room of that user and if it differs from the *id* of the sender a notification is shown.



**Figure 17: Sequence diagram representing the mechanism for unread messages notification**

#### 4.3.5 Data structure

The structural changes related to the client structure discussed in **Section 4.3.2** were resolved with the collapsible design of the chat window. However, the need for a 1-to-1 chat service revealed other difficulties concerning the backend chat models.

The NoSQL nature of Firebase makes the efficient structuring of the chat objects difficult, since it does not support data aggregation, allowing a single query to collect data from multiple tables like regular RDBMS. In NoSQL databases the whole database is represented as a large JSON object. And even though Firebase supports nesting up to 32 levels deep, nesting should generally be avoided since it is costlier. Therefore, having a collection called “messages” where we store all the messages with fields “sender”, “receiver”, “message” and so on would be a poorly made structure since getting the necessary messages for a

chatroom would require the database to iterate over all message to grab only the ones where the sender and receiver values match. For this reason, NoSQL databases like Firebase work best with denormalized data.

In the case of storing the chat messages for the separate chatrooms, a possible solution for flattening the “*messages*” collection would be to append the users’ unique ids on conversation start to form a unique string that would identify a document containing the whole chat history for a particular chat room. However, this strategy would lead to a duplication of all the messages, since the whole conversation would have to be recorded in both chatrooms for both users.

Solving this issue requires a string comparison between the two ids, which is an operation allowed in JavaScript, and appending always the greater value at the end. In this way when a conversation is started it would always have the same unique id, without the need for nesting and data duplication.



**Figure 18: Screenshot showing a possibility for a denormalized chat collection model**

The backend data modeling for the rest of chat service is required to follow the same principles of flatten data structure and composition that allows for fast retrieval of data without the need for querying the database.

#### 4.3.6 Summary

The chat functionality for this project provides the users with the ability to exchange instant messages. During the development process of the project, the initial requirement of the chat services for a simple common chatroom where everyone can post and see all messages evolved to meet the standards and capabilities of other similar applications.

The change in requirements for the chat service however, necessitated changes in other parts of the application to fit the new model. A new behavior of the chat components was

designed to meet the requirements for a separate 1-to-1 chat. The existing component structure was adjusted, together with the backend models for the chat service collections.

The current design of the chat functionality, can serve as a guideline for creating a reactive service in a Single Page Application by utilizing the Observer and Façade design patterns at its core. It is designed as a complementary service that is active through the whole time while a user is logged in, without taking the main focus off of the ticket management system.

## 4.4 Central state management design

### 4.4.1 Additional requirements

An additional requirement for this project is that a second implementation of it is developed, which utilizes a unidirectional data flow. This requirement serves the purpose of producing an application that would have the same functionality, but would utilize the central state management design pattern. In this way a comparison could be made between the two projects that could serve as a guideline for development of modern Single Page Applications.

### 4.4.2 Workflow

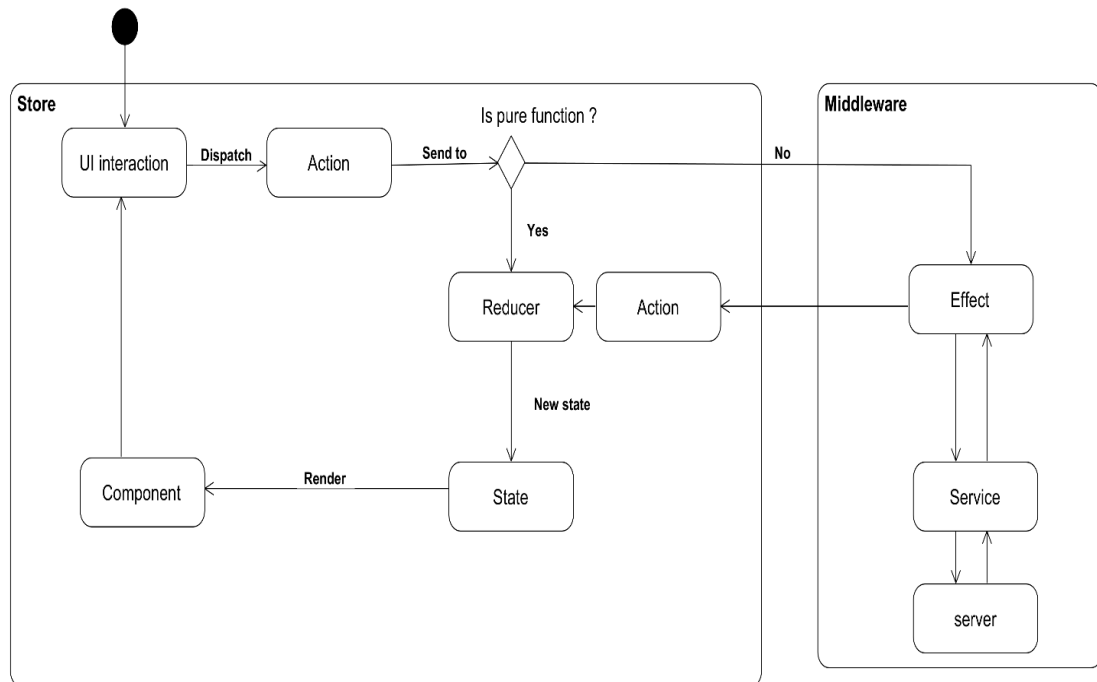
Development of SPAs is often complex because the application work cycle does not stop, and user activity on the application results in changes in the application state on the client. Some changes in the application state come through AJAX, but most of the changes do not come from the server, but from the local variables in the application. Managing the state is one of the most difficult parts of the SPA development in a safe, and time-wise sustainable environment [13].

During the development of the frontend for an application the most critical requirement is that the user interface reflects always the current application state. This means that the state has to change every time the user interacts with the application and this change in state has to be reflected in the UI in turn. Binding state variables to the DOM is the way all major frameworks like Angular, React and Vue provide this functionality. In most simple applications that would suffice to provide a coherent, in-time response of the UI to changes in the state. However, as the application grows and the state becomes more complex, this methodology often becomes inefficient for maintenance and reliability issues may occur.

The central state management pattern is intended to synchronize the state changing operations, providing in the end always a predictable way of mutating the state object. Moreover, since the reducers, which are the state mutating functions are built in an immutable way, the change detection algorithms can track the changes correctly and

update the state on the screen. The main concept of the centralized state management pattern is that it provides a unidirectional data flow.

By utilizing a unidirectional data flow, as a main concept in its architecture and the immutable update patterns, the central state management pattern intends to make the central state storage the single source of truth (SSOT) [14] for the application. Therefore, all the changes relevant to the UI can be synchronized and the UI can always be an image of the latest state object.



**Figure 19: A flow chart representing a central state management system design**

**Figure 19** illustrates the unidirectional data flow that is implemented in the central state management pattern called Redux.

The implementation of the Redux pattern does not require on its own any restructuring in the project's component architecture, it requires however a significant change to how the data flows through the components and the way the data is being handled.

The three key concepts about the Redux pattern are:

1. **Store** – there is a central place in the application called “Store” that holds the whole state of the application. The store is considered the single source of truth



and can be injected into any component, providing it with access to the latest object representing the application state.

2. **Reducer** – Reducers are pure functions, since they are not allowed to have any side effects such as mutating local variables, sending HTTP requests, etc. They define how the state will look like after an action has been dispatched, by transforming the current state in an immutable way. In this way the state object is never really mutated, instead a new copy of the state is always provided that contains the latest changes. This allows features such as time travel debugging and ensures that change detection will work properly since the reference of the object will change.
3. **Unidirectional data flow** – the flow of information from the UI to a *reducer* that modifies the state, which in turn is passed back to the UI is an important feature of the Redux pattern. In this way, it is ensured that the information flow is predictable and easy to trace.

Furthermore, in Angular the Redux pattern can utilize *Subjects* and *Observables* [15] for its functionality, allowing the user to treat its variables as streams to which the *store* can react appropriately over time. Therefore, building the chat functionality and the ticket management system with the Observer pattern facilitates an easier and better transition to the second implementation where the *store* will be handling the state of the application.

As mentioned in the beginning of this section, the Redux pattern does not require necessarily project restructuring, however for its implementation it requires a lot of boilerplate that needs to be implemented in order to secure the unidirectional data flow.

Part of this boilerplate code includes:

- **Actions and action creators** - the first building blocks which need to be designed when composing a Redux type of architecture are the *actions* and the *action creators*. While the *action creators* are the interfaces which create and return an action, the *action* itself is what the components use to notify the *reducers* that the state of the applications needs to be modified. The *action* works as a message that can be dispatched with the new information with which the state should be altered or they can serve as a notification that a function needs to be triggered.
- **Effects** – Also known as *side-effects*, provide means for the application to handle asynchronous events such as HTTP requests, server communication or other type of event which are not pure functions in a safe and predictable way. In the flow of the

---

Redux pattern they serve as middleware, listening for specific actions dispatched from the store to which they can react.

- **Selectors** – these are helper functions that allow the user to have quick access to particular parts of the state. Since the Redux pattern creates one big JavaScript object that represents the state of the application, the selectors are needed to extract efficiently smaller piece of data relevant for the separate components.

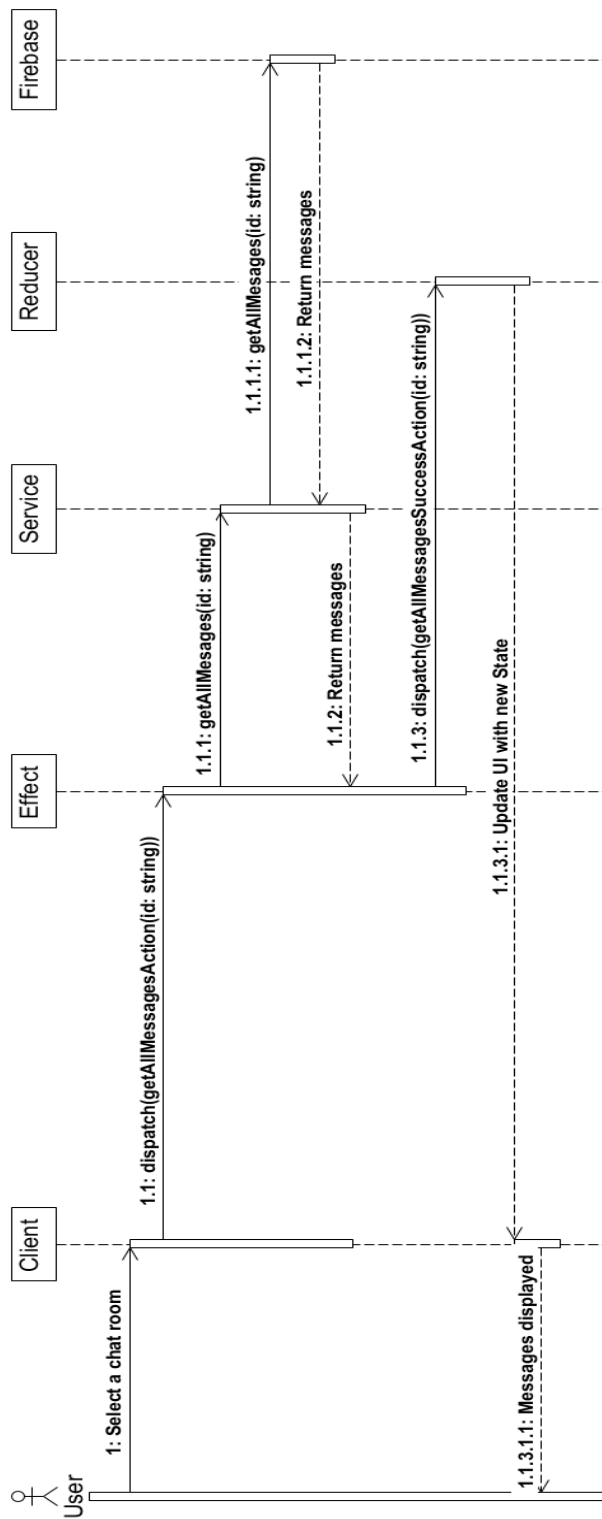
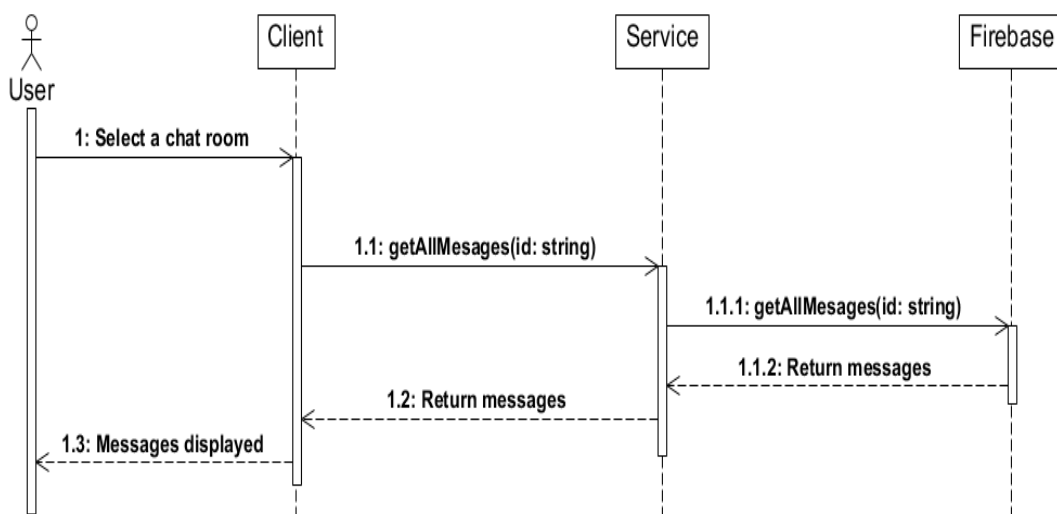


Figure 20: Sequence diagram showing a Redux implementation of a chat service function

**Figure 20** shows a sequence diagram representing the design of a single function from the chat service that uses the Redux pattern. The function, which gets the messages from the database after the user selects a chatroom, needs to go through all of the Redux middleware implementation in order to preserve the unidirectional flow of data. Since the function needs to reach out of the Redux environment to the database and perform an HTTP request, this request must be processed by an *effect* to make sure that the right information is being passed to the *reducer*. This *effect* will handle any errors if present and would dispatch the correct action once the information is there and the *reducer* can work with it in a synchronous manner.



**Figure 21: Sequence diagram showing an implementation of a chat service function**

**Figure 21** on the other hand, shows the same function design without the Redux pattern. The comparison between the two sequence diagrams reveals a much simpler data path for the second implementation. However, it can immediately be recognized that the messages are received by the client and the business logic is performed inside the client to transform this messages into the appropriate application state. Consequently, as the application grows and multiple components get nested within each other, the application state modifications will get harder to track down. Moreover, in cases where multiple components need access to the same state and can update this state in the same time, it becomes very difficult to manage this process.

While the Redux pattern allows for the containing of the application state, making the store the single source of truth for the application, it enforces simultaneously additional

complexity on that project. The library that provides the main implementation of Redux for Angular is built on top of another library that provides Reactive Extensions for JavaScript called RxJS. Even though this is the same library on which Angular is built and which allows the Observer pattern implementation of the services in this project, it forces the user to use reactive programming when this might not be necessary.

This additional complexity, together with the large amount of boilerplate code are the two main drawbacks of the Redux pattern. Therefore, it might be the case that for smaller and middle-sized applications this is not a desirable pattern to be used. However, since it requires a complete rewriting of every function that alters the application state, it is best to decide in the design phase of an application whether to use the pattern or not.

The benefits provided by Redux are not negligible at all. Securing a unidirectional data flow and a centralized application state can be extremely beneficial for sustaining maintainability and high performance of the application as it grows. Redux provides best practices guidelines that help with error handling and fixed rules to help improve the development process. Moreover, the store could be used as a cache. If other features are designed, such as offline accessibility of the application that requires a portion of the state to be stored in the Local Storage of the application, it would make sense to hydrate that state object in the Local Storage for offline use from the store.

Lastly, to improve the performance of the application, the store could be leveraged to provide *optimistic* UI updates [16] for an improved, snappier user experience as compared to a traditional “loading” experience. By displaying data immediately upon navigation change from the store, which can possibly be outdated, while requesting the latest version of that data from the backend, the application will provide a feeling of responsiveness and maximize in this way the user experience. This strategy is not applicable in every situation but is a possibility provided by a centralized management system.

#### 4.4.3 Summary

The central state management system represented by the Redux pattern provides an alternative for handling the state of an application. It deals with the flow of data through the application, imposing a unidirectional data flow. In this way, the UI can be updated in a safe and predictable way without any complication for the change detection mechanisms. On the other hand, the cost of implementing Redux in an application includes added complexity, a large amount of boilerplate code and therefore more time for development. **Section 5.3** shows the implementation of the data store and how this affects the data flow.

# 5 Implementation

## 5.1 Authentication

The implementation of an authentication process in this project is based on Angular's route guard interfaces. There are several types of guard interfaces, but for the needs of the project a *CanActivate* [17] interface has been implemented that can decide if the requested route comes from an authenticated user or not. Consequently, if a user who is currently not logged in attempts to call the *board* or *backlog* routes will be redirected to the login window.

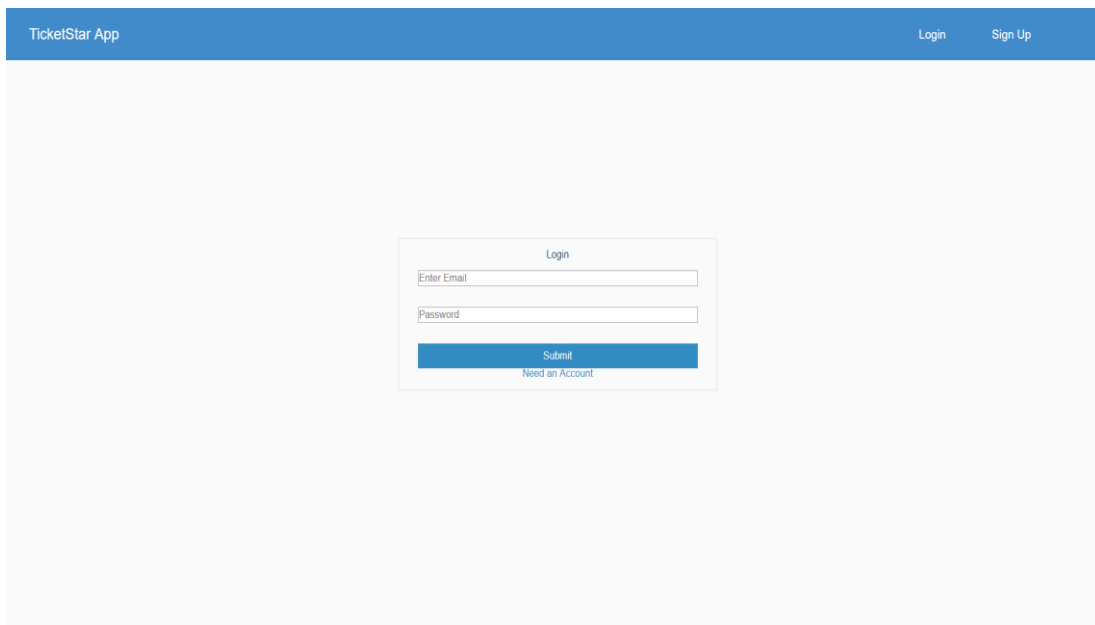
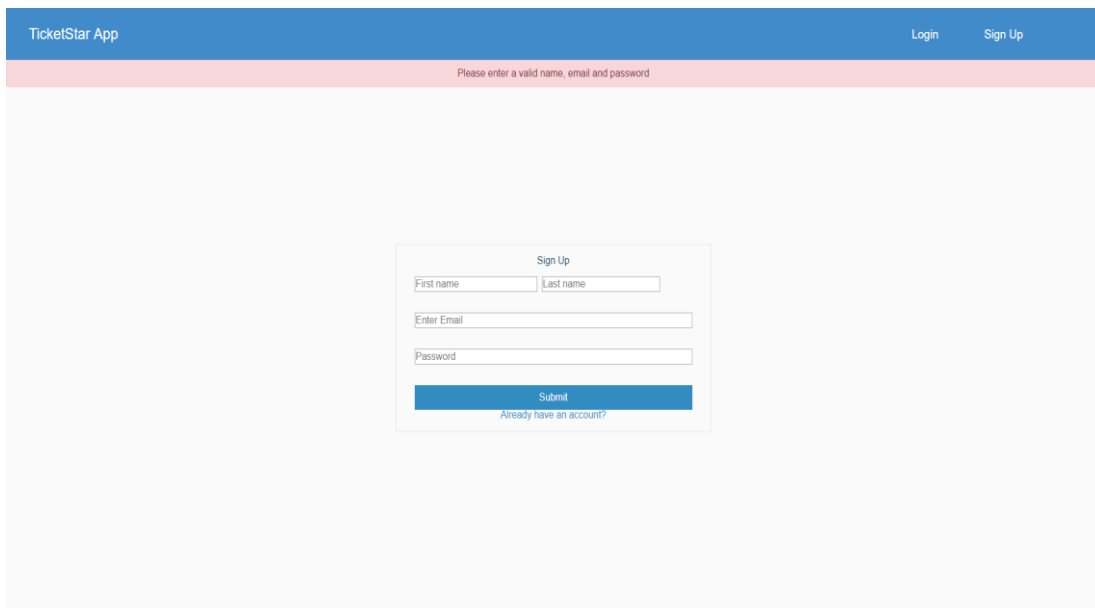


Figure 22: Screenshot of the UI representing the Login component view



The screenshot shows the TicketStar App interface. At the top, there is a blue header with "TicketStar App" on the left and "Login" and "Sign Up" on the right. Below the header, a pink alert bar displays the message "Please enter a valid name, email and password". The main content area is white and contains a "Sign Up" form. The form has the following fields: "First name" and "Last name" (two separate input boxes), "Enter Email" (one input box), and "Password" (one input box). Below these fields is a blue "Submit" button. Underneath the button, there is a link that says "Already have an account?".

**Figure 23: Screenshot of the UI representing the Sign Up component view with a wrong validation attempt alert**

For the purposes of logging in and registering, two components were created with the appropriate form. Both views share similar features and they work internally the same way. If a user creates a registration or logs in successfully, he is redirected to the main page of the application and the chat functionality is activated. However, if the user does not fulfill the criteria of the form validation or tries to log in with wrong credentials, an alert component is triggered that notifies the user appropriately.

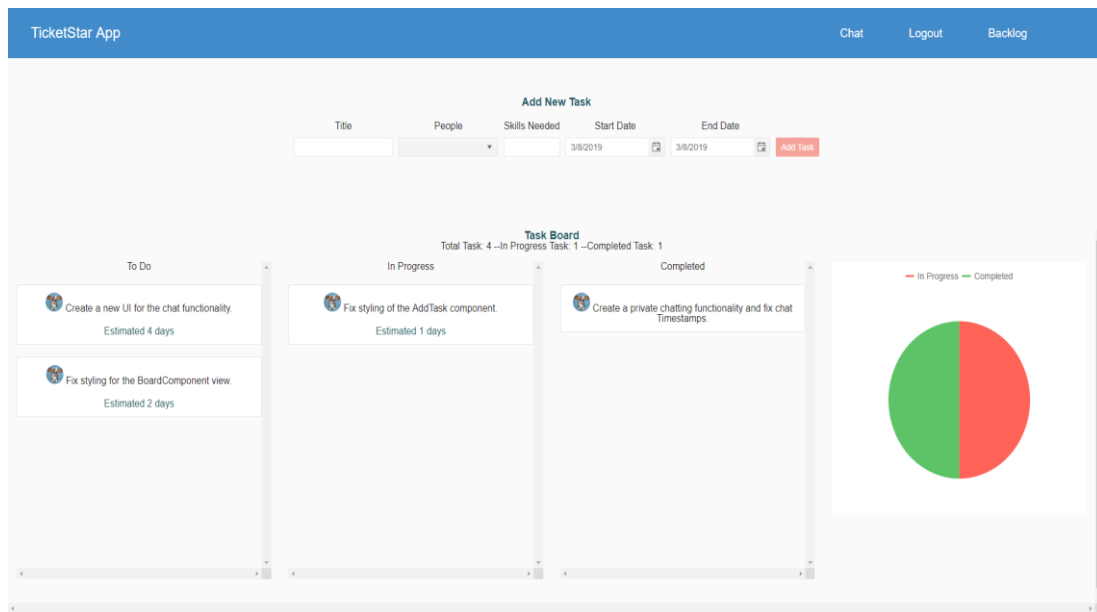
## 5.2 Ticket management system

The ticket management board implemented in this project provides a good range of functionalities for displaying and working with tickets. It supports several functionalities including:

1. Providing a tool for ticket creation.
2. Allowing the user to assign a person responsible for the ticket, skills required for the ticket and a time estimation for which the ticket needs to be completed.
3. Assuring ticket state synchronization between users. The ticket creation, causes a change in the ticket state of the application which in turn triggers a re-rendering

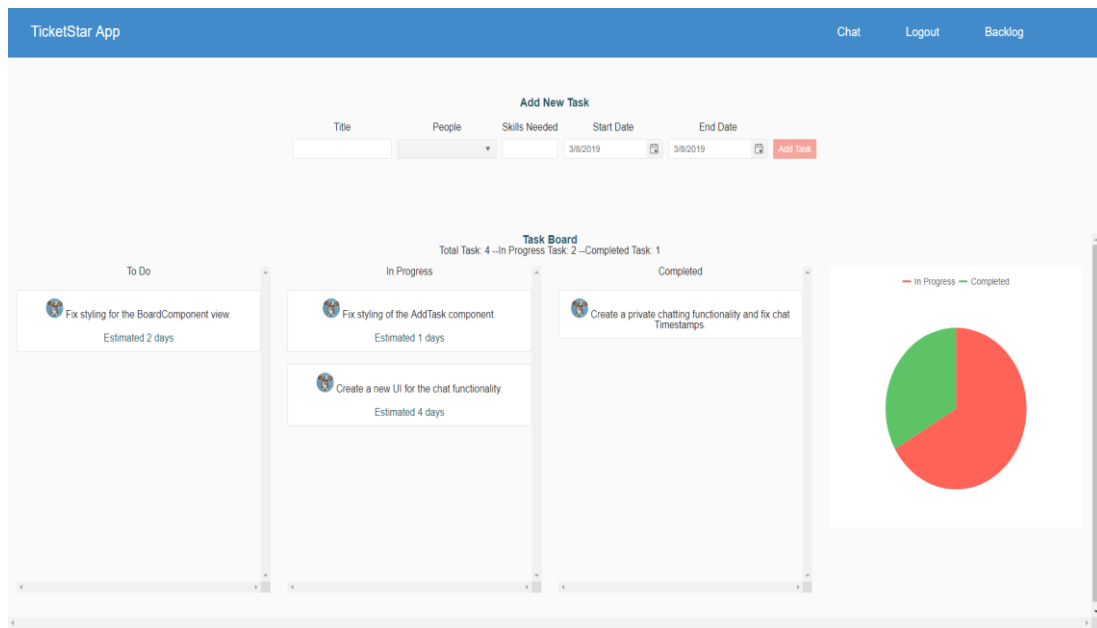
of the UIs of all people currently visiting this page without the need for refreshing the window.

4. Drag and drop functionality – The tickets can be dragged between columns and dropped in a desired column depending on the state of the ticket's progress. This updates the pie chart on the side and the heading of the board that provides a summary of all tickets.
5. Ticket editing functionalities, including editing of the ticket content, deleting the ticket or sending the ticket to the backlog.
6. Visual summary of the tickets state. The board contains a pie chart providing a visual representation of the ratio between tickets in progress and the ones already completed.



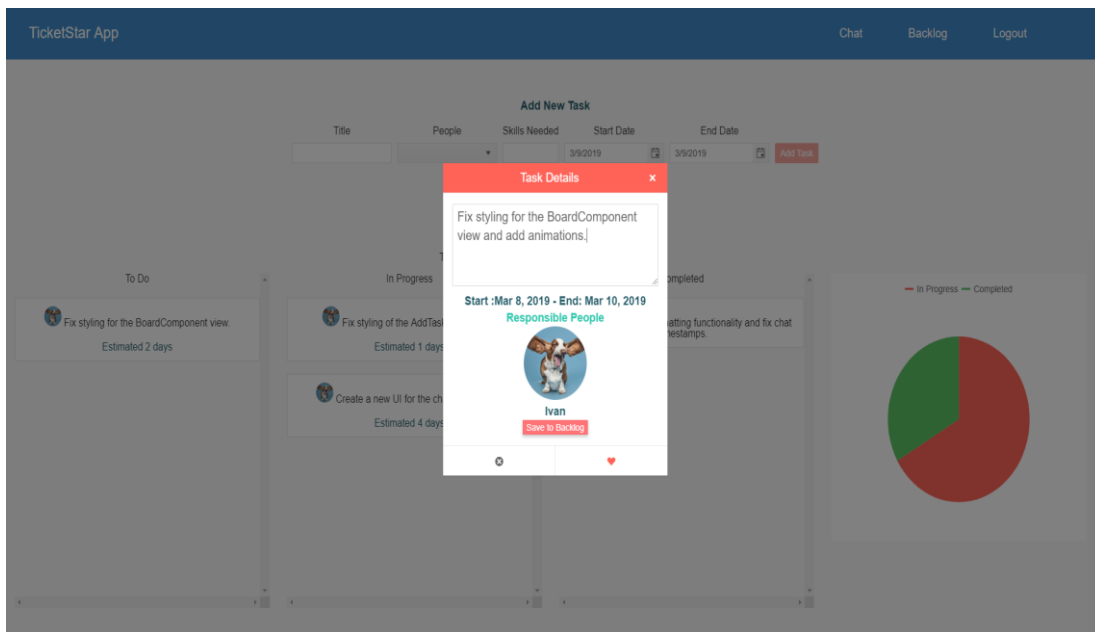
**Figure 24: Screenshot of the board component UI illustrating the main page view**





**Figure 25: Screenshot of the board component's UI illustrating the drag and drop functionality of the tickets**

The screenshot shown in **Figure 25** illustrates how a ticket can be moved from one column to the next, updating the summary on top of the board and the pie chart on the side. These changes are synced across all users simultaneously through the backend. No manual refreshing of the page is required to get the latest state of the objects.



**Figure 26: Screenshot of the ticket editing component's UI illustrating the available editing options**

**Figure 26** provides a screenshot of the editing tickets component which allows the user to edit the ticket's content as well as delete the ticket or send the ticket to the backlog through pressing of a button. The component uses an overlay object which covers the rest of the view and allows a single component to be active. As soon as the changes are saved, the view is synced across all users.

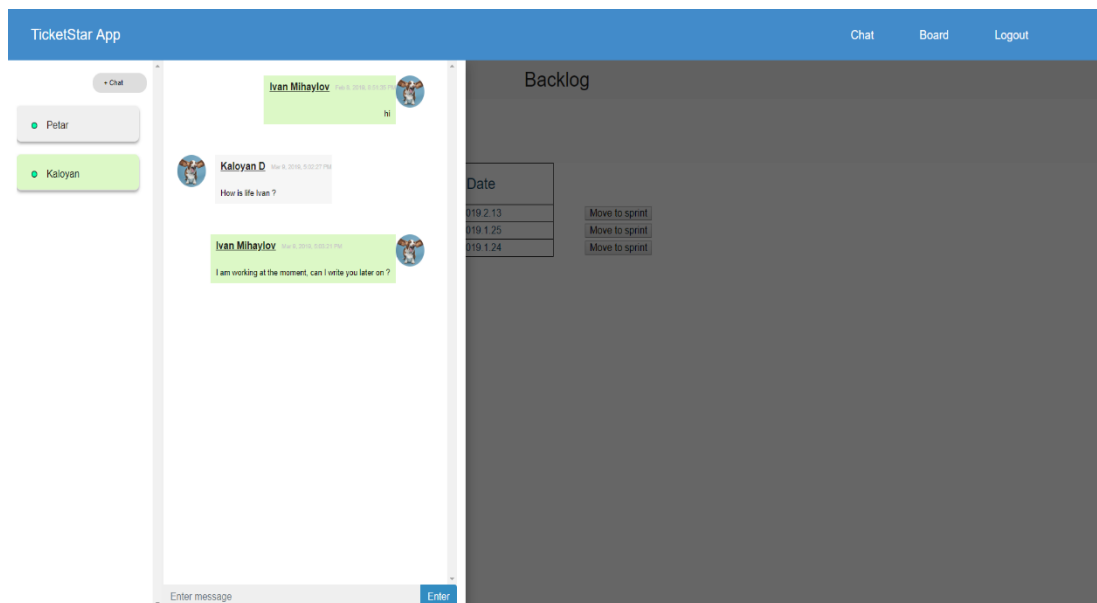
TicketStar App					Chat	Board	Logout
Backlog							
#	Summary	Assignee	Labels	Date Created			
1	New Task	Kaloyan D	Angular 6	2019.2.13	Move to sprint		
2	FinalTask	Kaloyan D	Angular 4	2019.1.24	Move to sprint		

**Figure 27: Screenshot of the backlog component's UI**

The backlog component implementation, captured in **Figure 27** shows a table structure representation of the tasks. It provides the users of the app with a container where tasks that would be postponed could be saved. Finally, it gives the user the possibility to move the backlog task from the backlog into the board component on demand.

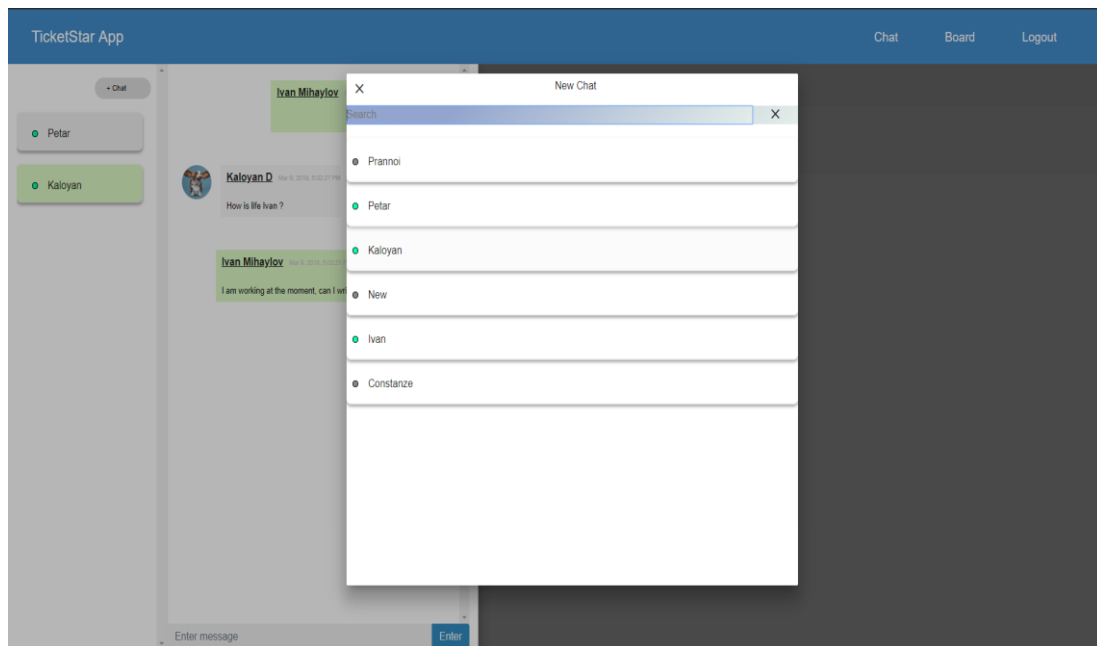
### 5.3 Chat Service

The implementation of the second feature of this application, namely the chat service is presented in **Figure 28**. The chat window opens up after the user clicks on the chat button on the navigation bar. The chat service is already active after the user login has completed and awaits the user to select a chatroom to start a conversation. The user is shown as online to all other users and can also see which users are online from the green marker in front of the chatroom. By selecting a chatroom, the user can see the messages left in this chatroom and can start typing new messages on demand.



**Figure 28: Screenshot of the chat functionality UI showcasing a chat conversation**

However, if the desired chat room is not present in the user's list, he can add a user to his list from the dedicated button found on top of his chatroom list "+ Chat". Selecting this button, opens up a window, that presents the user with a list with all available users from which he can select to be added to his chatroom list. **Figure 29** presents this scenario.



**Figure 29: Screenshot of the users list UI illustrating the option for adding a new user to the chat room's list**

**Figure 30** below shows the mechanism for displaying a notification for the unread messages. The flag is set every time a user receives a message in a chat room that differs from the currently active chat room of that user. This flag signals the client to display a notification which appears in two separate places in the user's interface. Since the only active component that is visible on all navigation paths is the navigation bar component, one of the notifications is set on it. The second notification appears in the chat room list component on the chat room element itself.

The clearing mechanism of the notifications works in a similar way. A notification is cleared from the chat room element if a user selects this element. The second notification is cleared if there are no chat rooms in the list of this user that have unread messages.

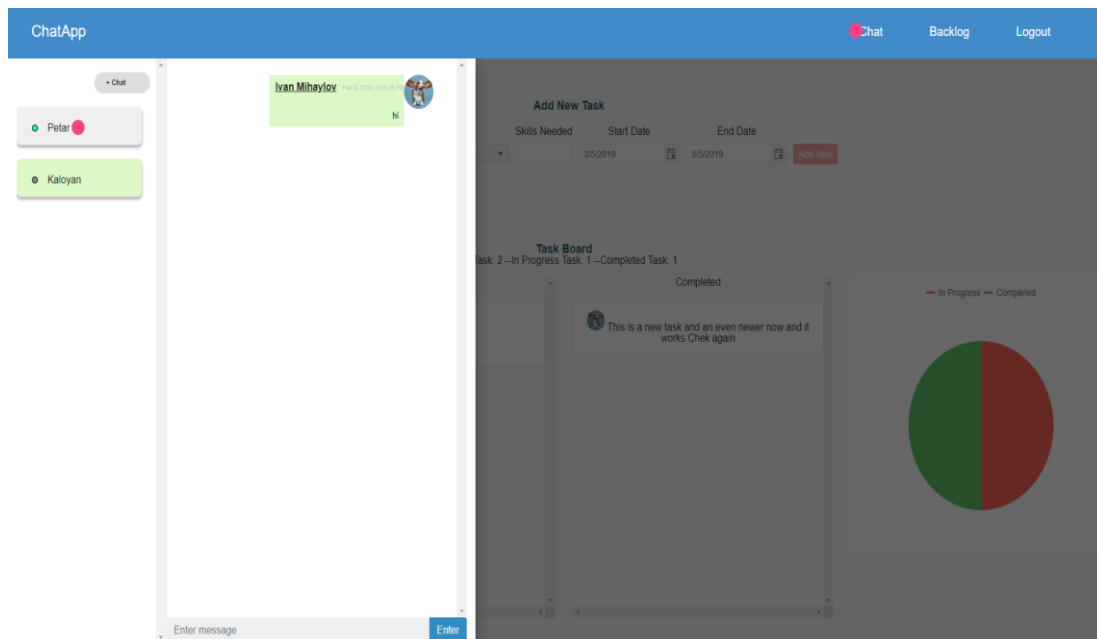


Figure 30: Screenshot of the chat service UI illustrating the unread messages notification implementation.

## 5.4 Central state management implementation

The Implementation of a central state management system does not come with changes to the user interface as described in **Section 4.4.2**. However, the Redux pattern introduces a new form of state management that is allowed through a central store, reducers and the implementation of middleware.

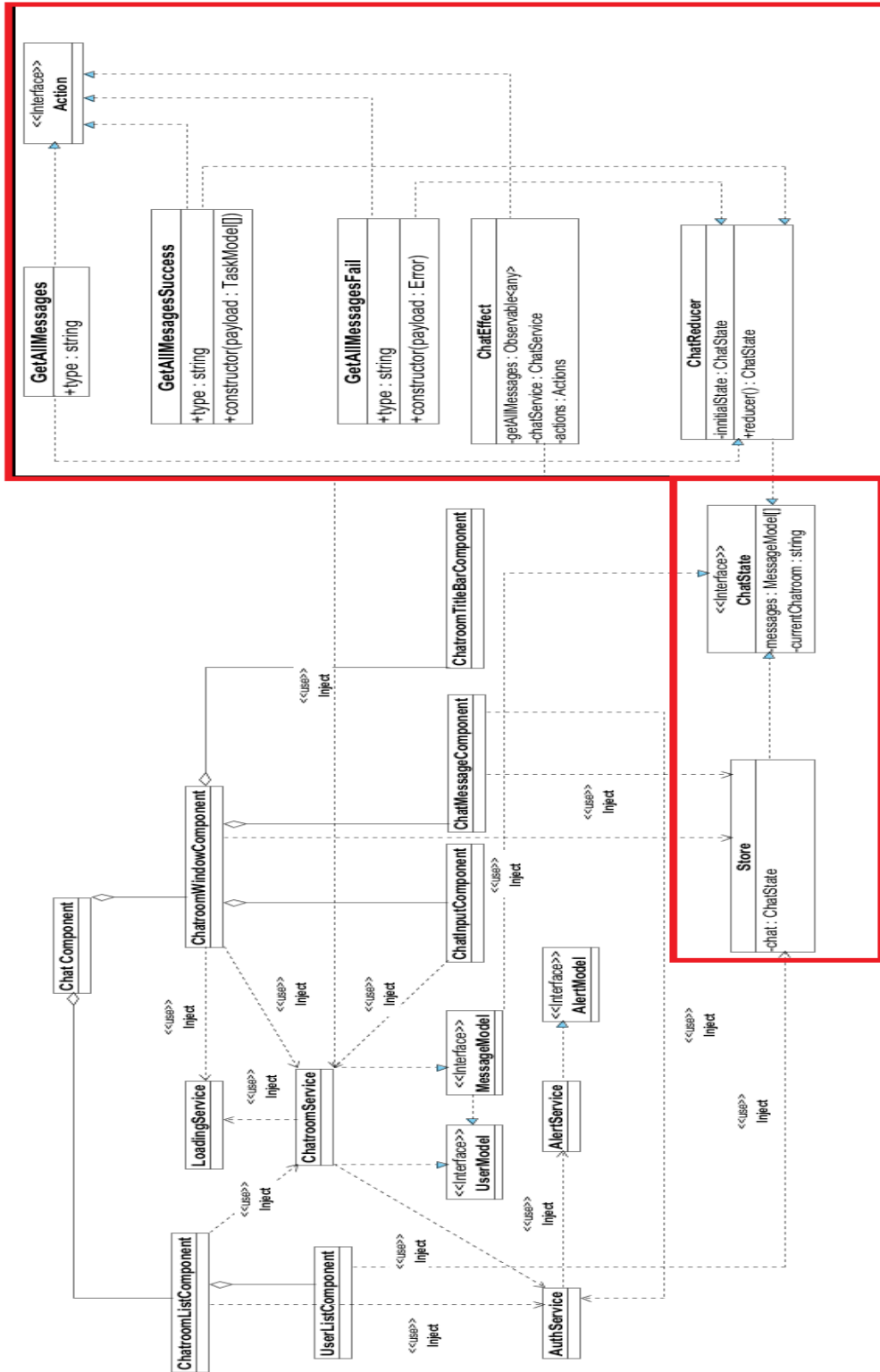


Figure 31: Class diagram of the chat service with Redux implementation

The class diagram on **Figure 31** shows the design of the Redux pattern in handling the state of the chat service. The classes of the chat service itself are left unfilled for readability purposes. Furthermore, following the example of the sequence diagrams in **Figure 20** and **Figure 21** showing the Redux implementation of a single function, the part of the class diagram highlighted in red boundaries represents the Redux middleware which needs to be included for the implementation of a single function.

In order for a single function, which makes an HTTP request, to be implemented using the central state management pattern, three separate *action* classes have to be created every time. Moreover, the listeners for these actions have to be created as well. Either an effect or a reducer has to react to these actions which signify an intend to change the state of the application. In this way the system can guarantee the unidirectional data flow and the immutability of the application state changes.

Since the ticket state and the chat state are independent of each other and the rest of the application state, they can be extracted each into a separate *reducer* that would manage the complete state for that feature. The implementation of the ticket state reducer could be found in **Appendix B**.

In a similar way as the Angular framework allows the encapsulation of a particular function inside a component and then extends this encapsulation mechanism to a modular level for a separate functionality of the application, we can modularize the reducers in the central state management system.

By providing a separate reducer to handle the ticket state and the chat functionality state we can compose our application state from both reducers by extending the abstraction further and creating a top level reducer that would compose all his children in an appropriate manner.

```
import { ActionReducerMap, createFeatureSelector } from '@ngrx/store';
import * as fromTasks from './task-reducer';
import * as fromChat from './chat-reducer';

// top level application state
export interface ModulesState {
  tasks: fromTasks.TaskState;
  chat: fromChat.ChatState;
}

// Top level reducer
export const reducers: ActionReducerMap<ModulesState> = {
  tasks: fromTasks.reducer,
  chat: fromChat.reducer
};

export const getModulesState = createFeatureSelector<ModulesState>('modules');
```

**Figure 32: Screenshot of a code snippet, representing the top level reducer**

The last step after creating a top level reducer for a particular feature or application is to create a *selector* for it as well. In this way the Redux pattern provides full flexibility in its implementation when it comes to handling application state.

### Overview

As a consequence of the added complexity and the large amount of boilerplate, a valid argument could be made that introducing a central state management system into an application could be counterproductive in many cases. Therefore, extracting separate functionalities into feature modules could allow developers to introduce a Redux environment only for certain features where state handling becomes otherwise unmanageable or very hard to manage. In this way, a hybrid application state management system can counteract the growing complexity of an application in a much more efficient way.

Since the current project at this stage of development does not require many of the features that are made possible with the implementation of a central state management system, the overall addition of complexity and the amount of extra code required for its implementation could not be appropriately justified. The Redux implementation however, could be a good stepping stone for developing the future features of this application.



# 6 Testing

## 6.1 Unit tests

All Single Page Application platforms provide tools with which the developers can perform unit tests. Every created component or service in Angular comes with a file that has a *.spec* extension that holds the relevant tests. The testing tools provided by Angular in the case of this project are named Karma and Jasmine [18]. Testing a single unit of work can become complicated if it has external dependencies. To deal with this issue, a mocking framework is used, which abstracts the external dependencies. This approach is very successful, because all dependencies are supplied through dependency injection [19].

**Appendix C** shows a unit test, which tests the dynamic adding and removal of HTML tags. The adding of the HTML tag, which provides the notification for unread messages into the DOM tree, is based on a response from the chat service. The service tries to retrieve the *unread* property of all chatrooms in the list of the current user and based on the response of this request an additional HTML tag is added or removed from the DOM. Since the response of the service, relies on Firebase which is not accessible in this testing environment, the call and response to it are mocked. The mocked response returns an already defined value for the *unread* property with which the testing platform checks whether or not the element is part of the DOM.

## 6.2 Manual testing

### 6.2.1 Authentication and board component

The methodology used for testing the correct functionality of the application involves mainly a set of scenarios and verifications that are conducted manually. The verifications are specified depending on the features tested, these involve evaluating visually the UI controls at runtime and in some cases inspecting the state of critical variables while debugging the application [20]. The testing scenarios are intended to cover all of the functional requirements described in **Table 2**.

Test sequence	Expected result	Outcome	Functional requirements references
<ol style="list-style-type: none"> <li>1. User tries to sign up without filling all of the fields out.</li> <li>2. User tries to sign up with invalid credentials</li> <li>3. User tries to sign up with credentials that are already in use</li> </ol>	Fail to sign up	Fail to sign up and display an alert with a message	IM2, IM4
<ol style="list-style-type: none"> <li>1. User fills all the fields correctly and submits the form</li> </ol>	Create registration and navigate to main page	Create registration and navigate to main page	IM3, IM7
<ol style="list-style-type: none"> <li>1. User tries to log in without filling all of the fields out</li> <li>2. User mismatches the username and password for his account</li> </ol>	Fail to log in	Fail to log in and display an alert with a message	IM8, IM9,
<ol style="list-style-type: none"> <li>1. User correctly inputs his login credentials</li> </ol>	Successful login and redirection to main page	Successful login and redirection to main page	IM6, IM8
<ol style="list-style-type: none"> <li>1. Navigation bar items change if authentication is successful</li> </ol>	Login and Signup are hidden after login and Chat, Backlog and Logout appear	Login and Signup are hidden after login and Chat, Backlog and Logout appear and alert with a message appears	IM10
<ol style="list-style-type: none"> <li>1. Main navigation paths are hidden if user is not logged in or immediately after he clicks on log out</li> </ol>	Main navigation routes are hidden and unavailable. Only Sign Up and Login paths are shown.	Main navigation routes are hidden and unavailable. Only Sign Up and Login paths are shown.	IM5, IM10
<ol style="list-style-type: none"> <li>1. User enters the ticket board component</li> </ol>	Be able to see all tickets	All tickets are present	IM11

1. User creates a ticket without filling all of the fields out	Fail to register a new ticket	Fail to register a new ticket and alerts shows a message	IM11
1. User creates a ticket and fills all fields out	A ticket is posted in the To Do column and all users can see it	A ticket is posted in the To Do column and all users can see it	IM11, IM12, IM13
1. User selects a ticket without dragging	A ticket is opened for editing	A pop up appears that allows editing functionality on the ticket	IM14
1. User presses a delete button on the edit ticket pop up window	Ticket is deleted for all users	Ticket is removed from the board for all users	IM15
1. User edits the ticket and saves it	Changes on ticket are saved	Ticket appears with changes to all users	IM14
1. User saves ticket to backlog	Ticket is removed from board and send to backlog	Ticket is removed from board for all users and appears in the backlog path for all users a message is displayed	IM18
1. User selects ticket and drags it to the next column	Ticket is moved from current column to the target column	Ticket is moved to the new column for all users	IM16
1. Selecting window refresh button on the main board page	The state before the refresh should be the same as the one after the refresh if no other user has made changes	The state of the tickets is persisted after refresh	IM19
1. User navigates to backlog	Backlog tickets are displayed	Backlog tickets are displayed	IM18
1. User selects "Move to sprint" button next to a	Backlog is sent to Board Component	Ticket is removed from the backlog	IM18

given backlog task		list. Ticket appears in the To Do column for all users	
--------------------	--	--	--

**Table 4: Functional requirements tests for the board component**

**Table 4** provides a summary of manual testing of all the functional requirements that relate to the board component functionality. During the testing procedures, critical variables were inspected for correctness using the Chrome debugger and the state of the database was supervised for correct behavior as well.

### 6.2.2 Chat service

The chat service functionality is tested using a predefined sequence of actions design to test the full range of features for this service. For this purpose, two windows are opened both of which connected to the internal Angular server running the application and a chat simulation between User A and User B is performed. During the sequence of actions all messages were transmitted and every action was reflected correctly in the database. The relevant chat notifications were shown and after restarting the application, the users could see their respective chat history correctly.

Since the chat service was developed to have different features than initially planned, the functional requirements for it are missing in **Table 2**. Therefore, the tests performed to test the correct functionality of the service do not have a reference to the “Requirements analysis” chapter.

Action sequence	Expected result	Outcome
User A tries to access the chat path before authenticating	User is redirected to login	A notification is displayed that user is not authenticated and user is redirected to log in screen
User A successfully authenticates	User A get access to the Chat functionality	User A get access to the Chat functionality
User A selects the chat tab in the Navbar component	Chat window opens	Chat window opens
User A selects the “+Chat” button to add a new user	Window containing all registered users appears	Window containing all registered users appears
User A selects User B to	User B is added as	The pop up window closes and User B

be added as a chatroom on the new window	a chatroom in User A's list	appears as a chatroom in User A's list
User A selects User B be added as a chatroom on the new window even though User B is already added	Nothing changes	Window with user list is closed and the list does not change. User B is not added for a second time as a chatroom.
User A selects the User B chatroom from his list	All messages appear in the chat window	<ul style="list-style-type: none"> <li>- User B chatroom gets marked green as selected</li> <li>- chat history appears ordered by time</li> <li>- chat messages are styled differently based on the sender property</li> </ul>
User A sends a message into User B chatroom	User B receives a chat message	<ul style="list-style-type: none"> <li>- if User B does not have User A already added as a chatroom, a notification is send to User B that he has unread messages and User A gets added automatically to his chatroom list.</li> <li>- if User B has user A in his chat room list but he is currently using a different chat room or has the chat window closed, a notification is send that he has unread messages and User A chat room gets highlighted.</li> <li>- if User B has User A in his chat room list and the chat room is currently selected, User B can see a received message from User A</li> </ul>
User A closes the chat window	Chat is closed	Chat is closed
User A navigates to the backlog component and opens the chat	Chat is opened	Chat is opened and the messages from the last selected chat are displayed.

**Table 5 Functional tests for chat service**

# 7 Conclusion

The work in this project provides an implementation of a Single Page Application using two different application state models. While the UI has been kept identical for both applications, one of them follows the Redux state management pattern to organize and distribute the application state, while the second application utilizes dependency injection and the parent-child component relationship to achieve this goal. Both state management patterns have been examined theoretically and examples have been provided from the project implementation. Moreover, to make the applications complete, a serverless backend has been introduced, which comes with a real-time database for the data persistence.

The main challenges faced during the work on this project include, creating a good structure and designing an application that has two different implementations. Since, the work on this application can continue in the future, a good structure, ensuring the extensibility of the project was required that would facilitate the implementation process of new features. Furthermore, for the purpose of comparing the two state management patterns the design of the application had to allow for a smooth transition, which would not require a complete revision of the codebase. Both of these challenges have been examined in depth in **Chapter 4** of this thesis.

In the process of developing the Chat and Ticket management system a deep understanding of frontend state management patterns was achieved. Moreover, extensive insights into the state-of-the-art technological stacks such as Angular, Firebase, RxJS, etc. were established. This work can serve as a modern guide for the development of Single Page Applications regardless of the development platform or size of the application. A thorough comparison, supported with examples of the benefits and drawbacks of the centralized state management pattern, was provided and recommendations were presented.

## 7.1 Future work

One of the main benefits of developing a Single Page Application using Angular and Firebase is the platform agnostic features of the frameworks. Using the current code base, the application could be wrapped with little effort into a basic Cordova [21] application that

---

would allow the distribution and native functionalities of the Chat and Ticket management system onto Android, IOS or Web platforms. Using this approach, we can minimize the time for development of an application that would target a variety of operating systems. Furthermore, because of the Redux implementation, the application could be enhanced to a Progressive Web Application [22] with offline capabilities in an easier and more structured manner. Finally, in order to achieve a true team separation capability, an additional authentication step specifying a team name could be introduced that would add one extra node, creating team database nodes inside Firebase.

## References

- [1] Y. T. W. Tiky, Software Development Life Cycle, The Hong Kong University of Science and Technology, 2016.
- [2] A. Hussain, Angular From Theory to Practice, vol. 1.2.0, 2017-11-24..
- [3] M. Wasson, "ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET," [Online]. Available: <https://msdn.microsoft.com/en-us/magazine/dn463786.aspx>. [Accessed 1 2 2019].
- [4] J. C. Michael S. Mikowski, Single Page Web Applications, Manning Publications, 2014.
- [5] "Angular Architecture overview," Google Inc., [Online]. Available: <https://angular.io/guide/architecture>. [Accessed 2 February 2019].
- [6] D. Shapiro, "Understanding Component-Based Architecture," A Medium Corporation, 2016. [Online]. Available: <https://medium.com/@dan.shapiro1210/understanding-component-based-architecture-3ff48ec0c238>. [Accessed 16 3 2019].
- [7] K. Lane, Overview Of The Backend as a Service (BaaS) Space, 2013.
- [8] P. S. E. F. O. Dr. Alshafie Gafaar Mhmoud Mohmmed, Journal of Multidisciplinary Engineering Science Studies (JMESS), 2017.
- [9] B. N. a. B. Lucie, Real-Time database: Firebase INFO-H-415 : Advanced database, 2017-2018.
- [10] "Angular Material," Google Inc, [Online]. Available: <https://material.angular.io/components/sidenav/overview>. [Accessed 27 2 2019].
- [11] A. Shvets, Dive Into Design Patterns, vol. 1.1, 2019.
- [12] R. H. R. J. J. V. Erich Gamma, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
- [13] K. T. Martin Kaluza, Comperison of Front-End Frameworks for Web Applications development, vol. 6, Zbornik Veleučilišta u Rijeci, 2018.



- 
- [14] K. Sabadir, "React Redux," *SiteOS*, 2017.
- [15] S. Mansilla, *Reactive Programming with RxJS*, The Pragmatic Programers, LLC, 2015.
- [16] E. Aybar, "Optimistic UI Updates in React," 2018. [Online]. Available: [https://medium.com/@\\_erikaybar/optimistic-ui-updates-in-react-9e139ffa2e45](https://medium.com/@_erikaybar/optimistic-ui-updates-in-react-9e139ffa2e45). [Accessed 5 February 2019].
- [17] A. Morgan, "How to test Angular CanActivate Guards," 2018.
- [18] A. V. Pranay Dutta, "Karma- The Test Runner, for Automated Testing of Web Based Applications".
- [19] P. Krastev, *Design and implementation of a microservice for deletion of resources in the Multi-Agent Research and Simulation distributed system*, Hamburg: HAW Hamburg, 2018.
- [20] A. Figueroa, *Development of a graphical user interface for X-ray*, HAW Hamburg, Hamburg, 2017.
- [21] A. Grieve, *The Cordova Development Lifecycle*, 2014.
- [22] D. A. Hume, *Progressive Web Apps*, Manning Publications, 2018.
- [23] D. S.Cohen, C. &. Control, Ed., *Int. J. of Computers*, 2010.

# Appendix A

The following code snippet has been taken from *ChatroomService* class, which can be found at the following path: `...src/app/services/chatroom.service.ts`

```
export class ChatroomService {

  public changeChatroom: BehaviorSubject<string | null> = new
  BehaviorSubject(
    null
  );
  public selectedChatroom: Observable<any>;

  this.selectedChatroom = this.changeChatroom.pipe(
    switchMap(chatroomId => {
      console.log(chatroomId);
      if (chatroomId) {
        this.loadingService.isLoading.next(true);
        db
        .doc(`chatrooms/${this.authService.currentUserSnapshot.id}/chatrooms
        /${chatroomId}`)
        .update({unread: false})
        .catch(err => console.log('Error in updating the unread property of
        the chatroom', err));

        return db
        .doc(`chatrooms/${this.authService.currentUserSnapshot.id}/cha
        trooms/${chatroomId}`)
        .valueChanges();
      }
      return of(null);
    }));
}
```

The following code snippet has been taken from ChatroomWindowComponent class, which can be found at the following path: ...src/app/pages/chat/components/chatroom-window/chatroom-window.component.ts

```
export class ChatroomWindowComponent implements OnInit, OnDestroy,
AfterViewChecked, OnChanges {
  this.subscriptions.push(
    this.chatroomService.selectedChatroom.subscribe(chatroom => {
      this.chatroom = chatroom;
      this.loadingService.isLoading.next(false);
    });
  ngOnChanges(changes: SimpleChanges) {

    this.chatroomService.changeChatroom.next(changes.chatroomID.currentV
alue);
  }
}
```

## Appendix B

The following code snippet has been taken from the task reducer file, which can be found at the following path: ...src/app/store/reducers/task-reducer.ts

```
import * as taskActions from '../actions/task-actions';
import { TaskModel } from 'src/app/shared/models/tasks.model';
import { Action } from 'rxjs/internal/scheduler/Action';

export interface TaskState {
  tasks: TaskModel[];
  backlog: TaskModel[];
}
// The state with which the reducer will be innitialized
export const innitialState: TaskState = {
  tasks: [],
  backlog: []
};

// State controller
export function reducer(
  state: TaskState = innitialState,
  action: taskActions.TaskActions
): TaskState {
  console.log(action);
  switch (action.type) {
    case taskActions.CREATE_TASK_FAIL:
    case taskActions.MOVE_TO_BACKLOG_FAIL:
    case taskActions.REMOVE_TASK_FAIL:
    case taskActions.REMOVE_TASK_FROM_BACKLOG_FAIL:
    case taskActions.GET_BACKLOG_TASKS_FAIL:
    case taskActions.GET_TASKS_FAIL: {
      console.log(action.payload);
      return { ...state };
    }
  }
}
```

```
}

case taskActions.CREATE_TASK:
case taskActions.GET_BACKLOG_TASKS:
case taskActions.MOVE_TO_BACKLOG:
case taskActions.REMOVE_TASK:
case taskActions.REMOVE_TASK_FROM_BACKLOG:
case taskActions.GET_TASKS: {
  console.log(action.type);
  return { ...state };
}

case taskActions.GET_TASKS_SUCCESS: {
  console.log('Reducer Tasks action success');
  const newState = { ...state, tasks: action.payload };
  console.log(newState);
  return newState;
}

case taskActions.CREATE_TASK_SUCCESS: {
  console.log(action.type);
  const newTaskArr = state.tasks;
  newTaskArr.push(action.payload);

  return { ...state, tasks: newTaskArr };
}

case taskActions.GET_BACKLOG_TASKS_SUCCESS: {
  console.log('Reducer BacklogTasks action success');
  const newState = { ...state, backlog: action.payload };

  return newState;
}

case taskActions.MOVE_TO_BACKLOG_SUCCESS: {
  const newState = { ...state };
  newState.tasks.forEach(task => {
    if (task.id === action.payload.id) {
      newState.backlog.push(task);
    }
  });
  newState.tasks = newState.tasks.filter(
    task => task.id !== action.payload.id
  );
  console.log(newState);
  return newState;
}

case taskActions.MOVE_TASK_TO_SPRINT: {
  const newState = { ...state };
}
```

```
    const backlogs = state.backlog;
    newState.backlog.forEach(task => {
      if (task.id === action.payload.id) {
        newState.tasks.push(task);
      }
    });
    newState.backlog = newState.backlog.filter(
      task => task.id !== action.payload.id
    );
    console.log(newState);
    return newState;
  }

  default: {
    return state;
  }
}

export const getTasks = (state: TaskState) => state.tasks;
export const getBacklog = (state: TaskState) => state.backlog;
```

## Appendix C

The following code snippet has been taken from file, which can be found at the following path: ...src/app/pages/chat/components/chatroom-list/chatroom-list.component.spec.ts

```
import { async, ComponentFixture, TestBed } from
 '@angular/core/testing';

import { ChatroomListComponent } from './chatroom-list.component';
import { ChatroomService } from 'src/app/services/chatroom.service';
import { of } from 'rxjs';
import { By } from '@angular/platform-browser';

describe('ChatroomListComponent', () => {
  let component: ChatroomListComponent;
  let fixture: ComponentFixture<ChatroomListComponent>;
  let chatroomService: ChatroomService;
  let spy: any;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ ChatroomListComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    chatroomService = new ChatroomService(null, null, null);
    fixture = TestBed.createComponent(ChatroomListComponent);
    component = fixture.componentInstance;
    component.chatroomService = chatroomService;
    fixture.detectChanges();
  });
```

```
it('should create', () => {
  expect(component).toBeTruthy();
});
it('should hide badge icon', () => {
  spy = spyOn(chatroomService, 'statArr').and.returnValue(
    of([{unread: false}])
  );

  expect(fixture.debugElement.query(By.css('.badge')).nativeElement).t
oBeUndefined();
});
});
```



# Declaration

*I declare within the meaning of section 25(4) of the Examination and Study Regulations of the International Degree Course Information Engineering that: this Bachelor report has been completed by myself independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.*

Hamburg, 21.03.2019

\_\_\_\_\_  
Signature: Ivan Mihaylov