Danh Nguyen Cong

PRACTICAL TRAINING

LOGGING SYSTEM

Technology and Communication
2020

ACKNOWLEDGEMENT

First and foremost, I would like to express my unending gratitude to my parents, whom despite all my flaws and shortcomings, have kept supporting me throughout the year.

Secondly, I would like to thank my teachers at VAMK. It has only been for their advice and training that allow me to progress in my studies.

Thirdly, I wish to thank Mr. Maximilian Schwarzmüller and Mr. Neil Cummings from whom I learn to use Angular and EF Core.

Finally, I would like to thank community on platforms such as Stack Overflow, GitHub, Gitter, Discord where I have spent countless hours on in order to complete this project.

Sincerely, thank you.

Vaasa, November 10, 2019

Nguyen Cong Danh

VAASAN AMMATTIKORKEAKOULU
Information Technology

ABSTRACT

Author	Nguyen Cong Danh
Title	Practical Training Logging System
Year	2019
Language	English
Pages	70
Name of Supervisor	Ghodrat Moghadampour

The objective of this thesis was to develop a program for students to log their practical training experience and working hours. This information can be used to calculate obtainable credit units during their training, while also help in writing practical training reports.

The application includes two layers. A Client where the user interacts with the possible operations, such as registering contract detail, adding new work diary, tallying working hours. This client was built for Android and Windows computers, using common web technologies at its core, such as HTML, TypeScript, CSS along with modern developing frameworks, such as Ionic Framework and Angular.

Reinforcing the Client is the API layer where logic and methods for communicating with the database of the application is found. The API layer allows the Client applications to read entries from the databases, adding new items, such as a new work diary and work hours and modifying existing data, such as the user information. Built using C#, with support from Entity Framework, this layer was programed following the repository pattern, which allows the application to be detached from Entity Framework in the future as needed.

The project created an application that allows students to log information during their practical training period and calculating the credit units a student can get for their working hours. The program was tested with Android as a native application and Window computer with Google Chrome.

Keywords Ionic Framework, Angular and Entity Framework

TABLE OF CONTENTS

1. INTRODUCTION 1

1.1. OBJECTIVES 2

2. TECHNOLOGIES 3

2.1. ANGULAR 3

2.2. IONIC 3

2.3. NODE PACKAGE MANAGER 4

2.4. QUILLJS 5

2.5. NGX-QUILL 5

2.6. NGX-PIPES 5

2.7. DATE-FNS 6

2.8. ENTITY FRAMEWORK 6

2.9. JSON WEB TOKEN 7

2.10. AUTOMAPPER 8

3. APPLICATION DESCRIPTION 9

3.1. GENERAL DESCRIPTION 9

3.2. QUALITY FUNCTION DEPLOYMENT 9

3.3. USE CASE DIAGRAM 11

3.4. REPOSITORY PATTERN AND CLASS DIAGRAM FOR SERVER-SIDE API 12

3.4.1. Repository Pattern 13

3.4.2. Class Diagram 13

3.5. SEQUENCE DIAGRAM 17

3.5.1. Register Sequence Diagram 17

3.5.2. Login Sequence Diagram 18

3.5.3. Editing Contract Information Sequence Diagram 19

3.5.4. Editing Work Diary Sequence Diagram 20

3.5.5. Adding New Work Session Sequence Diagram 21

3.5.6. Removing Work Session Sequence Diagram 22

4. DATABASE AND GUI DESIGN 24

4.1. DATABASE 24

4.2. GUI DESIGN 25

5. IMPLEMENTATION 35

5.1. GENERAL STRUCTURE OF THE PROJECT 35

5.1.1. Ionic Application 35

5.1.2. Server-side EF Core API 36

5.2. IMPLEMENTATION OF THE IONIC CLIENT 37

5.2.1. Authentication 38

5.2.2. Navigation Menu 40

5.2.3. User Action Notification 40

5.2.4. ContractInfo Page 41

5.2.5. ContractInfoEdit Page 42

5.2.6. WorkDiary Page 43

5.2.7. WorkDiaryEdit Page 45

5.2.8. TimeSheet Page 48

5.2.9. TimeSheetDetail Component 49

5.2.10. Preventing Accidental Page Navigation with CanDeactivate 51

5.2.11. Setting up API endpoint address 53

5.3. IMPLEMENTATION OF THE SERVER-SIDE API 53

5.3.1. Models 53

5.3.2. Automapper 54

5.3.3. Repository and DataContext 55

5.3.4. Controllers 57

6. TESTING 59

6.1. REGISTERING AS A NEW USER 59

6.2. EDITING CONTRACT INFORMATION 60

6.3. ADDING NEW WORK DIARY 61

6.4. EDITING AND REMOVING EXISTING WORK DIARY CARD 63

6.5. ADDING NEW WORK SESSION: 65

6.6. DELETING WORK SESSIONS. 67

7. CONCLUSIONS 69

REFERENCES 70

LIST OF FIGURES AND TABLES

Figure 1. Working with Ionic. /4/ 4
Figure 2. Sample of a decoded JWT. /11/ 7
Figure 3. Use case diagram. 12
Figure 4. Repository Pattern. 13
Figure 5. Diagram for DataContext 14
Figure 6. AuthRepository 14
Figure 7. UserActionRepository 15
Figure 8. AuthController 15
Figure 9. TimeMarkController 16
Figure 10. UsersController 16
Figure 11. WorkDiaryController 17
Figure 12. User register Sequence Diagram 18
Figure 13. Login Sequence Diagram**.** 19
Figure 14. Editing contract information Sequence Diagram 20
Figure 15. Adding, removing and editing work diary Sequence Diagram 21
Figure 16. Adding new work session Sequence Diagram 22
Figure 17. Removing work session Sequence Diagram. 23
Figure 18. ER diagram 24
Figure 19. Login form 25
Figure 20. Sign up form 26
Figure 21. ContractInfo desktop page 26
Figure 22. ContractInfoEdit desktop page 27
Figure 23. ContractInfo android page 28
Figure 24. ContractInfoEdit Android page 29
Figure 25. Navigation menu on Android 30
Figure 26. WorkDiary page 31
Figure 27. WorkDiaryEdit desktop page 31
Figure 28. WorkDiary Android page 32
Figure 29. WorkDiaryEdit Android page 33

Figure 30. TimeSheet desktop page 33
Figure 31. Timesheet Android page 34
Figure 32. Basic Ionic Application Structure 35
Figure 33. Client structure 36
Figure 34. Server-side API structure 37
Figure 35. Successful registration 59
Figure 36. Error during registration with message 60
Figure 37. Successfully redirected with contract information updated 61
Figure 38. Filling in the form with predefined template 62
Figure 39. New work diary card added 62
Figure 40. WorkDiaryEdit loaded with previously input data 63
Figure 41. Removal confirmation 64
Figure 42. Modified work diary card 65
Figure 43. Selecting the date 66
Figure 44. New work session with total hours and credit unit calculated 67
Figure 45. Removed work session with all information re-calculated 68

LIST OF SNIPPETS

Code Snippet 1. Simple property binding and interpolation with Angular.....3

Code Snippet 2. Adding checkbox components, with property binding using Ionic.....4

Code Snippet 3. Reverse a string using pipe.....6

Code Snippet 4. Quickly comparing different time 6

Code Snippet 5. Adding new entity to database using Entity Framework.....6

Code Snippet 6. Map user.Info to a similar InfoDto object with selected properties.....8

Code Snippet 7. AuthGuard placed on “contractinfo” page. 38

Code Snippet 8. AuthService validating login status.....38

Code Snippet 9. Implementation of AuthService and its methods. 39

Code Snippet 10. LoggedInGuard placed on Auth page.....39

Code Snippet 11. Implementation of log out as a service.....39

Code Snippet 12. Implementation of navigation menu.....40

Code Snippet 13. Generate a toast 41

Code Snippet 14. Resolver return an observable of type ContractInfo.....41

Code Snippet 15. Implementation of the UserService and its function. 42

Code Snippet 16. Reactive Form Input Field with validation.....42

Code Snippet 17. Validate and submit new information. 43

Code Snippet 18. List of WorkDiary. 44

Code Snippet 19. Methods to manage the list of WorkDiary. 45

Code Snippet 20. ngx-quill configuration. 46

Code Snippet 21. Adding new work diary. 47

Code Snippet 22. Editing existing work diary. 48

Code Snippet 23. Adding new session. 49

Code Snippet 24. Methods implemented in TimeMarkService. 49

Code Snippet 25. Using and binding child component. 49

Code Snippet 26. Formatting data input. 50

Code Snippet 27. Sending total hour back to the main page. 50

Code Snippet 28. Removing work session. 51

Code Snippet 29. PreventUnsavedChanges guard and its alert function. 52

Code Snippet 30. Applying canDeactivate guard to the page. 52

Code Snippet 31. Environment setting. 53

Code Snippet 32. User model 53

Code Snippet 33. WorkDiary model. 54

Code Snippet 34. Automapper profile 55

Code Snippet 35. Verify user. 55

Code Snippet 36. Registering new user and create password hash. 56

Code Snippet 37. Retrieve user work session. 56

Code Snippet 38. AuthController implementation of login. 57

Code Snippet 39. Implementation of add and update diary. 58

LIST OF ABBREVIATIONS

API	Application Programing Interface
SPA	Single Page Application
CLI	Command Line Interface
Dto	Data Transfer Object
GUI	Graphical User Interface
SDK	Software Development Kit
JSON	JavaScript Object Notation
IDE	Integrated Development Environment
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
SASS	Syntactically Awesome Style Sheets
URL	Uniform Resource Locator
NPM	Node Package Manager
JWT	Json Web Token
HMAC	Hash-based Message Authentication Code
RSA	Rivest-Shamir-Adleman
ECDSA	Elliptic Curve Digital Signature Algorithm

1. INTRODUCTION

1.1. Background and motivation

Many students are doing practical training as part of their degree programmes every year. At the end of the training, they are expected to turn in a report detailing the workflow and experience earned in the working environment. With how busy the situation can be, not everyone is able to keep their thought in good collection. This fuels the need to have a platform where notes can be kept in an organized way for future references.

The program is needed to help students log in their work experience during their practical training period. As many students only work on their reports near the end of their training, there can be a problem where students do not remember clearly what their job was during the early periods. In form of work diary cards, the application allows students to write in the content of their tasks, includes pictures for the reports.

The program also provides questions that teachers want to get in a practical training report, which can guide students on what information to include in the report. The program also helps calculating the number of credit units the student is qualified for at the end of the training period.

Due to the widespread use of mobile devices, the application can be made to be both a web and mobile application. This will allow students easy access to their data to make quick and simple editions. This is especially useful for when the student only needs to add, remove or modify work hours. The mobile application also helps in cases where desktops are not immediately available, such as when working outdoor in the field.

1.2. Objectives

The project should have user friendly interfaces for users to manage working notes and timesheet, allowing rich documents to be submitted and saved. The client should be responsive, easy to navigate, secured with authentications, and being able to work as a native application on Android.

The backend API should have reliable security mechanism in place to prevent un-authorized access request.

2. TECHNOLOGIES

2.1. Angular

Angular, also known as “Angular 2+” is a TypeScript-based open-source web application framework led by the Angular Team at Google, community of individuals and corporation /1/.

Comparing to other popular solutions, such as React, Angular offers many advanced out-of-the-box features, such as two-way databinding, services and dependency injection, routing, and http request. Bring both advantages and disadvantages to the development process. An example of how Angular uses two-way databinding and interpolation is demonstrated in the code snippet below /2/.

```
<!-- Bind button disabled state to `changed` property -->
<!—Label for the button is “Button label 2”-->
<button [disabled]="changed">Button label {{ 1+1 }}</button>
```

Code Snippet 1. Simple property binding and interpolation with Angular**.**

Angular is still being actively maintained and improved. While major versions of the framework are published approximately biannualy, most of the underlying code remain similar to older version, making changes easily upgradable.

For this project, Angular 8 was used, with support of third-party components.

2.2. Ionic

Ionic is an open-source SDK for hybrid mobile application development. Originally released in 2013 for AngularJS and Apache Cordova, the framework was re-built and released as a set of web component, which allows the user to choose any user interface framework, such as Angular, Vue.js or React. It can also be used as a standalone package.

Ionic allows developing hybrid mobile, desktop, and progressive web applications based on modern familiar web technologies, such as CSS, HTML, and SCSS.

```
<ion-checkbox disabled="true"></ion-checkbox>
<ion-checkbox color="primary"></ion-checkbox>
<ion-checkbox color="secondary"></ion-checkbox>
<ion-checkbox color="danger"></ion-checkbox>
```

Code Snippet 2. Adding checkbox components, with property binding using Ionic.

Mobile applications can be built and distributed as a normal native application with these technologies by utilizing Cordova or Capacitor /3/.

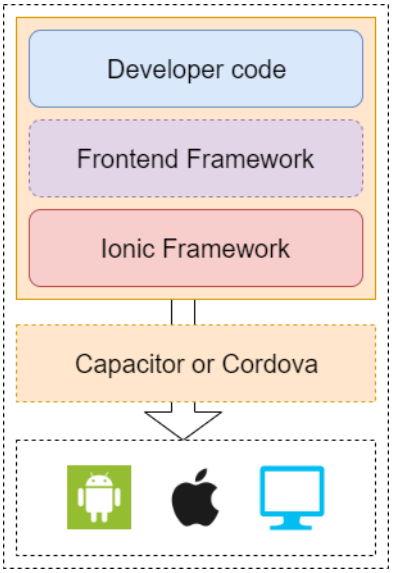

Figure 1. Working with Ionic. /4/

Ionic 4 is the latest major version of Ionic, and is the version used for this project.

2.3. Node Package Manager

Node package manager (NPM) is a package manager for JavaScript programming language. Consisting of a command line client and an online database of packages, it allow dependencies to be installed and managed straightforwardly, removing the

overhead of handling external libraries manually, thus minimizing human error during development /5/.

2.4. Quilljs

Quill is an open source editor built for modern web. Quill is modular and customizable where many of its features can be removed if they are not required by the project. Documents created from Quill are in the Delta format, which can also be parsed into formats such as plain text, JSON or HTML.

Quilljs includes many common functionalities of a modern rich text editor, such as adding bold, italic, underscore, setting text size and fonts, adding list items, embedding URL to strings, adding images and videos. /6/.

2.5. ngx-quill

ngx-quill is a third-party open source Angular module for Quilljs. It is currently frequently updated and maintained by user killercodemonkey. ngx-quill allow developers to work with Quilljs as an Angular module, rather than a regular JavaScript extension. This includes calling for Quilljs as a component in the html template with the <ngx-quill> tag and binding it with attributes, such as any other Angular components./7/.

2.6. ngx-pipes

ngx-pipes is a third-party open source library consisting of custom pipes for Angular that are commonly used. These pipes can also be used as injectable and used in components or services. These pipes allow the quick modification of variable displayed on the Html. Some of the functions of ngx-pipes include trimming strings, reversing, filtering and sorting arrays, doing minor mathematical operation, such as taking average and square roots, comparing strings, numbers and objects and returning Booleans on conditions.

```
<p>{{'foo bar' | reverse }}</p>
<!-- Output: "rab oof" -->
```

Code Snippet 3. Reverse a string using pipe.

2.7. date-fns

date-fns is a light weight, simple, yet comprehensive JavaScript library consisting of toolset for manipulating JavaScript dates. It allows various changes to date time objects, such as adding, subtracting, getting a specific date, getting week numbers, and comparing different time marks. /8/.

```
// Compare 11 February 1987 and 10 July 1989:
var result = compareAsc(new Date(1987, 1, 11), new Date(1989, 6, 10))
//=> -1
```

Code Snippet 4. Quickly comparing different time.

2.8. Entity Framework

Entity Framework, or more commonly known as EF Core, is a set of technologies in ADO.NET that support the development of data-oriented software applications. It enables developers to work with data in form of objects and properties, without having to concern with the underlying database.

Developers can work at a higher level of abstraction when dealing with data and be able to create and maintain data-oriented applications with less code than traditional applications /9/.

```
public class UserActionRepo: IUserActionRepo {
        private readonly DataContext context;
        public void Add<T>(T entity) where T : class{
                this.context.Add(entity);
        }
};
```

Code Snippet 5. Adding new entity to database using Entity Framework.

While the latest version is EF Core 3, EF Core 2 is used to develop this application.

2.9. Json Web Token

JSON Web Token is an open standard that allows data to be securely transmitted between parties as a JSON object. This data can be used to send authentication detail, and possible small, confidential, yet critical information. JWTs can be signed using a secret (with the HMAC algorithm) or a public and private key pair using RSA or ECDSA /10/.

In this application, we use a secret key encrypted with HMAC algorithm for authentication purpose.

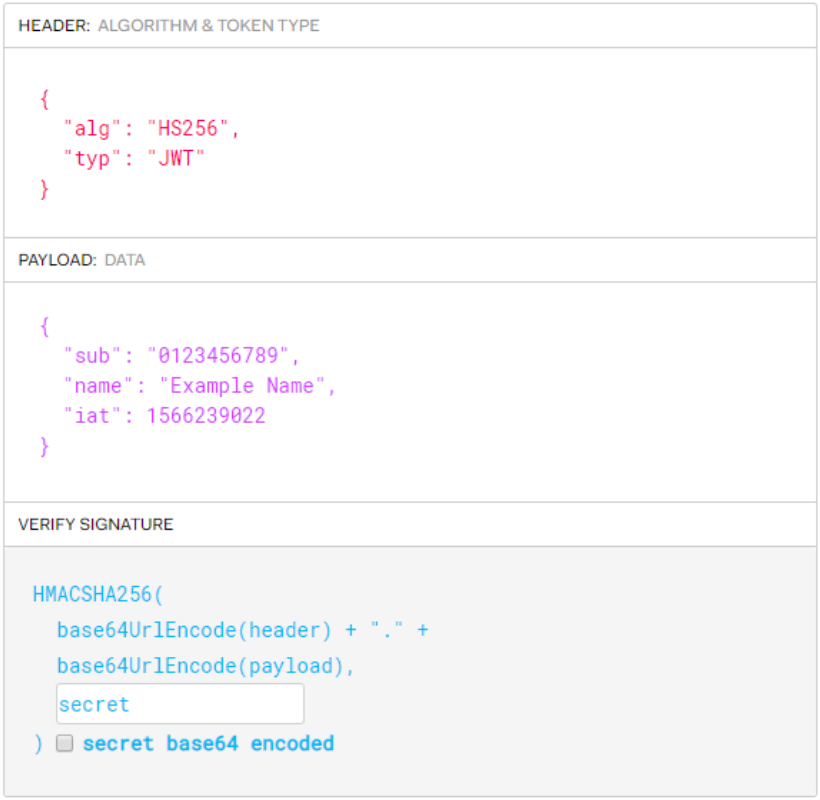

Figure 2. Sample of a decoded JWT. /11/

A token can also carry additional information in its payload, but one must exercise caution doing so as the data is not secured.

2.10. AutoMapper

AutoMapper is a third-party library built to help automate mapping one object to another during development. As these types of code are long and unnecessarily complex to write, this library will bootstrap the process and minimize human errors.

```
var infoToReturn = this.mapper.Map<InfoDto>(user.Info);
```

Code Snippet 6. Map user.Info to a similar InfoDto object with selected properties

3. APPLICATION DESCRIPTION

3.1. General Description

The main requirement of the application is to be able to register work experience of a user during their practical training periods. The application should then communicate with a remote database to store such information. A native Android application should also be included along with a web application. The client and API of the application must include the following features:

- Client:

  Display information of the practical training contract, working log, worked time. The user is able to save rich documents with appropriate text, image media files, URLs timesheet in appropriate formatting.

- Server-side API

  Allow communication from the client side with operations, such as creating a new user, managing contract information, working logs, add and remove user worked time. The API must prevent unauthorized access to all user data.

3.2. Quality Function Deployment

Base on the priorities, the requirements of the application can be divided into three different parts: must-have, should-have and good-to-have. Must-have features are the main requirements that the application should be able to accomplish to be considered a success. Should-have and good-to-have features are features that are additional for better user experience and future development of the application.

3.2.1. Must-have requirements

The must-have requirement of the application are listed below:

- Client application:
    - Registering as a new user.
    - Signing in and out of the application.
    - Displaying the information of the contract, the user working diary, the user worked sessions.
    - Editing contract information.
    - Adding and removing worked sessions and working logs.
- Server-side API:
    - Allowing signing in and out of the client application.
    - Distributing tokens for future authenticating attempts.
    - Preventing unauthorized user to access data from the API.
    - Editing contract info.
    - Add and remove work time and diary upon client request.

3.2.2. Should-have requirements

The should-have requirements of the application are listed below:

- Client application:
    - Editing past working diary with more detailed information.

- Adding loading screen during tasks requiring server communication.
- Displaying messages indicating the result of the client when communicating with the API for better user experience.

- Server-side API:
  - Allow editing past working diary with new information.
  - Storing account password in a secured and encrypted way as a fail-safe mechanism in an event of a database breach.

3.2.3. Nice-to-have requirements

The nice-to-have requirements of the application are listed below

- Client application
  - Responsive UI
  - Calculate total working hours.
  - Ask for use confirmation when leaving the page if the application detects there is unsaved input to prevent accidental loss of data.
  - Calculate the number of credit units the user can get from the total working hours

3.3. Use Case Diagram

The use case diagram for the client application is presented below.

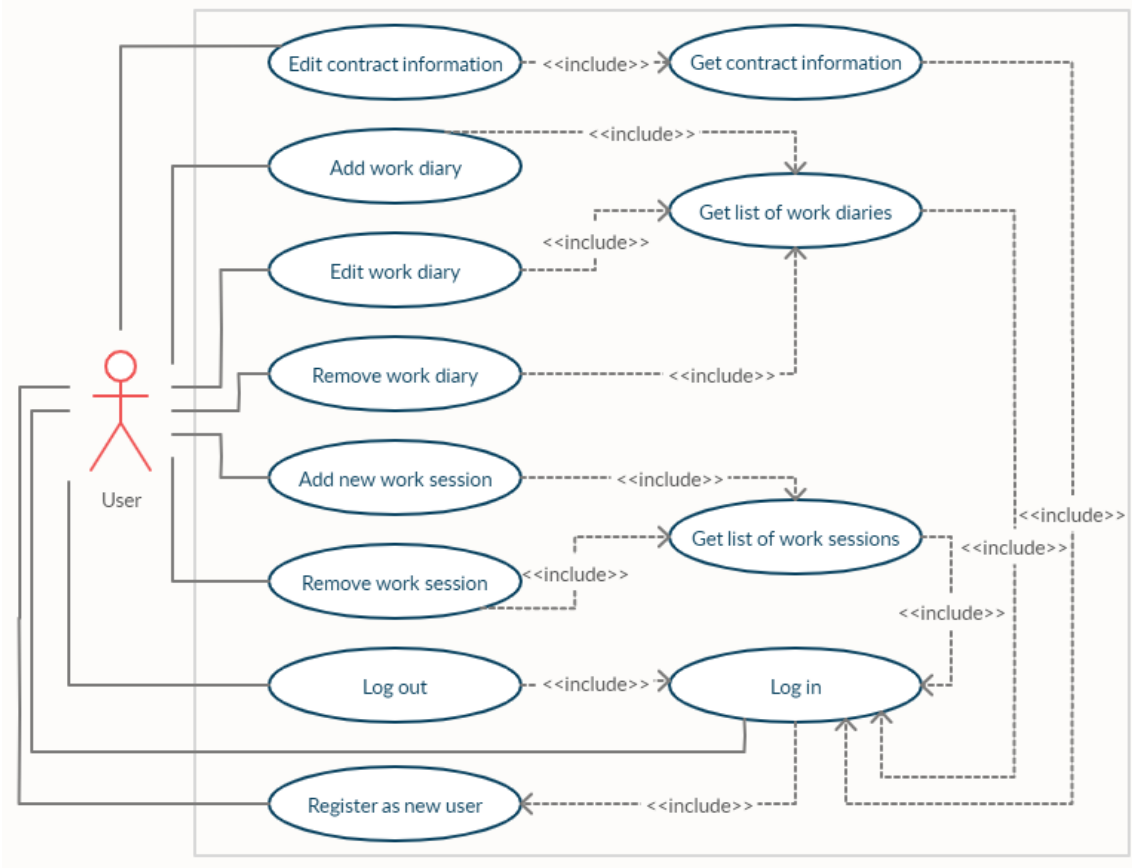

Figure 3. Use case diagram.

From the use case diagram, the student can get their contract information, work diaries and work sessions. These actions also include the option to remove, edit or add new information. All operation requires the user to be authenticated by logging in and registering an account with the application prior.

3.4. Repository Pattern and Class Diagram for Server-side API

The repository pattern and class diagram for the server-side API are introduced in this chapter.

3.4.1. Repository Pattern

This application follows the repository pattern.

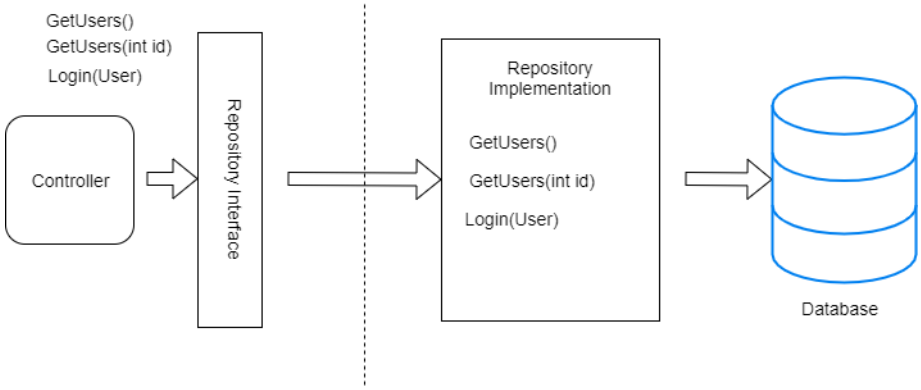

Figure 4. Repository Pattern.

Benefits of repository pattern includes:

- The repository pattern minimizes repeating query logic
- Allows the application to be decoupled from persistence framework
- All database queries are centralized for the ease of code management

The classes that expose the possible operations to the client application from the server side are called the controllers. These controllers interact with the interfaces defined in the repository interface, which then continue to use the framework-specific implementations to access and modify the database. This allows any future development, if required, to separate the rest of the application from Entity Framework.

3.4.2. Class Diagram

The following diagram demonstrates the functions included within DataContext.cs file.

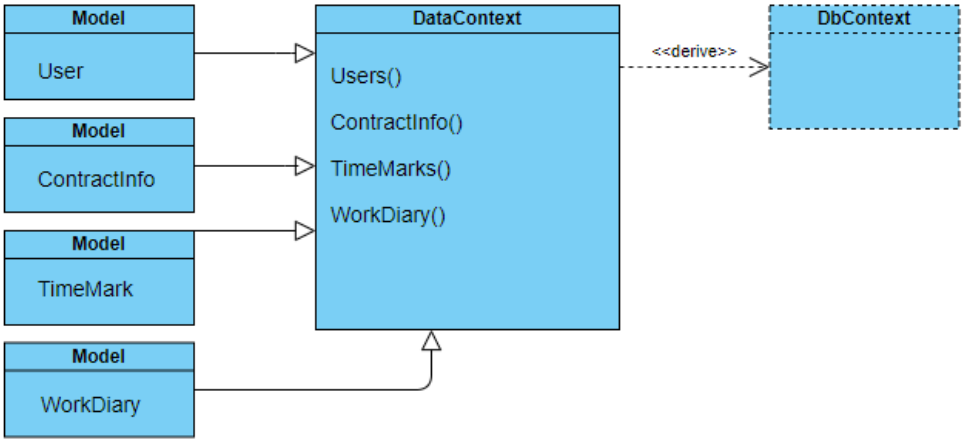

Figure 5. Diagram for DataContext

- DataContext is responsible for scaffolding the database with predefined models. Inheriting DbContext class that comes with Entity Framework Core allows it to query and save instances of entities to the database.

For tasks related to authenticating the users, the solutions are implemented in AuthRepository.cs file:

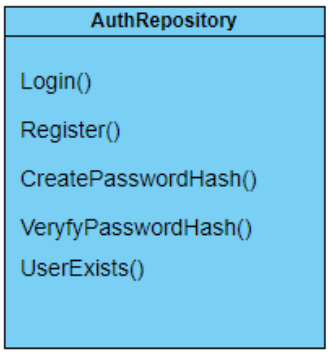

Figure 6. AuthRepository

- This repository is responsible for logging and creating new users. It also handles checking creating hashes that are used to securely store the password in the database.

For user related activities, methods are defined in UserActionRepository.cs

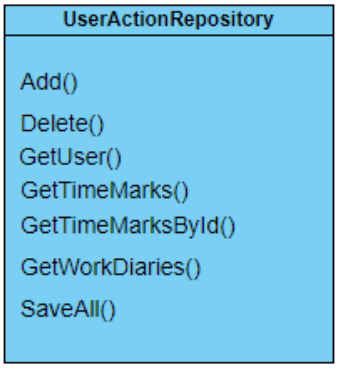

Figure 7. UserActionRepository

- Generic methods, such as Add(), Delete(), SaveAll() are responsible for updating entities within the database
- Dedicated methods, such as GetUser(), GetTimeMarks() or GetTimeMarkById() are used by the API controllers

Controller files are responsible for connecting the API URLs into the correct operation. The functions of these controller are described in the diagrams below.

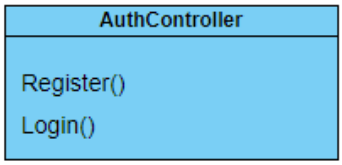

Figure 8. AuthController

AuthController allows users to register and login into the application. The login method will also set a JWT token which allows the user to access the application without additional login until they are logged out

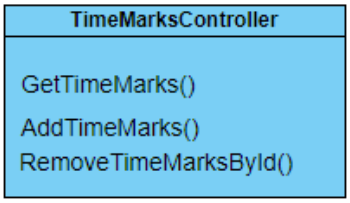

Figure 9. TimeMarkController

TimeMarksController allow users to add new, remove and modified existing time-sheet entries. If any errors occur when handling the request, the application will return an error message indicating what failure has occurred during the process.

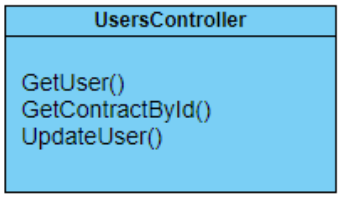

Figure 10. UsersController

UsersController returns user information with GetUser() and GetContractById() method. Any requests to modify such information will also be handled by this controller

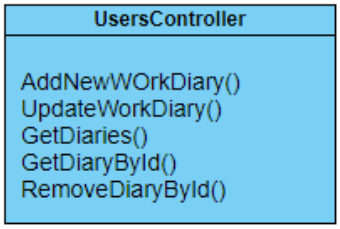

Figure 11. WorkDiaryController

WorkDiaryController allows access to existing work diary with the GetDiaries() and GetDiaryById() method. The need to add, update, or remove the work diaries will also be handled here.

In the Helpers folder, there are two classes named Extensions and AutoMapperProfile. These classes offer no standalone method and are used as configuration for error handling AutoMapper respectively.

3.5. Sequence Diagram

In this section we discuss the flow of the application, by using sequence diagrams:

3.5.1. Register Sequence Diagram

The following diagram describe how the registration is handled:

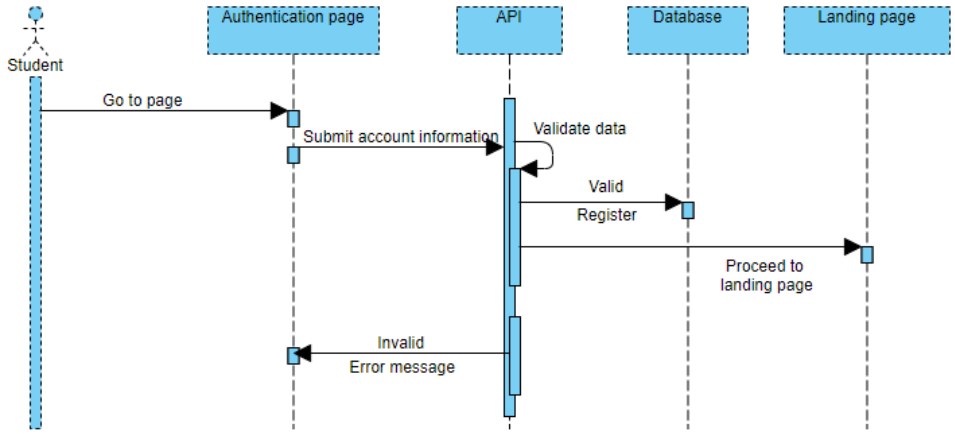

Figure 12. User register Sequence Diagram

Firstly, the user enters a new account information. The application will send a request to create a new user upon user submitting the form. The server-side API will then start to validate the data input and will either register the new account to the database or give back an error message to the client. The application will then automatically log in and redirect to the landing page.

3.5.2. Login Sequence Diagram

The sequence diagram in Figure 13 demonstrates the procedure involved in login of users.

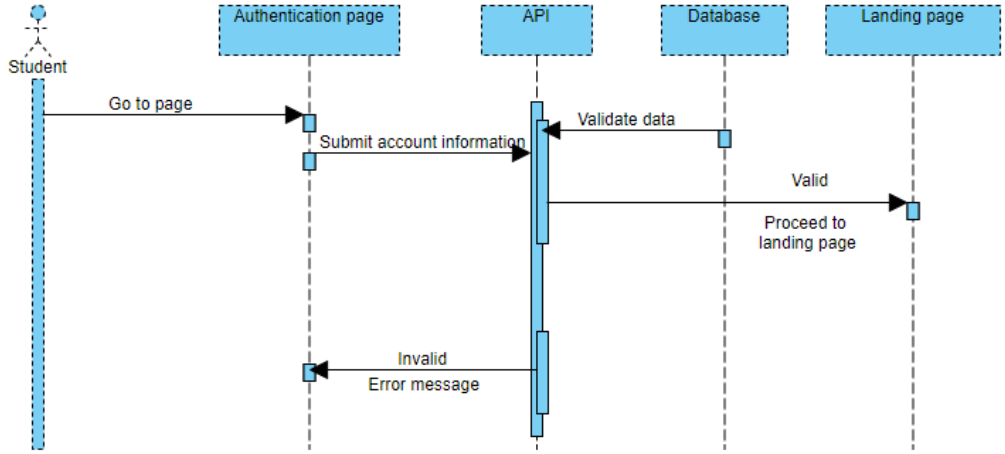

Figure 13. Login Sequence Diagram**.**

In order to login into the application, the user needs to submit a form with the application. The client will send a request to the server-side API asking for validation for the input. If the account information is valid, it proceeds to redirect the user to the landing page or shows an error message informing the user of invalid input.

3.5.3. Editing Contract Information Sequence Diagram

In order to manage contract information, the user needs to navigate to the contract-info page, which also serves as the landing page of the application.

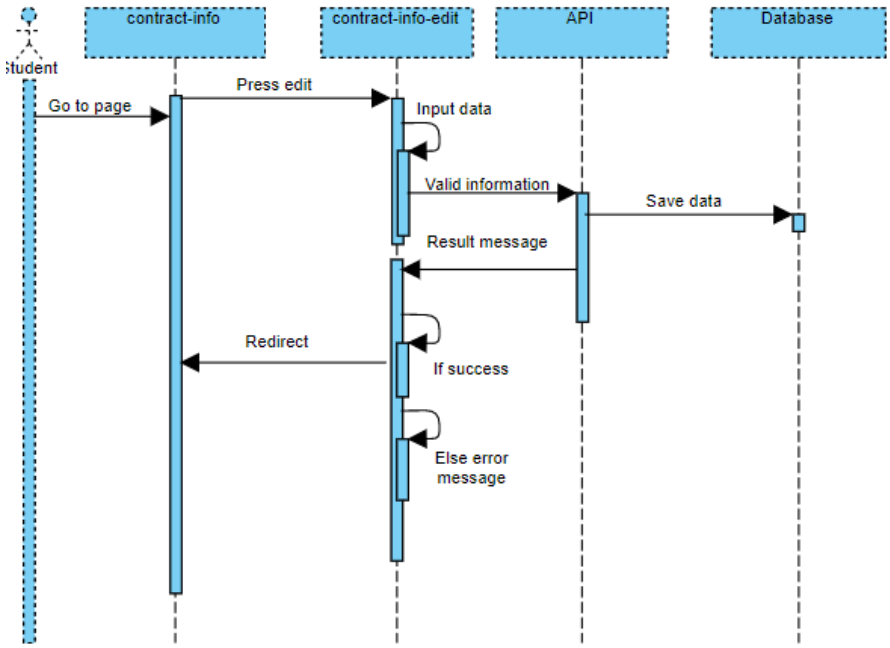

Figure 14. Editing contract information Sequence Diagram

Pressing the Edit button will further redirect the user to another page where data is presented as an easily editable form. Upon submit, the form will send a request to the API. Validations are in place to ensure user input appropriate data. The application will then redirect the user back to contract-info upon successful save or show an error message to user upon failure.

3.5.4. Editing Work Diary Sequence Diagram

To manage new work diaries, the user navigates to the work-diary page and presses the appropriate button indicated for each operation. This will redirect the user to the work-diary-edit page where the user can work on the work diary.

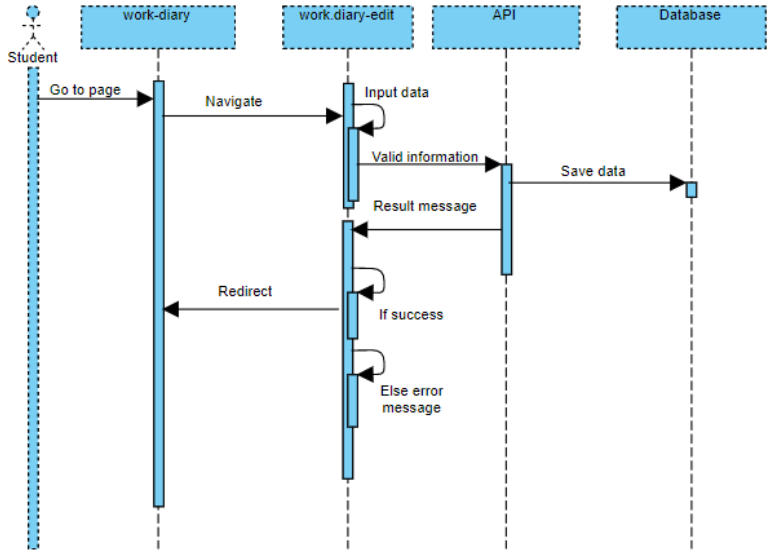

Figure 15. Adding, removing and editing work diary Sequence Diagram

Upon submission, the client will request the server-side API to add, remove or edit an entry of work diary. The user will be redirect to the work-diary upon successful save or shown an error message upon failure.

3.5.5. Adding New Work Session Sequence Diagram

Figure 16 below illustrates the step the user can take to add or remove work diary to their account.

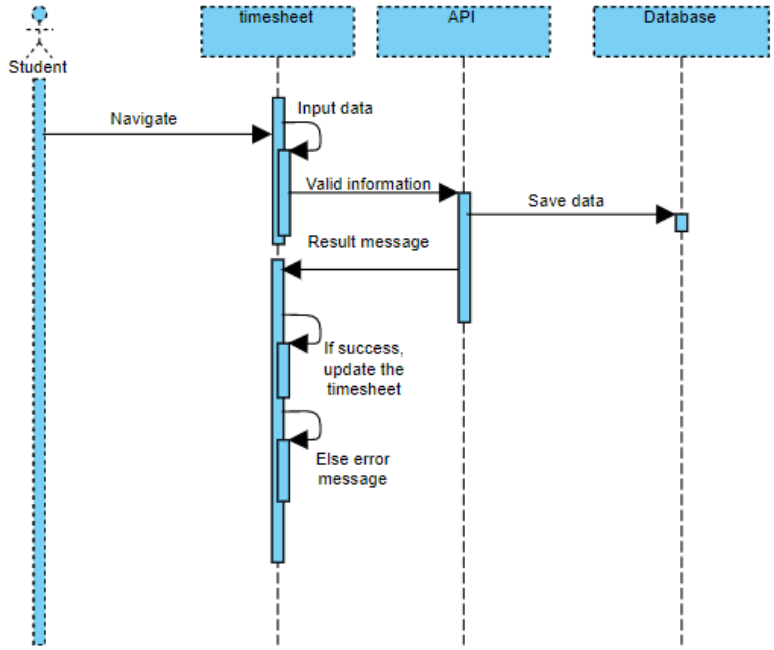

Figure 16. Adding new work session Sequence Diagram

As shown in Figure 16, the user navigates to the timesheet page, and submits the appropriate time. This information will be sent to the API to validate and save to the database. Upon successful addition, the timesheet within the page will be refreshed to reflect the change. An error message will be shown in case of an error.

3.5.6. Removing Work Session Sequence Diagram

Figure 17 below demonstrates the process of removing the user's work sessions.

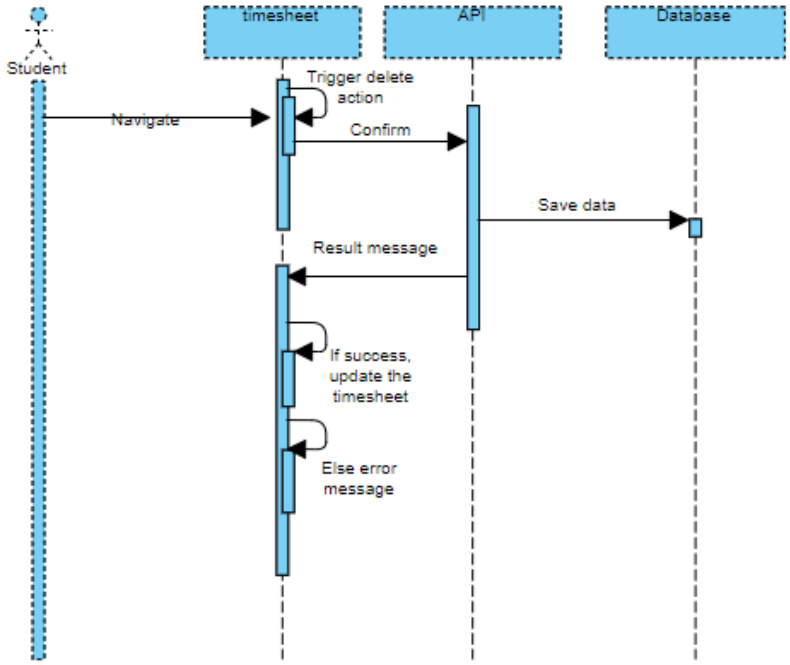

Figure 17. Removing work session Sequence Diagram.

The steps required to remove a work session is simple. The user navigates to the timesheet page, presses the X icon located on top of the card containing the work session, and triggers the delete action. Upon a short confirmation prompt, the client will request the API to remove the work session and send back the result. Upon success, the timesheet will be updated, or display an error message upon failure.

4. DATABASE AND GUI DESIGN

The database and graphical user interface used in the application are discussed in this chapter.

4.1. Database

The application utilizes MySQL as its database. The server-side API can also be customized to use different sources of database by adding appropriate database providers in Startup.cs file.

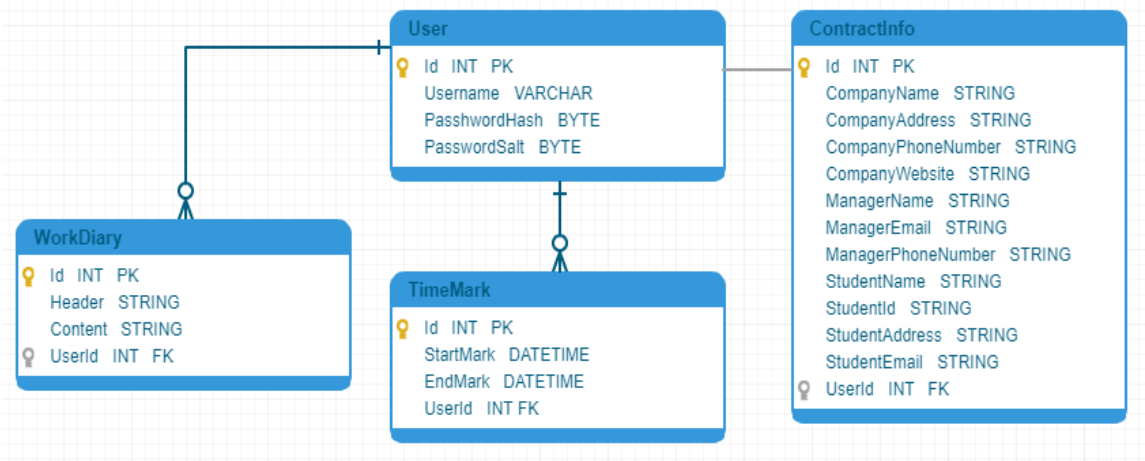

Figure 18. ER diagram

- As shown on the ER diagram, the User table and its primary key ID are connected to all remaining table.
- The ContractInfo table has a one-to-one relationship with the User table. This means one user can only have one contract information at any time. Its UserId attribute is a foreign key connecting to ID attribute from the User table.
- The TimeMark table and the WorkDiary table have many- to- one relationship with the User table. Their UserID attributes are connected to the ID attribute from the User table as a foreign key. This translates to one user

being able to have multiple TimeMarks (or work session), and multiple WorkDiaries at the same time.

4.2. GUI Design

The first page when a new user lands on the application will be the authentication page, where the user can either login as an existing user, or register a new account.

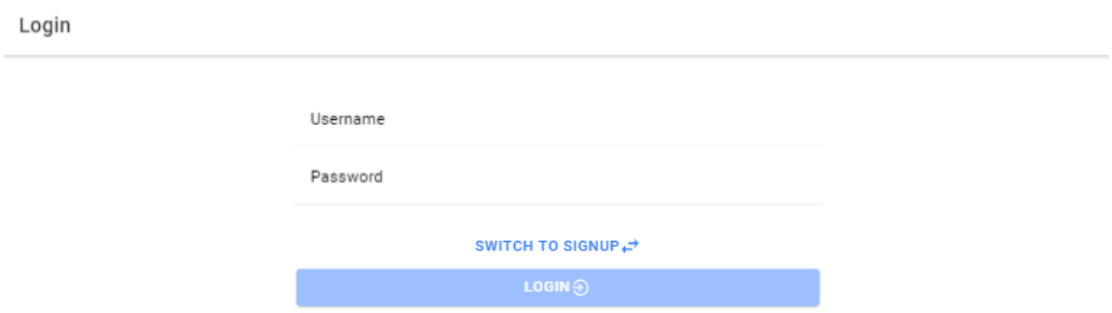

Figure 19. Login form

Login into the application require a username and password pair. The “LOGIN” button is unclickable until both text fields are filled. If a user is new to the application, they can sign up to the application with the “SWITCH TO SIGNUP” button

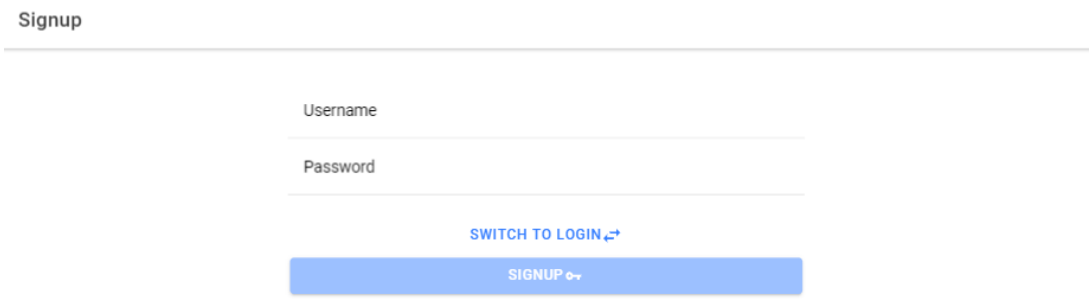

Figure 20. Sign up form

The previous figure demonstrates how “sign ups” are handled. “Login” and “Signup” share a single page with an identical form consisting of two input fields: “Username” and “Password”.

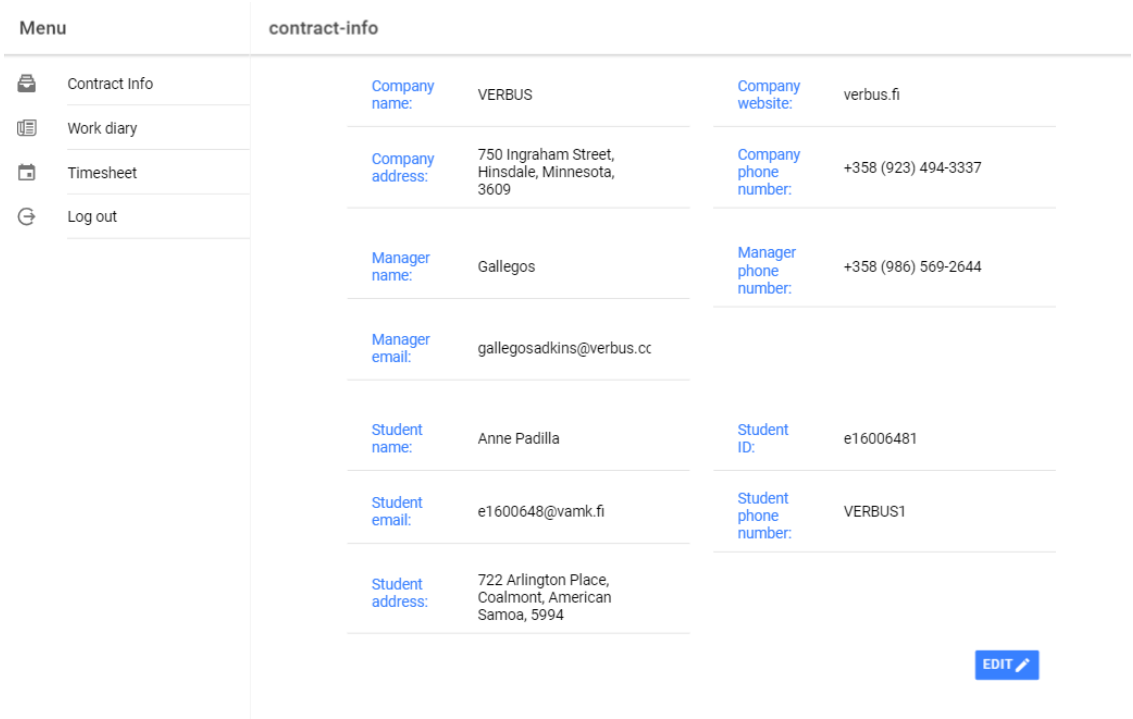

Figure 21. ContractInfo desktop page

There is a link to a switch between the two modes for the user to authenticate. The registration process is kept at minimum requirements, offering new users a seamless way to gain access to the application. The UI is identical on the desktop and Android versions of the client application.

Additional information of the user can be supplemented subsequently in "contract-info-edit" page as shown in Figure 22 and Figure 24 located below.

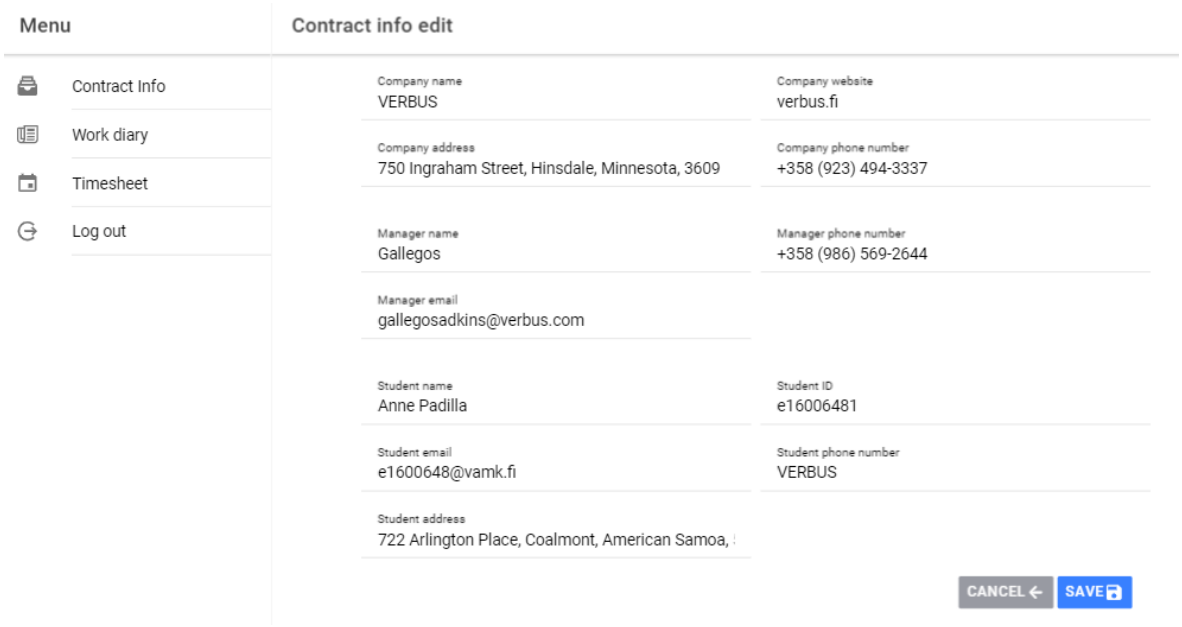

Figure 22. ContractInfoEdit desktop page

The "ContractInfo" page shown in Figure 21 allows the user to review the current information of their contract. This page also serves as the landing page of the application. Information can be edited using the "edit" button that is linked to "ContractInfoEdit" page. Once users have finished editing the information within the available fields, the user can save the data using the "save" button, or cancel the changes using the "cancel" button. Both buttons will redirect the user back to "ContractInfo" page.

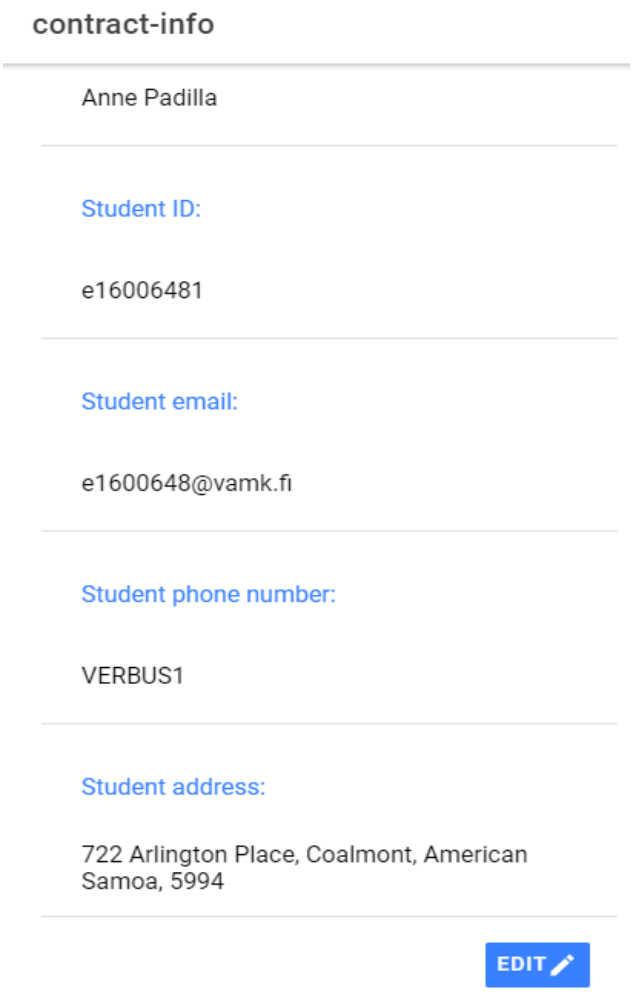

Figure 23. ContractInfo android page

There is a slightly different view of the application on the Android version, as demonstrated by the figure below.

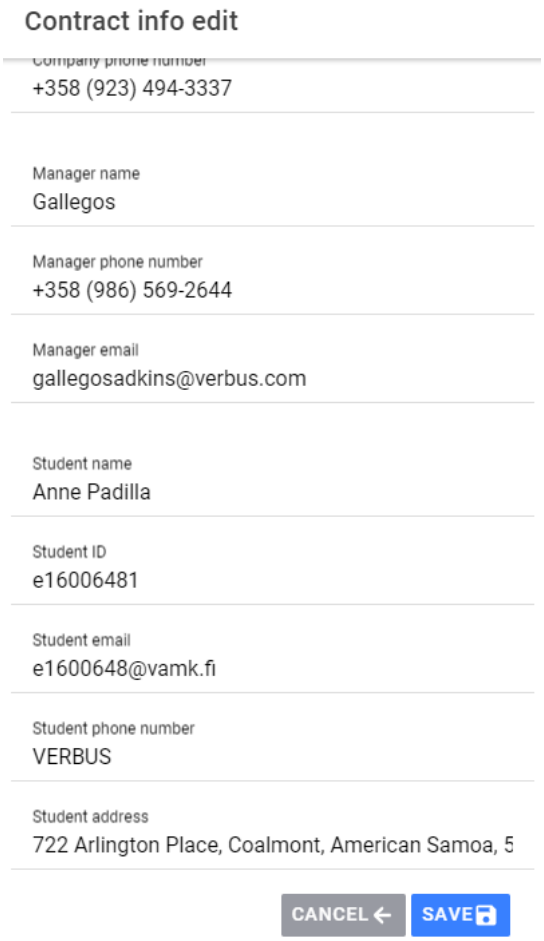

Figure 24. ContractInfoEdit Android page

As shown in the figure, the fields containing the data are arranged as a column in contrast to the desktop version. The user can scroll to the information on a mobile phone and take advantage of the available space to work adequately with devices with small screens. However, the UI of the application remains very similar across the platforms.

Navigation to other pages can be done with the side panel located to the left of the application. This menu is hidden on the Android version and can be accessed by sliding the panel in from the left. Figure 25 below demonstrates how this panel will look on a mobile phone.

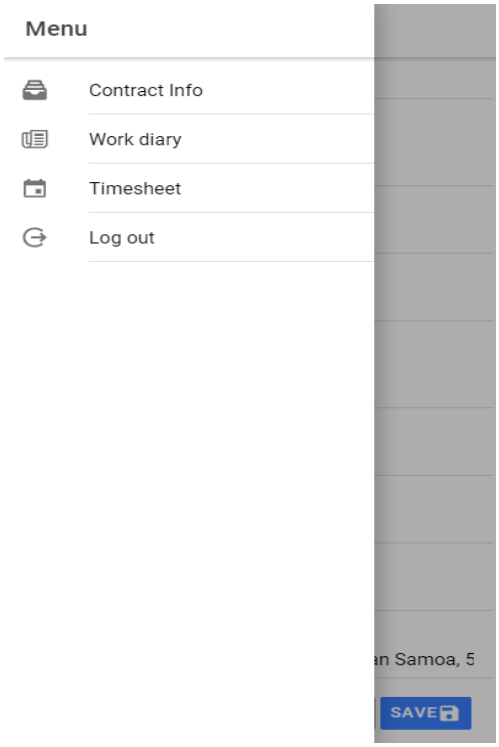

Figure 25. Navigation menu on Android

The work diary (Figure 26) can be viewed and registered in the “WorkDiary” and “WorkDiaryEdit” respectively.

Previously registered work diaries are arranged as panel within “WorkDiary” page and can be deleted with the “X” icon or edited with the “Pen” icon located on top of the panels.

A new work diary can be added by pressing the “Plus” icon situated on top of the work diary page.

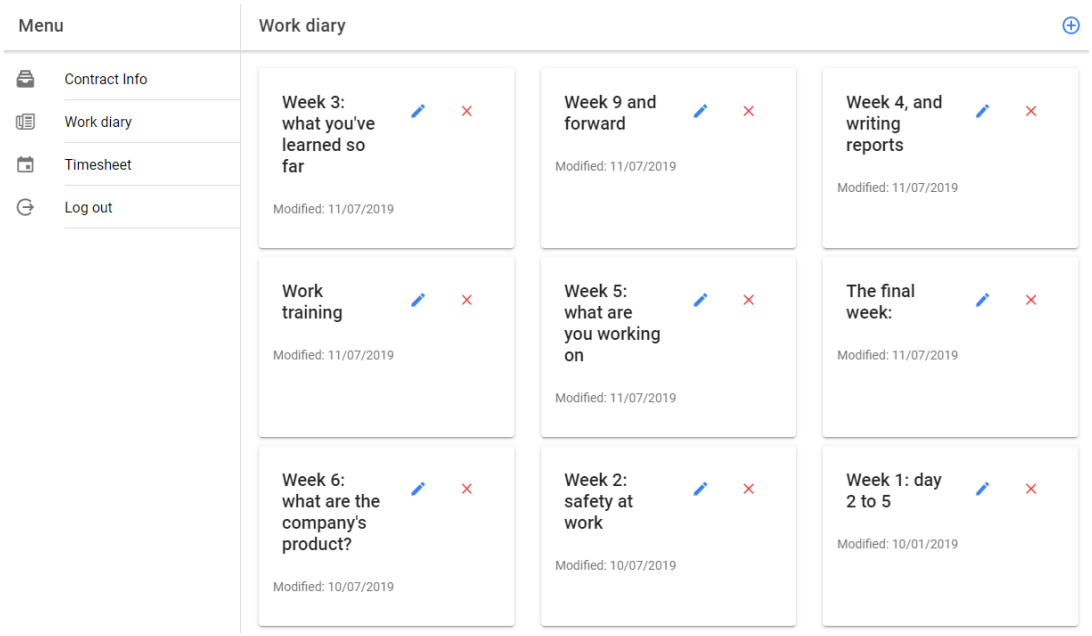

Figure 26. WorkDiary page

On the "WorkDiaryEdit" page (Figure 27), the user can register the title of the work diary.

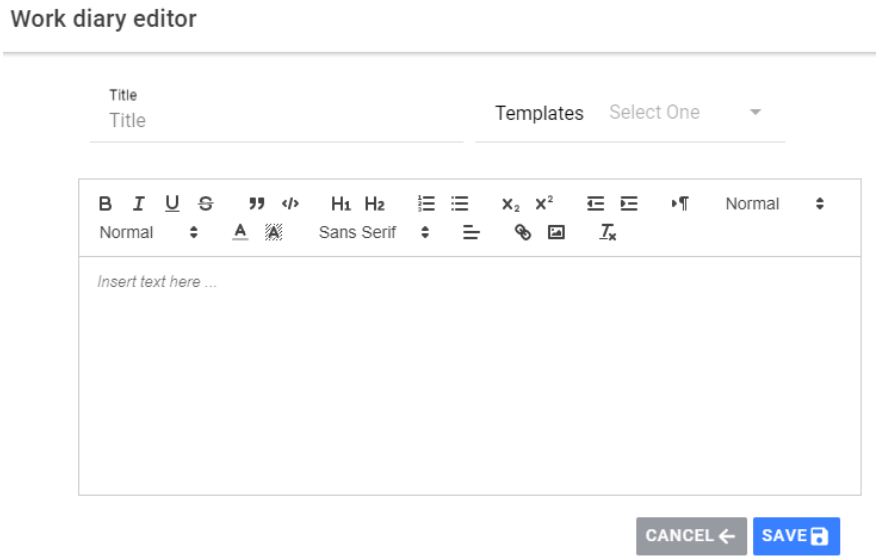

Figure 27. WorkDiaryEdit desktop page

The text editor of the page supports rich document and support operations, such as bold, italic, list, URL, font type, and image. The user can also select from a list of

predefined templates containing common questions that are needed to make a good practical training report.

The Android version applies minimal changes to the “WorkDiary” page, allowing better interaction with the application on smaller screens, as shown in Figure 28.

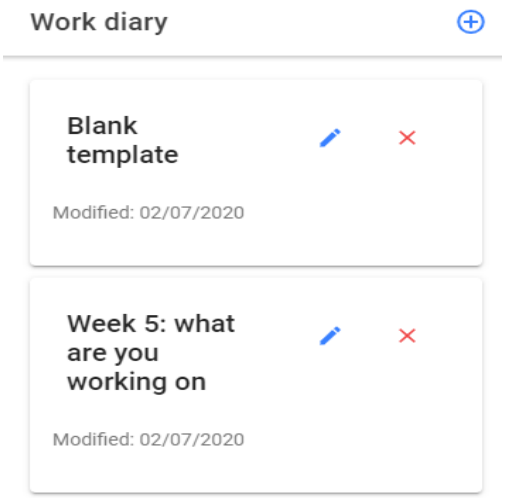

Figure 28. WorkDiary Android page

As WorkDiary pages on Android (Figure 29), the WorkDiaryEdit page is also very similar to its desktop counterpart.

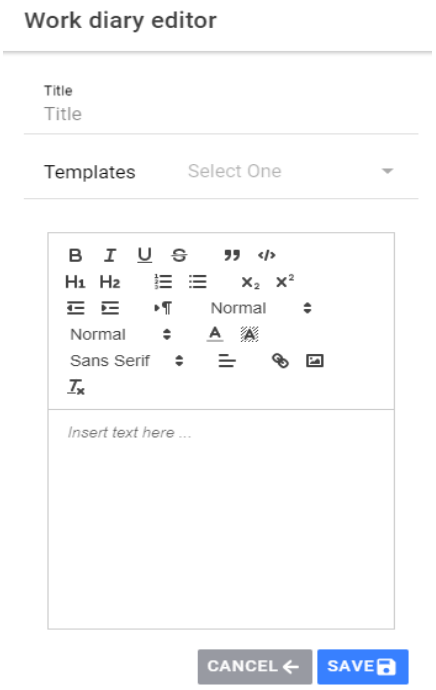

Figure 29. WorkDiaryEdit Android page

A new work session can be added on the "TimeSheet" page where the application will automatically calculate the total working hours and credit units from user input. The application removes thirty minutes from any work sessions that are six-hour long or more as time as a lunch break.

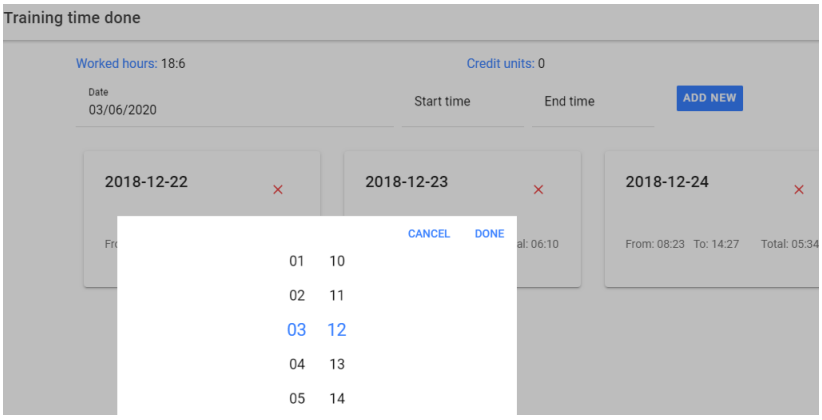

Figure 30. TimeSheet desktop page

Time inputs are in form of date and time picker. The date picker is limited to the current date as upper limit, and up to five years in the past as a lower limit.

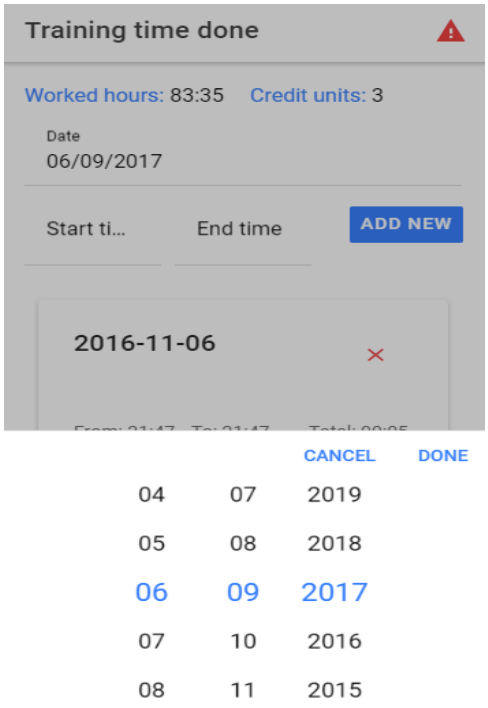

Figure 31. Timesheet Android page

As with the previous pages, small changes are made to accommodate smaller screen on Android.

5. IMPLEMENTATION

5.1. General Structure of the Project

This chapter gives a general overview of the implementation of the application.

5.1.1. Ionic Application

This section discusses the structure of the Ionic client application

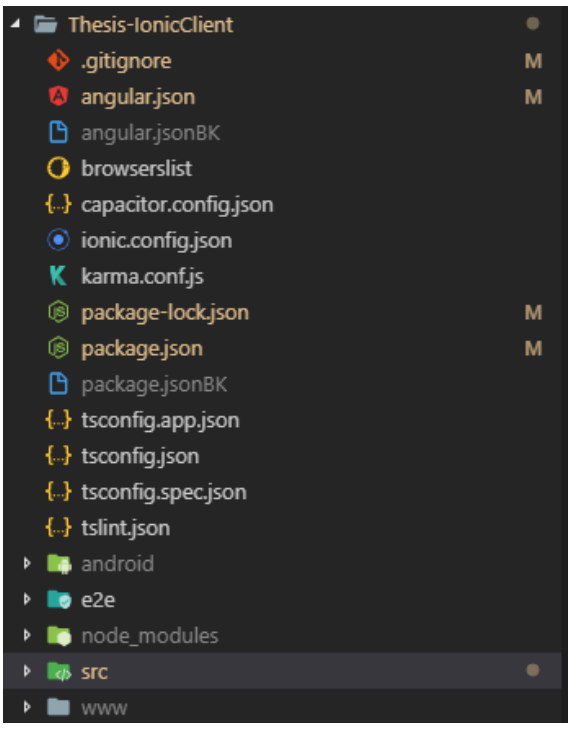

Figure 32. Basic Ionic Application Structure

By default, Ionic Framework and node generate environment files to help with the developing process. All application code is located in the “src” folder

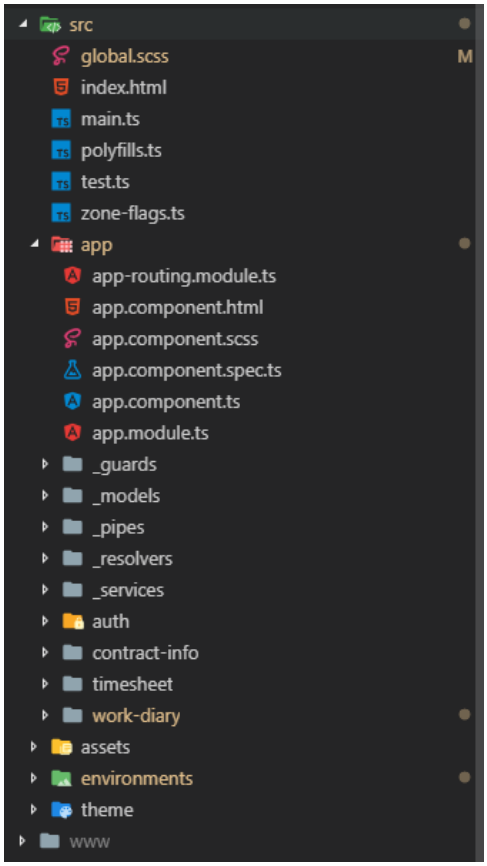

Figure 33. Client structure

Under the "src" directory, there are the main files that support the application. "Index.html" is the entry point of the application. Ionic with the Angular support will automatically convert components in the Angular team, or pages in the Ionic term into a SPA. "app" is the main component that binds all the smaller components, such as "auth", "contract-info", "timesheet" and "work-diary" together, some of which can be used to bind smaller component together. Directories whose names start with an underscore, such as "_guards" contain utility classes that can be used across the application.

5.1.2. Server-side EF Core API

This section discusses the structure of the server-side API.

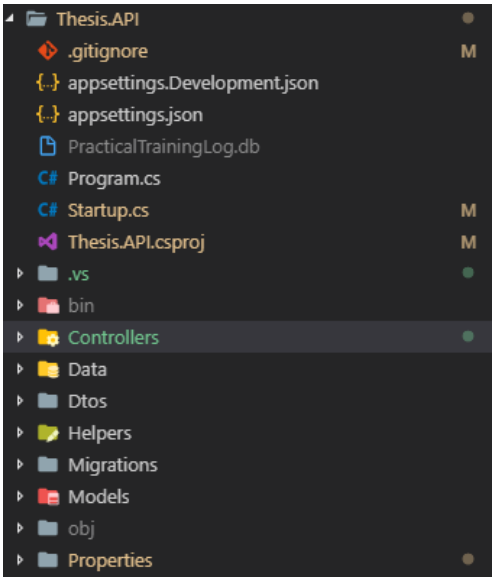

Figure 34. Server-side API structure

- By default, EF Core CLI generates “Properties”, “bin” and “obj” directory, “appsettings.json”, “Program.cs” and “Startup.cs”, containing the configuration of the basic API.
- “Controllers” directory is home to the API controllers which handle the access point for the outer application to connect with the server.
- “Data” directory contains repository implementations which interact with the database.
- “Dtos” directory contains DTO that hand data across the application.
- “Helpers” directory contains utility classes.
- “Models” directory contains models, from which EF Core will base on to implement the database.

5.2. Implementation of the Ionic Client

This section discusses the implementation of the Ionic client application.

5.2.1. Authentication

The user is denied access to the application until a successful authentication. This is done by placing a similar guard on all the routes to the application except for the “auth” page.

```
{
  path: 'contract-info',
  loadChildren: './contract-info/contract-info.module#ContractInfoPageModule',
  canActivate: [AuthGuard],
  resolve: { contractinfo: ContractInfoResolver }
},
```

Code Snippet 7. AuthGuard placed on “contractinfo” page

“AuthGuard” relies on the “AuthService” to check if the user has a valid JWT token, then returns a Boolean which decide if the user is authenticated.

```
loggedIn() {
  const token = localStorage.getItem('token');
  return !this.jwtHelper.isTokenExpired(token);
}
```

Code Snippet 8. AuthService validating login status

On “auth” page, the user can login and register. These operations are handled by the “AuthService” which will send an http request to the API to validate the request. “AuthService” utilizes a third-party library to decode a JWT token.

```
login(model: any) {
  return this.http.post(this.baseUrl + 'login', model).pipe(
    map((response: any) => {
      const user = response;
      if (user) { //if user exist
        localStorage.setItem('token', user.token);
        this.decodedToken = this.jwtHelper.decodeToken(user.token);
      }
    })
  );
```

```
}

register(model: any) {
  return this.http.post(this.baseUrl + 'register', model);
}
```

Code Snippet 9. Implementation of AuthService and its methods

Once the user has successfully logged in, there is a guard on the “Auth” page to prevent the user from accidentally accessing the authenticating form again using canActivate from Angular.

```
{
  path: 'auth',
  loadChildren: './auth/auth.module#AuthPageModule',
  canActivate: [LoggedInGuard]
},
```

Code Snippet 10. LoggedInGuard placed on Auth page

Upon the user logging out of the application, the token containing authenticating information will be removed. This will also redirect the application to the “Auth” page. This is implemented as a service in AuthService and called from app.component.ts. Additional log out implementation will be discussed in section 5.2.10.

```
logOut() {
  const currentRouteConfig = this.router.config.find(f => f.path === this.router.url.substr(1));
  if (this.router.url !== 'contract-info-edit' && !this.router.url.includes('work-diary-edit') && this.router.url !== '') { //log out from these routes are implemented later
    this.implementLogOut();
    this.router.navigate(['/auth']);
  }}
implementLogOut() {
  localStorage.removeItem('token');
  this.menuCtrl.enable(false);
  this.toaster.message('Logged out');
}
```

Code Snippet 11. Implementation of log out as a service

5.2.2. Navigation Menu

Entries on the navigation menu are generated using a loop from a predefined object consisting of the text, icon and URL of the entry which are then bonded to the properties of the HTML element. This navigation menu is placed in the “app” component to ensure that the user is able to access the navigation from any point of the application.

```
<ion-menu-toggle auto-hide="false" *ngFor="let p of appPages">
  <ion-item [routerDirection]="'root'" [routerLink]="[p.url]">
    <ion-icon slot="start" [name]="p.icon"></ion-icon>
    <ion-label>
      {{p.title}}
    </ion-label>
  </ion-item>
</ion-menu-toggle>
```

Code Snippet 12. Implementation of navigation menu

5.2.3. User Action Notification

On any user interaction which requires access to the API, a notification which is called toast in this application is a display, informing the user of the result.

These toasts are generated from the “ToasterService” as a service to be injected to other components.

```
private async generateToaster(message: string, type: string) {
  let color: string;
  switch (type) {
    case 'success': {
      color = 'success';
      break;
    }
    case 'error': {
      color = 'danger';
      break;
    }
```

```
      case 'warning': {
        color = 'warning';
        break;
      }
      case 'message': {
        color = 'secondary';
        break;
      }
    }
    const toast = await this.toastCtrl.create({
      message,
      duration: 2000,
      position: 'bottom',
      color,
      animated: true
    });
    toast.present();
  }
```

Code Snippet 13. Generate a toast

5.2.4. ContractInfo Page

User contract information is delivered to the page using a resolver called "ContractInfoResolver", which ensures that the information is available on the page and is always up-to-date.

```
  resolve(): Observable<ContractInfo> {
    return this.userService.getContractByUserId(this.authService.decodedToken.nameid).pipe(
      catchError(error => {
        this.toaster.error('Problem retrieving your data');
        this.router.navigate(['/contract-info']);
        return of(null);
      })
    );
  }
```

Code Snippet 14. Resolver return an Observable of type ContractInfo

This resolver in turn collects the information from the “UserService” with the authentication information received from “AuthService”.

```
getUser(id): Observable<User> {
  return this.http.get<User>(this.baseUrl + 'users/' + id);
}
getContractByUserId(id): Observable<ContractInfo> {
  return this.http.get<ContractInfo>(this.baseUrl + 'users/contractbyid/' + id);
}
updateUserContractInfo(id: number, contractInfo: ContractInfo) {
  return this.http.put(this.baseUrl + 'users/contractbyid/' + id, contractInfo);
}
```

Code Snippet 15. Implementation of the UserService and its function

5.2.5. ContractInfoEdit Page

Angular Reactive Form is used in the ContractInfoEdit page to allow the validation of user inputs. On submit, the page will inspect, and forward changes made to the data using “UserService”. Upon completion, the user will be navigated to the “ContractInfo” page with the updated information, and a “ToasterService” will informs the user if the operation was successful

```
      managerPhoneNumber: new FormControl(this.contractinfo.managerPhoneNumber),
      managerEmail: new FormControl(this.contractinfo.managerEmail, {
        validators: [Validators.email]
      }),
      studentName: new FormControl(this.contractinfo.studentName, {
        validators: [Validators.required]
      }),
```

Code Snippet 16. Reactive Form Input Field with validation

```
      if (JSON.stringify(this.contractinfo) === JSON.stringify(this.newContractInfo)) {

        loadingEl.dismiss();

        this.toaster.message('No changes detected.');
```

```
        this.form.reset(this.contractinfo);

        this.router.navigate(['contract-info']);

      } else {

        this.userService.updateUserContractInfo(this.authService.decodedTo-
ken.nameid, this.newContractInfo).subscribe(next => {

          loadingEl.dismiss();

          this.toaster.success('Successfully update your contract.');

          this.form.reset(this.newContractInfo);

          this.router.navigate(['contract-info']);

        }, error => {

          loadingEl.dismiss();

          this.toaster.error(error);

        });

      }
```

Code Snippet 17. Validate and submit new information

5.2.6. WorkDiary Page

The log of work diary is made up of a list of cards containing ID, header, and last modified time of the diaries. This list is generated from an array of type “WorkDiary” using Angular “for” loop. The size of each card is bonded to a variable called “numberOfCol”, which allows future customization on how many cards there can be in a row.

```
<ion-row *ngFor="let dList of formattedDiaryList ">
  <ion-col *ngFor="let d of dList" size-Xs='12' [sizeMd]="12 / numberOfCol">
    <ion-card class='diaryCard'>
      <ion-card-header>
        <ion-card-title>
          <ion-grid>
```

```
                <ion-row>
                    <ion-col>{{d.header}}</ion-col>
                    <ion-col>
                        <ion-button fill='clear' color='danger' (click)='re-
moveDiary(d.id)'
                            class='ion-float-right'>
                            <ion-icon name="close"></ion-icon>
                        </ion-button>
                        <ion-button fill='clear' color='primary' class='ion-float-right'
                            [routerLink]="['/work-diary-edit', d.id]" routerLinkActive =
"router-link-active">
                            <ion-icon name="create"></ion-icon>
                        </ion-button>
                    </ion-col>
                </ion-row>
            </ion-grid>
        </ion-card-title>
    </ion-card-header>
    <ion-card-content>
        Modified: {{d.lastEdited}}
    </ion-card-content>
  </ion-card>
 </ion-col>
</ion-row>
```

Code Snippet 18. List of WorkDiary

Due to the page entanglement with the WorkDiaryEdit page, there is a need for data to be fetched and updated directly from the route, similarly to the ContractInfo page, while also having a way to fetch new data independently. The solution for this is illustrated in Code snippet 11 below, along with the method the page needs to remove registered work diaries.

```
 ngOnInit() {
  this.route.data.subscribe(data => { //subscribe and get information from the
route
   this.diaryList = data.diaries;
   this.formatWorkDiaryList(this.diaryList);
  });
```

```
  }
  fetchDiary() { //fetch new data
      this.workDiaryService.getDiariesFromUser(this.authService.decodedTo-
ken.nameid).subscribe(data => {
      this.diaryList = data;
      this.formatWorkDiaryList(this.diaryList);
    }, error => {
      this.toaster.error('Failed to fetch data.');
    });
  }
  removeDiary(id: number) {
    this.alertService.confirmAlert('You sure you want to delete this?', () => {
      this.workDiaryService.deleteDiary(this.authService.decodedToken.nameid,
id).subscribe(() => {
        this.toaster.success('Diary removed.');
        this.fetchDiary();
      }, error => {
        this.toaster.error('Failed to remove diary.');
      });
    });
  }
  formatWorkDiaryList(diaryList: WorkDiary[]) {
    this.formattedDiaryList = this.chunkPipe.transform(
      this.reversePipe.transform(diaryList), this.numberOfCol
    );
  }
```

Code Snippet 19. Methods to manage the list of WorkDiary

5.2.7. WorkDiaryEdit Page

"WorkDiaryEdit" contains an Angular Reactive Form, from which the user can input information of the work diary. The main editor is a ngx-quill component which was configured to allow predefined form to be patched in the main input.

```
    @ViewChild('editor', {
    static: true
  }) editor: QuillEditorComponent;
  config = {
```

```
  toolbar: {
    container:
    [
        ['bold', 'italic', 'underline', 'strike'],
        ['blockquote', 'code-block'],
        [{ header: 1 }, { header: 2 }],
        [{ list: 'ordered' }, { list: 'bullet' }],
        [{ script: 'sub' }, { script: 'super' }],
        [{ indent: '-1' }, { indent: '+1' }],
        [{ direction: 'rtl' }],
        [{ size: ['small', false, 'large', 'huge'] }],
        [{ header: [1, 2, 3, 4, 5, 6, false] }],
        [{ color: [] }, { background: [] }],
        [{ font: [] }],
        [{ align: [] }],
        ['link', 'image']   ,
        ['clean']
    ], }};
  onTemplateChange(template) {
    this.editorForm.get('title').setValue(template);
this.editorForm.get('content').patchValue(JSON.stringify(predefinedForms.get-
FormContent(template).content));
    this.editor.quillEditor.setSelection(this.editor.quillEditor.getLength(), 0, 'api');
  }
```

Code Snippet 20. ngx-quill configuration

Depending on from which entry point the user accesses the page, the application will use a dedicated method to add a new work diary or edit an existing work diary with a diary ID provided by the route. This ID does not require a resolver to access and can be seen directly on the URL.

Upon successful add or edit, the user will be directed to “WorkDiary” page with newly updated information.

```
  addNewDiary() {
    this.loadingCtrl.create({
      message: 'Updating your info...',
```

```
      keyboardClose: true
    }).then(
      loadingEl => {
        loadingEl.present();
        const today = new Date();
        const dd = String(today.getDate()).padStart(2, '0');
        const mm = String(today.getMonth() + 1).padStart(2, '0'); // January is 0!
        const yyyy = today.getFullYear();
        const newDiary: WorkDiary = {
          header: this.editorForm.get('title').value,
          content: this.editorForm.get('content').value,
          lastEdited: mm + '/' + dd + '/' + yyyy
        };
        this.workDiaryService.addDiary(this.authService.decodedToken.nameid,
newDiary).subscribe(() => { //calling method from the workDiaryService to add new
entry
          loadingEl.dismiss();
          this.toaster.success('New entry added.');
          this.editorForm.reset();
          this.router.navigate(['/work-diary']);
        }, error => {
          loadingEl.dismiss();
          this.toaster.error('Failed to add new entry.');
          this.toaster.error(error);
        });});}
```

Code Snippet 21. Adding new work diary

```
  editDiary() {
    this.loadingCtrl.create({
      message: 'Updating your info...',
      keyboardClose: true
    }).then(
      loadingEl => {
        loadingEl.present();
        const today = new Date();
        const dd = String(today.getDate()).padStart(2, '0');
        const mm = String(today.getMonth() + 1).padStart(2, '0');
        const yyyy = today.getFullYear();
        const newDiary: WorkDiary = {
```

```
      header: this.editorForm.get('title').value,
      content: this.editorForm.get('content').value,
      lastEdited: mm + '/' + dd + '/' + yyyy
    };
    this.workDiaryService.updateDiaryEntry(this.authService.decodedTo-
    ken.nameid, this.diaryId, newDiary).subscribe(() => { //calling service to mod-
    ify entry
      loadingEl.dismiss();
      this.toaster.success('Entry updated.');
      this.editorForm.reset();
      this.router.navigate(['/work-diary']);
    }, error => {
      loadingEl.dismiss();
      this.toaster.error('Failed to update entry.');
    });});}
```

Code Snippet 22. Editing existing work diary

5.2.8. TimeSheet Page

The “TimeSheet” page is divided into two parts.

The first part includes main inputs from which the user can pick the correct date and time. These inputs are made with the Angular Form Module and Ionic date-time picker. Calculation done with the DateTime value from these input uses the third-party library Date-Fns.

```
addTimemark() {
  if (this.date !== undefined && this.startTime !== undefined && this.endTime !==
undefined) {
    if (compareAsc(new Date(this.endTime), new Date(this.startTime) ) > 0) {
      const newTimeMark: TimeMark = {
        startMark: format(new Date(this.date), 'yyyy-MM-dd') + ' ' + format(new
Date(this.startTime).setSeconds(0), 'HH:mm:ss'),
        endMark: format(new Date(this.date), 'yyyy-MM-dd') + ' ' + format(new
Date(this.endTime).setSeconds(0), 'HH:mm:ss'),
      };
      this.timeMarkService.addTimeMark(this.authService.decodedToken.nameid,
newTimeMark).subscribe(() => { // calling service to add new time entry
        this.fetchData(); // update the timesheet list displayed on the page.
        this.toaster.success('New entry added.');
```

```
          this.router.navigate(['/timesheet']);
        });
      } else { this.toaster.error('Invalid time input'); }
    } else { this.toaster.error('Invalid time input'); }}
```

Code Snippet 23. Adding a new session

Upon entering the view the user's work session information is fetched with the "TimeMark" service.

```
getTimeSheetByUserId(id): Observable<TimeMark[]> {
   return this.http.get<TimeMark[]>(this.baseUrl + 'users/' + id + '/timemarks');
 }
 deleteTimeMark(userId: number, id: number) {
  return this.http.delete(this.baseUrl + 'users/' + userId + '/timemarks/' + id);
 }
 addTimeMark(userId: number, timemarks: TimeMark) {
  return this.http.post(this.baseUrl + 'users/' + userId + '/timemarks', time-
marks);
 }
```

Code Snippet 24. Methods implemented in TimeMarkService

The result data will be passed onto the child component embedded within the page called "TimeSheetDetail". This allows the component to generate the list of work sessions for the parent page.

```
<app-timesheetdetail *ngIf="timesheet" [timesheet]="timesheet"
      (totalHrsCalculated)="totalHrsCalculated($event)"></app-timesheetdetail>
```

Code Snippet 25. Using and binding child component

5.2.9. TimeSheetDetail Component

The "TimeSheetDetail" component is responsible for formatting the input received from the API and displaying the information in a readable format. This is done in the "formatRawTimeMark" function.

```
 formatRawTimeMark(timeMark: TimeMark[]) { //format the data into displayable
form
   this.formattedTime = [];
```

```
  timeMark.forEach(time => {
    let workTime = differenceInMinutes(new Date(time.endMark), new Date(time.startMark));
    if (workTime >= 6 * 60) {
      workTime = differenceInMinutes(new Date(subMinutes(new Date(time.endMark), 30)), new Date(time.startMark));
    }
    this.formattedTime.push({
      id: time.id,
      date: format(new Date(time.startMark), 'yyyy-MM-dd'),
      startTime: format(new Date(time.startMark), 'HH:mm'),
      endTime: format(new Date(time.endMark).setSeconds(0), 'HH:mm'),
      workHrs: workTime.toString()
    });
    this.formattedTime.sort((a, b) => { //sort the list in order
      if (a.date < b.date) {
        return -1;
      } else if (a.date > b.date) {
         return 1;
      } else {
        if (a.startTime < b.startTime) {
          return -1;
        } else if (a.startTime > b.startTime) {
          return 1;
        } else {
          return 0;
        }
      }
    });
  });
```

Code Snippet 26. Formatting data input

The component will trigger an event back to the parent page when it has finished calculating total work hours.

```
this.totalHrs = this.calculateTotalHrs(this.formattedTime);
this.totalHrsCalculated.emit(this.formatTotalHrsReturn(this.totalHrs));
```

Code Snippet 27. Sending total hour back to the main page

Removing a work session is a handled by “removeTimeMark” function, using the “TimeMark” service

```
removeTimeMark(id: number) {
   this.alertService.confirmAlert('You sure you want to delete this?', () => {
     this.timeMarkService.deleteTimeMark(this.authService.decodedToken.nameid,
id).subscribe(() => { //calling the service to remove timesheet
       this.timesheet.splice(this.timesheet.findIndex(tm => tm.id === id), 1);
       this.toaster.success('Timemark removed.');
       this.ngOnChanges();
     }, error => {
       this.toaster.error('Failed to remove timemark.');
     });
   });
 }
```

Code Snippet 28. Removing work session

5.2.10. Preventing Accidental Page Navigation with CanDeactivate

Additional guards are applied on the “ContractInfoEdit” and “WorkDiaryEdit” page. These guards are similar and will prevent any navigation attempt when detecting changes without a prior saving made. The guards are also responsible for logging out users from the “ContractInfoEdit” and “WorkDiaryEdit”.

```
canDeactivate(page: ContractInfoEditPage, currentRoute: ActivatedRouteSnap-
shot,
             currentState: RouterStateSnapshot, nextState?: RouterStateSnap-
shot): boolean|Promise<boolean> {
      if (page.form.dirty) {
         return this.sendAlert().then(result => {
            if (nextState.url === '/auth') { //if attempting to log out.
               if (result) {
                  this.authService.implementLogOut();
               }
            }
            return result;
```

```
            });
        } else if (!page.form.dirty) {
            if (nextState.url === '/auth') {
                this.authService.implementLogOut();
            }
        }
        return true;
    }
    async sendAlert() { //confirmation alert when trying to navigate with dirty forms
        return new Promise<boolean>(async (resolve) => {
            const alert = await this.alertCtrl.create({
                message: 'Are you sure? Unsaved changes will be lost',
                buttons: [{
                    text: 'Cancel',
                    handler: () => {
                        resolve(false);
                    }
                }, {
                    text: 'Confirm',
                    handler: () => {
                        resolve(true);
                    }
                }]
            });
            await alert.present();
        });
    }
```

Code Snippet 29. PreventUnsavedChanges guard and its alert function

```
const routes: Routes = [
  {path: '',
    component: ContractInfoEditPage,
    canDeactivate: [PreventUnsavedChanges]
  }];
```

Code Snippet 30. Applying canDeactivate guard to the page

5.2.11. Setting up API endpoint address

The API end point addresses are stored in “environment.ts” for the development mode and “environment.prod.ts” for the production mode. These files are located under the “environment” directory. The API address esstored in these files are used across the application.

```
export const environment = {
production: true,
apiUrl: 'http://localhost:5000/api/'
};
```

Code Snippet 31. Environment settings

5.3. Implementation of the Server-side API

This section discusses the implementation of the backend API.

5.3.1. Models

Model classes located under the “Models” directory form the structure of the application database. The relationship of the table “User” is demonstrated by designating the “User” model to have properties of other models as their types, such as “TimeMark” and “WorkDiary” model. In turn, these models are required to have the property of type “User” and property “UserId” as a foreign key.

```
public class User
{
    public int Id { get; set; }
    public string Username { get; set; }
    public byte[] PasswordHash { get; set; }
    public byte[] PasswordSalt { get; set; }
    public ContractInfo { get; set; }
    public ICollection<TimeMark> TimeMarks { get; set; }
    public ICollection<WorkDiary> WorkDiaries { get; set; }
}
```

Code Snippet 32. User model

```
public class WorkDiary
{
    public int Id { get; set; }
    public string Header { get; set; }
    public string Content { get; set; }
    public string LastEdited { get; set; }
    public User { get; set; }
    public int UserId { get; set; }
}
```

Code Snippet 33. WorkDiary model

5.3.2. Automapper

Automapper is a third-party library which automatically maps properties between objects. Configuration is required for Automapper to operate correctly. This correction is called AutomapperProfile.

```
public class AutoMapperProfile : Profile
  {
     public AutoMapperProfile()
     {
        CreateMap<User, UserInfoDto>();
        CreateMap<ContractInfo, ContractInfoForUserDto>();
        CreateMap<ContractInfoForUserDto, ContractInfo>()
           .ForMember(c => c.Id, opt => opt.Ignore())
           .ForMember(c => c.User, opt => opt.Ignore())
           .ForMember(c => c.UserId, opt => opt.Ignore());
        CreateMap<TimeMark, TimeMarksForUserDto>();
        CreateMap<TimeMarksForUserDto, TimeMark>()
           .ForMember(tm => tm.Id, opt => opt.Ignore())
           .ForMember(tm => tm.User, opt => opt.Ignore())
           .ForMember(tm => tm.UserId, opt => opt.Ignore());
        CreateMap<WorkDiary, WorkDiaryDto>();
        CreateMap<WorkDiaryDto, WorkDiary>()
           .ForMember(wd => wd.Id, opt => opt.Ignore())
           .ForMember(wd => wd.User, opt => opt.Ignore())
           .ForMember(wd => wd.UserId, opt => opt.Ignore());
```

```
        CreateMap<MiniWorkDiaryDTO, WorkDiary>()
            .ForMember(wd => wd.Id, opt => opt.Ignore())
            .ForMember(wd => wd.User, opt => opt.Ignore())
            .ForMember(wd => wd.Content, opt => opt.Ignore())
            .ForMember(wd => wd.UserId, opt => opt.Ignore());
        CreateMap<ContractInfo, User>()
            .ForMember(u => u.Id, opt => opt.Ignore())
            .ForMember(u => u.Username, opt => opt.Ignore())
            .ForMember(u => u.PasswordHash, opt => opt.Ignore())
            .ForMember(u => u.PasswordSalt, opt => opt.Ignore())
            .ForMember(u => u.TimeMarks, opt => opt.Ignore())
            .ForMember(u => u.WorkDiaries, opt => opt.Ignore())
            .ForMember(u => u.ContractInfo, opt => opt.MapFrom(ci => ci));
    }
}
```

Code Snippet 34. Automapper profile

5.3.3. Repository and DataContext

Under the "Data" directory there are the repositories and data context of the application.

"DataContext.cs" is responsible for scaffolding the database for the application.

There are two repositories supporting the application. "AuthRepository.cs" contains methods to verify the user account, registering a new user and creating a password hash to protect the password in the database.

```
public async Task<User> Login(string username, string password)
    {
        var user = await this.context.Users.FirstOrDefaultAsync(x =>x.Username ==
username);
        if (user == null)
            return null;
        if (!VerifyPasswordHash(password, user.PasswordHash, user.PasswordSalt))
            return null;
        return user;}
```

Code Snippet 35. Verify user

```
public async Task<User> Register(User, string password)
      {
         byte[] passwordHash, passwordSalt;
         CreatePasswordHash(password, out passwordHash, out passwordSalt);
         user.PasswordHash = passwordHash;
         user.PasswordSalt = passwordSalt;
         await this.context.Users.AddAsync(user);
         await this.context.SaveChangesAsync();
         return user;
}
private void CreatePasswordHash(string password, out byte[] passwordHash, out byte[] passwordSalt)
      {
         using(var hmac = new System.Security.Cryptography.HMACSHA512()) {
            passwordSalt = hmac.Key;
            passwordHash = hmac.ComputeHash(System.Text.Encoding.UTF8.Get-Bytes(password));
         }
}
```

Code Snippet 36. Registering new user and create password hash

UserActionRepository contains methods which are used to access, add and remove entities from the user, such as a work diary and contract info.

```
  public async Task<IEnumerable<TimeMark>> GetTimeMarks(int userId)
        {
         var timemarks = await this.context.TimeMarks.Where(tm => tm.UserId == userId).ToListAsync();
           return timemarks;
        }


        public async Task<TimeMark> GetTimeMarksById(int tmId)
        {
           var timemark = await this.context.TimeMarks.Where(tm => tm.Id == tmId).FirstOrDefaultAsync();
           return timemark;
  }
```

Code Snippet 37. Retrieve user work session

5.3.4. Controllers

Controllers are the end point of the API where remote clients can access and request, update, and add new resources.

“AuthController” allows the client to log in and register a new user

```
[HttpPost("login")]
public async Task<IActionResult> Login(UserForLoginDto userForLoginDto)
{
var userFromRepo = await this.repo.Login(userForLoginDto.Username, userFor-
LoginDto.Password);
        if (userFromRepo == null)
           return Unauthorized();
        var claims = new[]
        {
        new Claim(ClaimTypes.NameIdentifier, userFromRepo.Id.ToString()),
      new Claim(ClaimTypes.Name, userFromRepo.Username)
        };

         var key = new SymmetricSecurityKey(Encoding.UTF8
         .GetBytes(this.config.GetSection("AppSettings:Token").Value));

Varcreds = new SigningCredentials(key, SecurityAlgorithms.HmacSha512Signa-
ture);
        var tokenDescriptor = new SecurityTokenDescriptor{
           Subject = new ClaimsIdentity(claims),
           Expires = DateTime.Now.AddDays(1),
           SigningCredentials = creds
        };
        var tokenHandler = new JwtSecurityTokenHandler();
        var token = tokenHandler.CreateToken(tokenDescriptor);
        return Ok(new
        {
           token = tokenHandler.WriteToken(token)
        });}
```

Code Snippet 38. AuthController implementation of login

“UsersController”, “TimeMarkController” and “WorkDiaryController” provide methods to operate on user information, such as a work diary and work session.

```
 [HttpPost]
public async Task<IActionResult> AddNewWorkDiary(int userId, WorkDiaryDto workDiaryDto)
      {
         if (userId != int.Parse(User.FindFirst(ClaimTypes.NameIdentifier).Value))
         {return Unauthorized();}
         var newDiary = this.mapper.Map<WorkDiary>(workDiaryDto);
         var userFromRepo = await this.repo.GetUser(userId);
         userFromRepo.WorkDiaries.Add(newDiary);
         if (await this.repo.SaveAll())
            return NoContent();
         throw new System.Exception("Adding work diary failed on save");
      }
 [HttpPost("{id}")]
public async Task<IActionResult> UpdateWorkDiary(int userId, int id, WorkDiaryDto newWorkDiaryDto)
      {
         if (userId != int.Parse(User.FindFirst(ClaimTypes.NameIdentifier).Value))
         {return Unauthorized();}
         var user = await this.repo.GetUser(userId);
         if (!user.WorkDiaries.Any(wd =>wd.Id == id)) return Unauthorized();
         var diaryFromRepo = await this.repo.GetDiaryById(id);
         var oldDiary = user.WorkDiaries.Where(wd => wd.Id == id);
         this.mapper.Map(newWorkDiaryDto,  user.WorkDiaries.FirstOrDefault(wd => wd.Id == id));
         if (await this.repo.SaveAll())
            return NoContent();
         throw new System.Exception("Updating work diary failed on save");
}
```

Code Snippet 39. Implementation of add and update diary

6. TESTING

6.1. Registering as a New User

- Testing steps:

  Go to the application, click the “Switch to signup” button. Fill all the fields and click “SIGNUP”.

- Expected result:

  If the registration process is done successfully, the application will redirect the user onto the “contract-info” page. Actual result:

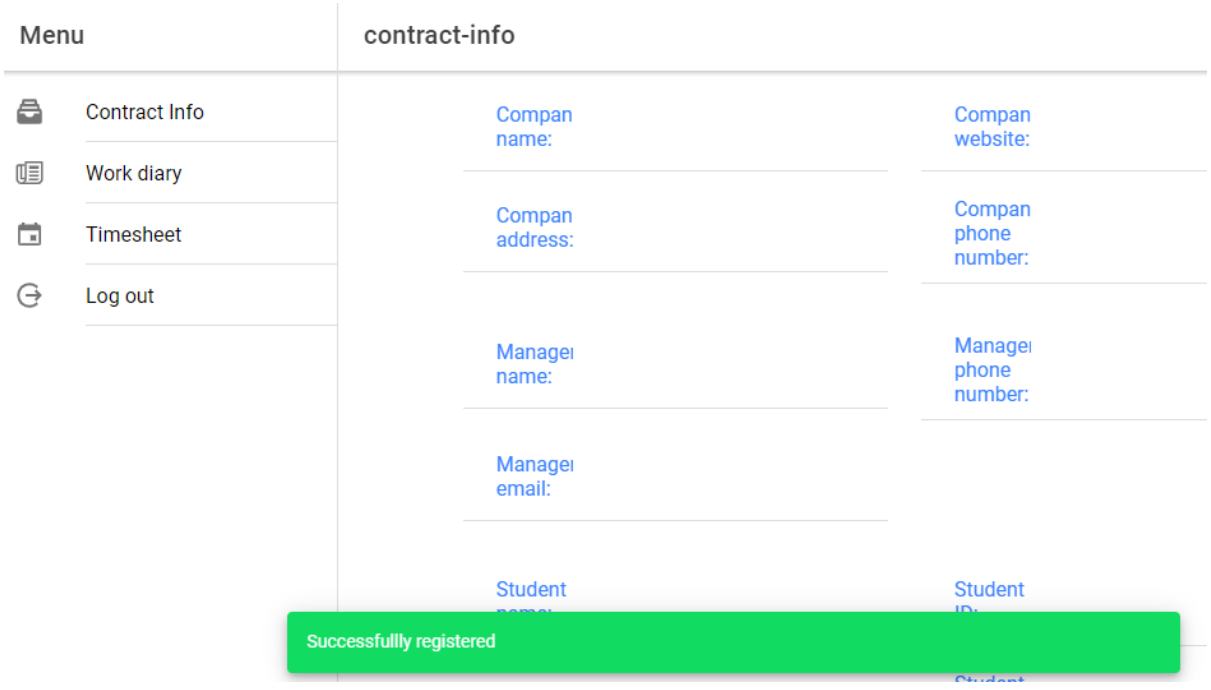

Figure 35. Successful registration

In case of an error occurring during the process, an error message will be shown.

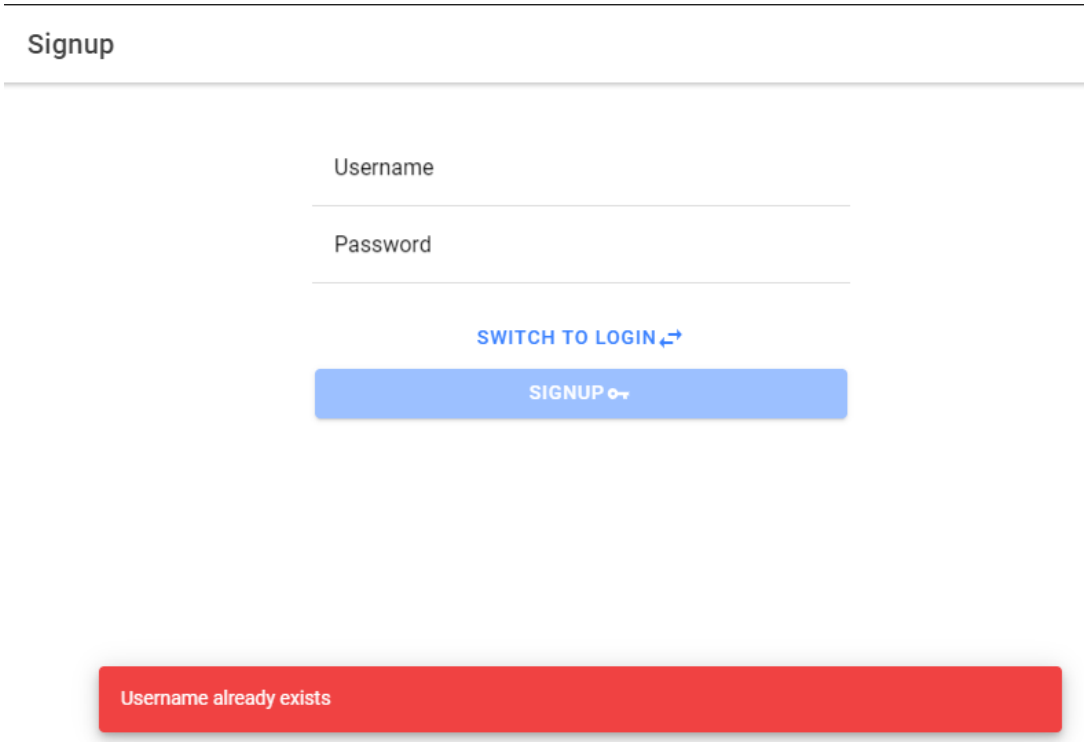

Figure 36. Error during registration with message

The message will include the reason why the error happened.

6.2. Editing contract information

- Testing steps:

  On “contract-info” page click “EDIT”, and fill the form provided.

  Click “SAVE”

- Expected result:

  Get redirected to the “contract-info” page with newly updated information

- Actual result:

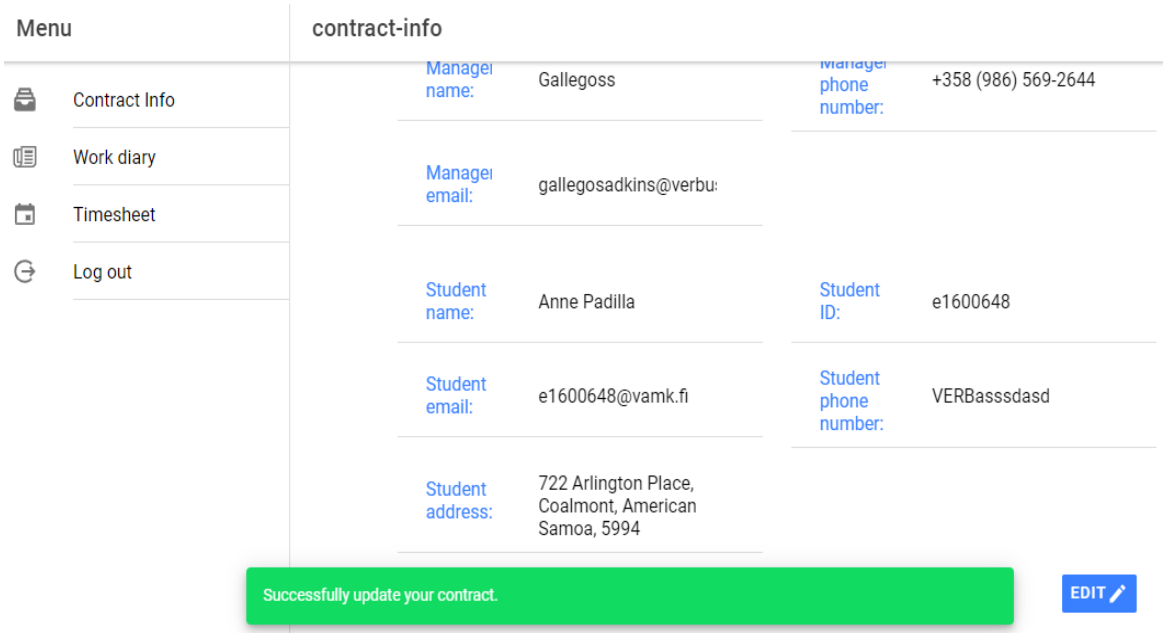

Figure 37. Successfully redirected with contract information updated

If the quest is completed without any error, a message will be shown to inform the user.

6.3. Adding new Work Diary

- Testing steps:

  Navigate to the “Work diary” page. Click the plus icon on the title bar. Once navigated to “work-diary-edit”, fill in the form and submit by pressing “SAVE”. A predefined template is available by using the “Templates” select box

  Click “SAVE”

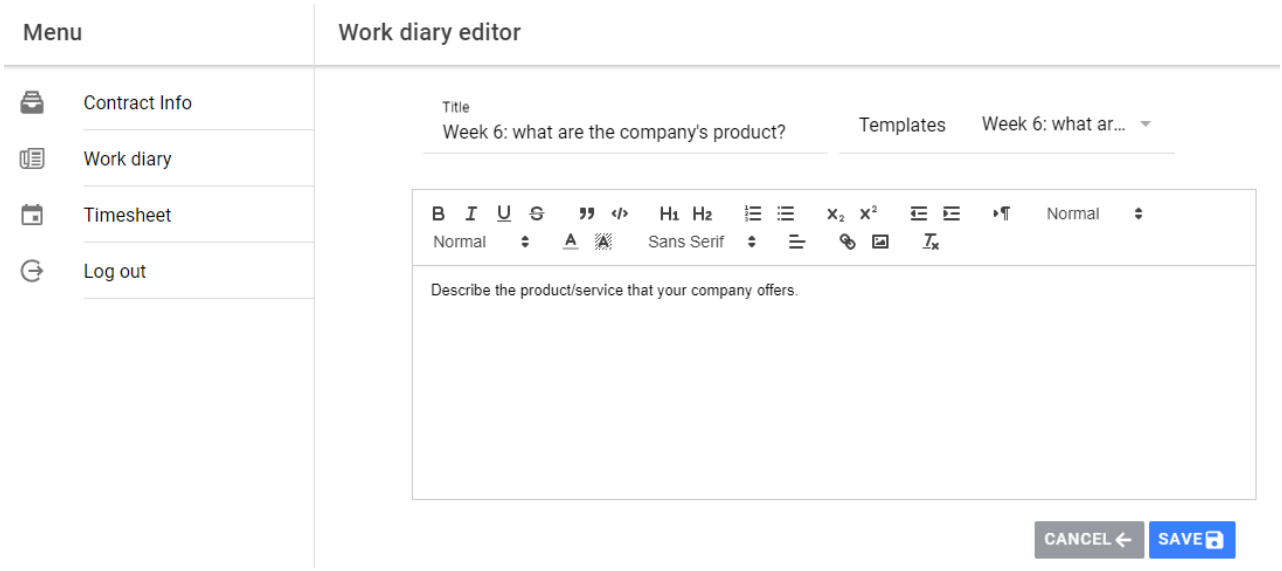

Figure 38. Filling in the form with predefined template

- Expected result:

  Upon saving, the page will automatically navigate to the "WorkDiary" page, and a new work diary card will be added.

- Actual result:

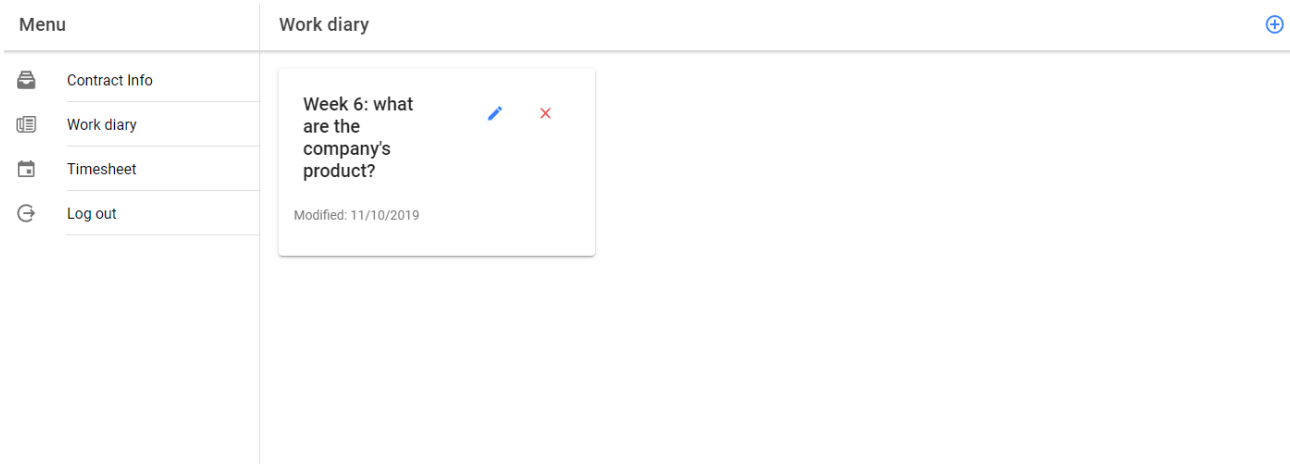

Figure 39. New work diary card added

A new diary card was added and can be modified or removed in the future.

6.4. Editing and Removing Existing Work Diary Card

- Testing steps:

  Press the pen icon to edit the card information. The page will navigate to "WorkDiaryEdit" loaded with previously registered information.

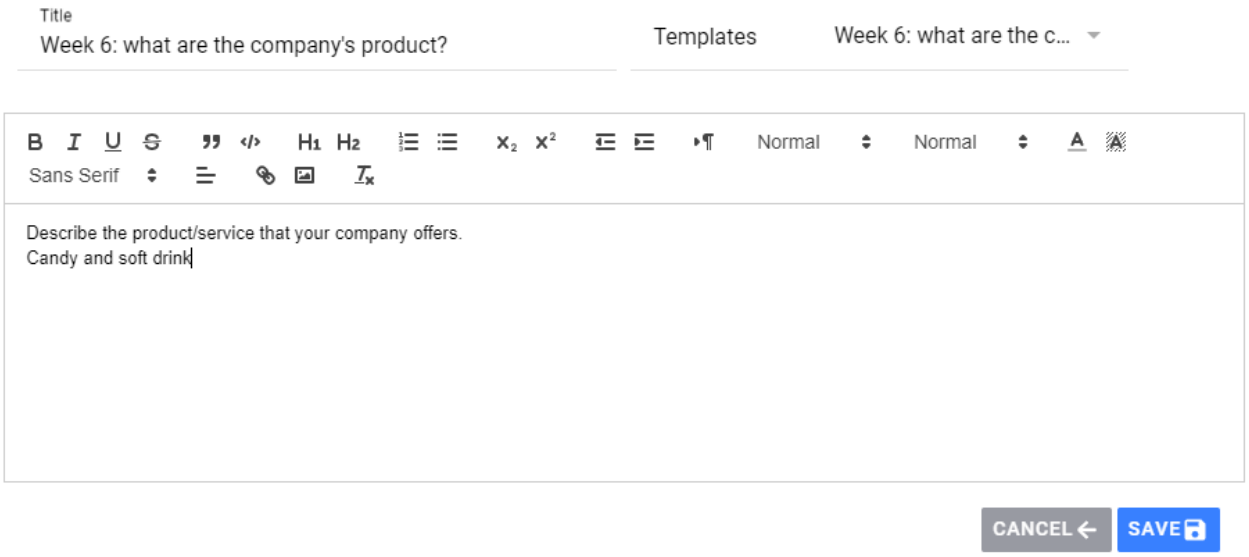

Figure 40. WorkDiaryEdit loaded with previously input data

Click "SAVE" to save and get directed to the "WorkDiary" page.

Press the X button to remove a "WorkDiary" card. Confirming with the popup by clicking "Confirm".

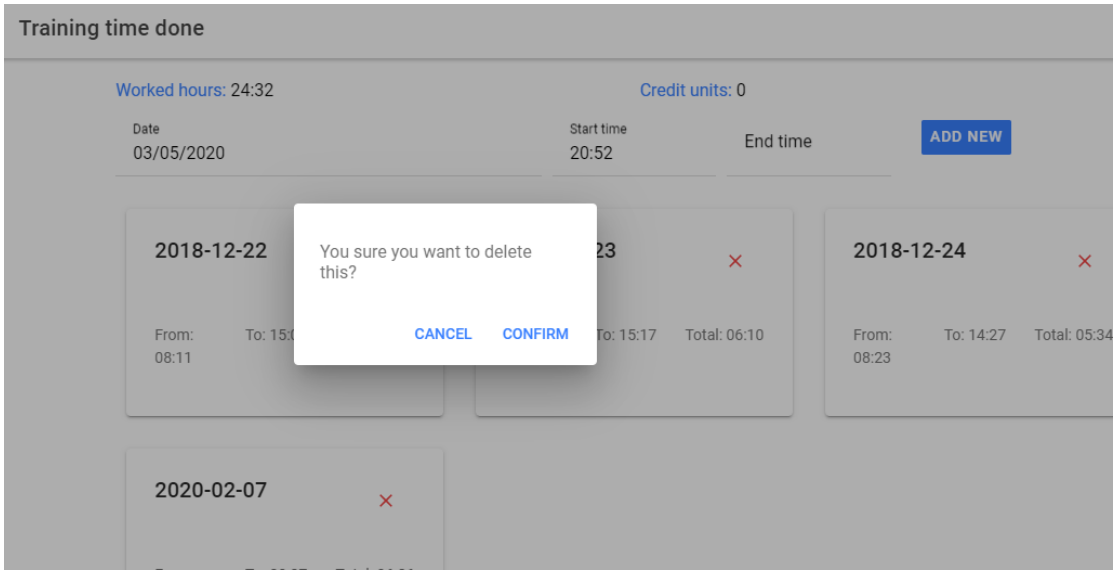

Figure 41. Removal confirmation

- Expected result:

  The card should be modified with new information after the edition.

  The removed card will be deleted from the page and the database.

  Upon saving, the page will automatically navigate to “WorkDiary” page, and a new work diary card will be added.

- Actual result:

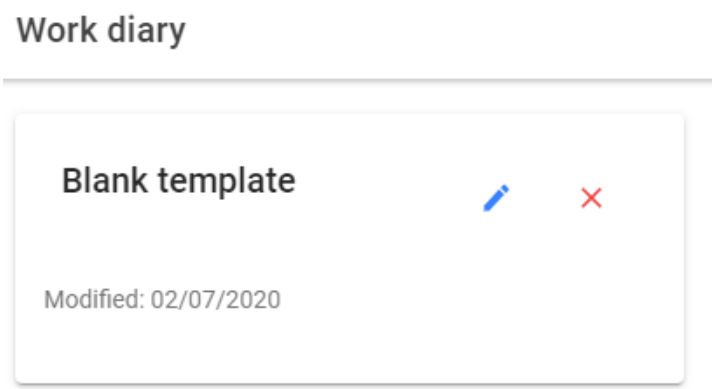

Figure 42. Modified work diary card

The modified card will also log the date on which the card was modified.

6.5. Adding New Work Session:

- Testing steps:

  Click the Timesheet icon from the side panel and navigate to “TimeSheet” page.

  Select the date with starting and ending time as new work session.

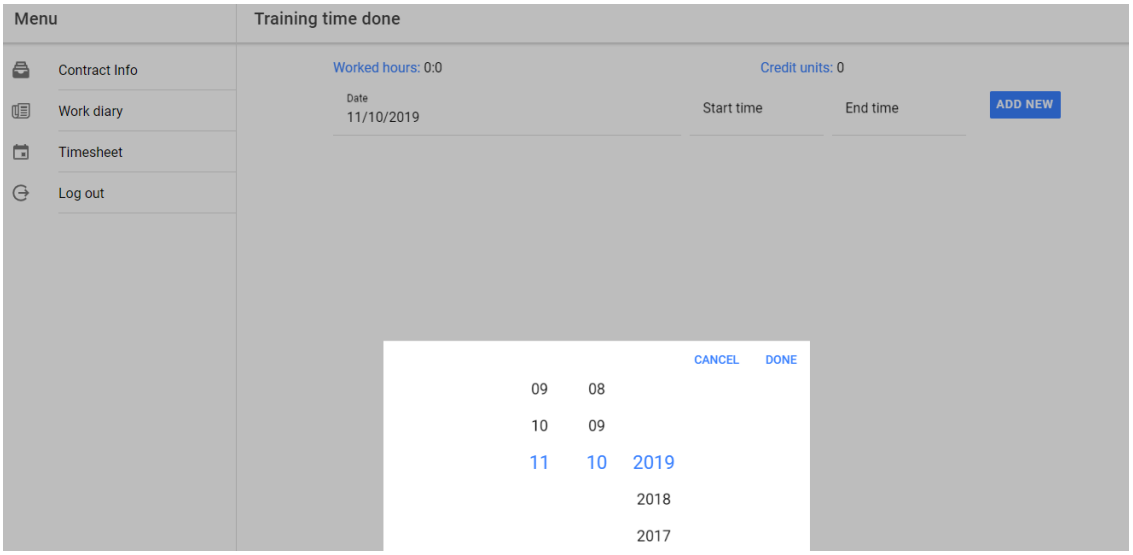

Figure 43. Selecting the date

Click “ADD NEW”

- Expected result:

  A new card will be added with appropriate data. The application should also calculate the total work hours and credit units.

  Thirty minutes will be automatically reduced from work sessions spanning more than six hours.

- Actual result:

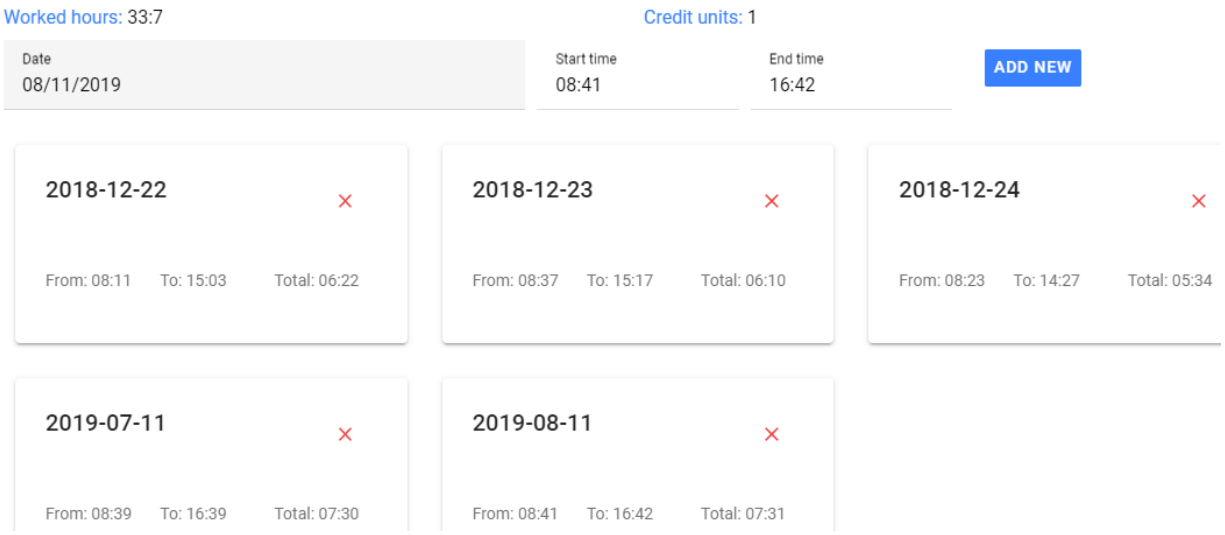

Figure 44. New work session with total hours and credit unit calculated

6.6. Deleting Work Sessions.

- Testing steps:

  Click the X button on the work session card.

  Click confirm from the popup

- Expected result:

  The work session should be removed.

  Total work hours and credit unit should be updated automatically.

- Actual result:

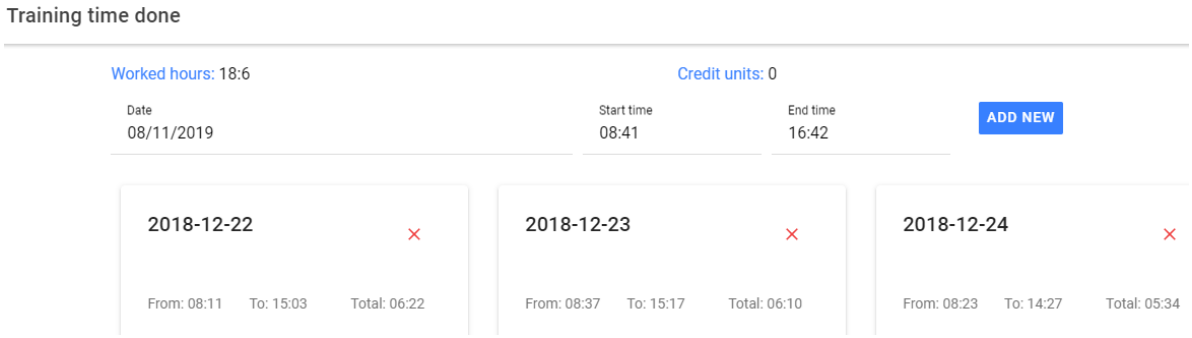

Figure 45. Removed work session with all information re-calculated

7. CONCLUSIONS

The project has developed an application which can be used on desktop using conventional internet browser, while also being able to work as a native Android application. Aiming to help student writing a more comprehensive practical training report, this application can help students registering important experience during their training period. It allows students to log in the details of their practical training periods, calculating the number of credit units that the working period is worth. The frontend application and the backend application are developed individually, so they can both be used and further developed independently.

During the developing process, some of the more prominent challenges were finding the correct framework that fit the requirements of the project to have an Android application along with a web application, technical difficulties that came from working with the technologies, such as working with JWT token, error that came from version conflict between ngx-quill and Ionic, ensuring data is available for the client application before a route is loaded, and difficulties that come from working with the difference between Ionic and Angular.

As with many applications, the project can be further improved upon.

Some suggestions for future development if the application include exporting the timesheet into other popular file format, such as PDF, xlsx, docx. When connected to the school system, student information can be automated to have all information regarding the student filled in the application database beforehand. The work diary can also be condensed into a single rich text document that can be submitted to the responsible supervisor of the student as the practical training report to be graded.

REFERENCES

/1/ Angular (web framework). Accessed 07.11.2019.
https://en.wikipedia.org/wiki/Angular_(web_framework)

/2/ Angular Architecture. Accessed 07.11.2019.
https://angular.io/guide/architecture

/3/ Ionic (mobile app framework). Accessed 07.11.2019.
https://en.wikipedia.org/wiki/Ionic_(mobile_app_framework)

/4/ Angular. The complete guide by Maximilian Schwarzmüller. Accessed 07.11.2019.
https://www.udemy.com/course/the-complete-guide-to-angular-2/

/5/ Npm (software). Accessed 07.11.2019.
https://en.wikipedia.org/wiki/Npm_(software)

/6/ Quill. Accessed 07.11.2019.
https://quilljs.com/

/7/ ngx-quill. Accessed 07.11.2019.
https://www.npmjs.com/package/ngx-quill

/8/ date-fns. Accessed 07.11.2019.
https://date-fns.org/

/9/ Entity Framework. Accessed 07.11.2019.
https://en.wikipedia.org/wiki/Entity_Framework

/10/, /11/ JSON Web Tokens. Accessed 07.11.2019.
https://jwt.io /