Universitetet
i Stavanger

Faculty of Science and Technology

BACHELOR'S THESIS

Study program/Specialization: Bachelor of Science in Computer Science	Spring semester, 2022 Open / Restricted access
Authors: Benjamin Chandler, Eirik Danielsen	
Faculty supervisor: Nejm Saadallah External supervisor: Eyasu Gebremichael	
Thesis title: Graphical representation of fluid data	
Credits (ECTS): 20	
Keywords: Azure, iOS, Android, Python, React, React Native, Rheology, Expo	Pages: 90 + attachments/other: 13 Stavanger 15. mai 2022

Contents

Content **i**

Summary **vii**

Terminology **vii**

1 Introduction **1**

1.1 Project Description 1

1.2 Objective 1

1.3 Outline 2

2 Background **3**

2.1 Drilling fluids 3

2.2 Industry use of rheology data 4

2.3 Rheology 4

2.4 Intelligent Mud Solutions AS 5

2.5 Defining the project 5

3 Development Tools & libraries 8

3.1 Development methodology & choice of platforms 8

3.1.1 Development methodology & Git 8

3.1.2 Browser, iOS & Android 9

3.2 React 10

3.3 Expo & React Native 13

3.3.1 React Native & Cross-platform 13

3.3.2 Expo 14

3.4 Python 15

3.4.1 Flask 16

3.4.2 Pandas 16

3.4.3 Matplotlib 17

3.5 Microsoft Azure 18

3.5.1 Azure SQL Database 18

3.5.2 Logic App 18

3.5.3 App Service 19

4 Backend Development 20

4.1 Setting up a development environment 20

4.2 Simulating The Rheosense 21
4.2.1 Cleaning the data 22
4.3 Building the database 25
4.3.1 Creating the database schema 25
4.3.2 Connecting Python with the database 26
4.3.3 Inserting data 26
4.3.4 Querying data 27
4.3.5 Stats query 28
4.4 Creating The APIs 29
4.4.1 Configuring the app 30
4.4.2 Establishing a connection with the database 32
4.4.3 Closing the connection to the database 34
4.4.4 Creating the first API 34
4.4.5 Get and GetInterval API 37
4.4.6 GetStats API 38
4.4.7 Securing the APIs 38
4.5 Login and registration 39
4.6 Deployment to Azure 41
4.6.1 Creating the App Service on Azure 41
4.6.2 deploying the Flask app to Azure 41

4.7 Automated Reporting Feature 42
4.7.1 Collecting data 43
4.7.2 Creating the plot 43
4.7.3 Sending emails to users 46
4.7.4 Azure Logic Apps 49

5 Web Development 51

5.1 Structure 51
5.1.1 Routing 51
5.1.2 Rendering 53
5.1.3 State and props 53
5.1.4 Hooks 54
5.2 Charting 56
5.2.1 Chart component 57
5.3 Stats 61
5.4 Obtaining data 61
5.4.1 Fetching data 62
5.5 Creating Pages 65

6 Mobile Development 68

6.1 Structure 68

6.1.1 Custom styled components 70

6.2 Login screen 72

6.2.1 Creating the form 73

6.2.2 Sending a login request 77

6.2.3 Sign in via Google 79

6.3 Registration screen 81

6.4 Home screen 82

6.5 Routing 84

7 Results, Conclusion and future development **86**

7.1 Results 86

7.2 Discussion and further work 87

7.2.1 Web application 87

7.2.2 Mobile application 88

7.3 Conclusion 89

Bibliography **93**

Vedlegg **93**

A Programlisting **94**

A.1 LiveData.py 94

A.2 Setup$_d$*b.py* 97

A.3 Helper Functions 98

A.4 Stats component 100

A.5 Page components 102

Abstract

In this Bachelor thesis we will work together with Intelligent Mud Solutions to explore the Graphical representation of fluid data. We will give an introduction about IMS and Rheology. We create a development environment mimicking the existing database and provide it with data from a python script utilizing real data.
We use React and React-Native to create live and historical charts in the format used by mud engineers.
To achieve this we create an API using Flask.py, this API is then used to automate the creation and sending of PDF reports by e-mail.

Chapter 1

Introduction

1.1 Project Description

In this bachelor thesis we will explore how to present live and historical data. The bachelor thesis came about trough a partnership with Intelligent Mud Solutions, who will be referred to as IMS.
IMS is a Technology firm based out of Stavanger who focuses on digitalizing the drilling industry. One of their products is a sensor rig to provide rheology data from drilling wells.It is this data that will need to be presented in a satisfactory manner.

IMS goal with this data is to present it in real time to their customers trough a web application and possibly trough a mobile app. In addition they would also like reporting of historical data both on the web and mobile application and as an email PDF report.

1.2 Objective

Our overarching objective for this bachelor thesis is therefore to explore the real-time and historical graphical presentation of fluid data. This is a very broad objective that can be solved in a multitude of ways, thus we will first need to examine background information about both IMS and rheology. This will be the subject of chapter 2 and will culminate in a detailed list of objectives that will be the goal of our bachelor thesis.

1.3 Outline

The thesis is structured as follows:

- **Chapter 1** Introduction of the project, a short introduction of the main objectives and motivation for the project.
- **Chapter 2** introduces the reader to the company and the necessary petroleum background needed to make sense of the data.
- **Chapter 3** specifies the technologies and development tools used for this project.
- **Chapter 4** covers the backend development process.
- **Chapter 5** details the web app development process.
- **Chapter 6** describes mobile app development process
- **Chapter 7** concludes the results and further discussion.

Chapter 2

Background

2.1 Drilling fluids

Drilling Fluid or drilling mud as it is often referred to is a fluid used in the drilling of wells. The simplest form of drilling fluid is simply water. Different forms of chemicals can be added to the mixture to give it certain properties. The second type of drilling fluids are oil based instead of water based.
Drilling fluid is pumped trough the well while its being drilled. In the well it provides several benefits to the drilling operation[1]. As it flows trough the well it removes drill cuttings, where a high viscosity will allow cuttings to be brought up easier.
While the mud is the well it also make contact with the drill and the borehole itself, lubricating the wellbore and cooling down the drill bit. This allows for more force being exerted to the drill bit while increasing lifespan of the bit and wellbore, reducing maintenance costs.
The presence of drilling fluid also increases the buoyancy. Making the drill string weigh less and increasing the stability of the well.
Some drilling tools even use the mud itself to transmit signals to the surface.
The composition of the drilling fluid dictates the effect it has on the well.
Changes can be made to the drilling fluid by adding or subtracting chemicals from the mixture. This allows the properties of the fluid to be changed as one drills further down into the well.

2.2 Industry use of rheology data

As mentioned above there are many uses of drilling fluids. Therefore it is important to have up to date and correct information about the drilling fluids property.
Mud engineers monitor the drilling fluids properties to ensure that it will have the desired effect in the well.
Historically these properties have been monitored by historical reports about the previous shift, with new reports being available every 12 or 24 hour periods. The drilling engineer can then make necessary changes if needed.
To measure the properties of drilling mud one uses rheology data.

2.3 Rheology

Rheology can be defined as the science of flow and deformation of matter [2]. An important topic in rheology when analyzing drilling fluids is viscosity as the viscosity of the mud greatly impacts its effect in the well.
Viscosity is in rheology defined as the ratio between shear stress and shear rate[2].Shear stress is the force required to keep a constant flow of a fluid [3]. Shear rate is the measurement of the rate of change in velocity between fluid layers [4].
In a Newtonian fluid the viscosity is independent from the shear rate, unlike non-newtonian fluids in which the viscosity is a function of the shear rate.

Bingham's plastic model is a rheological model used in the oil and gas industry to describe drilling mud, in which plastic viscosity and yield point is derived from shear stress and shear rate.
Plastic viscosity is a measurement of the muds viscosity under infinite shear rate, a low plastic viscosity indicates that one can maintain a high drill speed [5].
Yield point is a measurement derived from the viscosity when the sheer rate is set to zero[6].The yield point can be used to determine if a fluid is Non-Newtonian. A Non-Newtonian fluid is a fluid that changes behaviour when force is applied to it. This is important as non-Newtonian fluids can carry drill cuttings better than a Newtonian fluid even if the two fluid have

the same density.
The Yield point and Plastic viscosity can both be derived by using viscosity measurements made with different rotations per minute.

2.4 Intelligent Mud Solutions AS

Intelligent Mud Solutions (IMS), is a technology firm based in Stavanger. Since their formation in 2010 they have developed tools for fluid analysis and management. Tools that see use in the oil an gas section, in particular well drilling. Their products are a part of the digitalization of the oil and gas sector, and they are currently furthering the research and development of these tools. The relevant tools for this thesis are;

- RheoSense, Onsite analysis tool
- RheoMax, Onsite analysis tool with additional sensors
- RheoReport, report tool for historic data
- RheoView, live view of analysis data

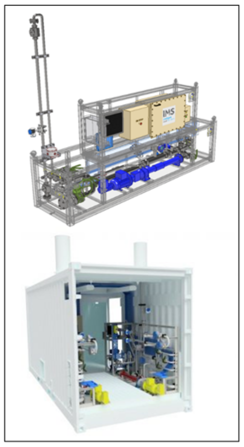


Figure 2.1: RheoSense unit.

In particular the Rheosense as it is the unit gathering the data we are visualizing.
The data points gathered from the Rheosense is currently;
viscosity measurements from the following rotations per minute; 3, 6, 100, 200, 300 and 600.
Yield point, Plastic viscosity, density, PH and the inlet temperature.

2.5 Defining the project

To define the scope of our thesis project we will look at the following 2 main factors, what technology is cur-

rently in place and what is required.
The current technology in use is a Microsoft SQL-server Database hosted on azure cloud, with Power BI being used for presentation. It is the presentation of data for RheoReport and RheoView that IMS wants to improve, to present it in a manner better suited for mud engineers.
IMS currently have a good solution for pushing data from their RheoSense unit and hosting in the cloud but want to improve their graphical presentation and ability to retrieve data from the database.

The solution made in this project is not expected to be ready for customers immediately, but rather a presentation module that can be implemented into their RheoView and RheoReport tools by a third party.
IMS wants a working prototype for testing and customer demos.

This gives us our main objectives, which are the following;

- Build a modular view for presenting live and historical data.
- Create a web application for demo and testing purposed, using the presentation module.
- Build a tool for automating the creation and sending of email PDF reports.
- Explore the creation of a mobile app that presents live and historical data.

Too achieve this we also need to complete several sub objectives, which include the following;

- Simulate a RheoSense to provide data.
- Determine how to best present data in a manner that corresponds with how the data will be used.
- Create a development environment that resembles the production environment already in use.
- Create an API to enable the presentation modules to talk with the database

- Create a Log in and authorization system to be used for the demo application

Chapter 3

Development Tools & libraries

3.1 Development methodology & choice of platforms

3.1.1 Development methodology & Git

Due to previous experience our version control system of choice was git. Git is a free and open source version control system that enables us to cooperate and manage the projects workflow with ease. One of git's many features we utilized was the branching model. The branching model lets you have multiple local branches of your project where creation, deletion or merging of said branches is a quick and simple process.

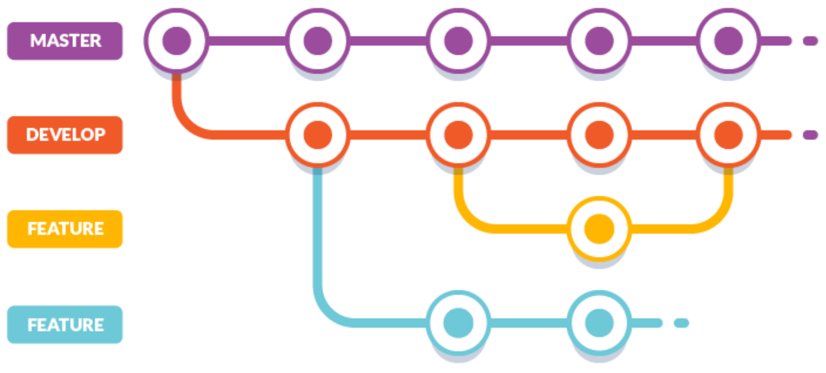


Figure 3.1: Visualization of git branches [7]

The power behind the utility of the branching model is the separation of project. This makes it easy to add new feature, bug fixes or experiment without interfering with your own or other developers' progress. This proved to be useful as we followed scrum methodology for development where weekly sprints and backlogging was made easier using git.

3.1.2 Browser, iOS & Android

In this project we are tasked with building both a web application and a mobile application. The web application should be able to support all modern browser such as google chrome, Firefox, safari, Microsoft Edge etc. The mobile application should be available to nearly all users. To achieve this goal we have to develop the mobile application for both android and iOS as those are the biggest mobile platforms worldwide. Figure 3.2 illustrates the market shares of mobile platforms in Norway over the last 12 months.

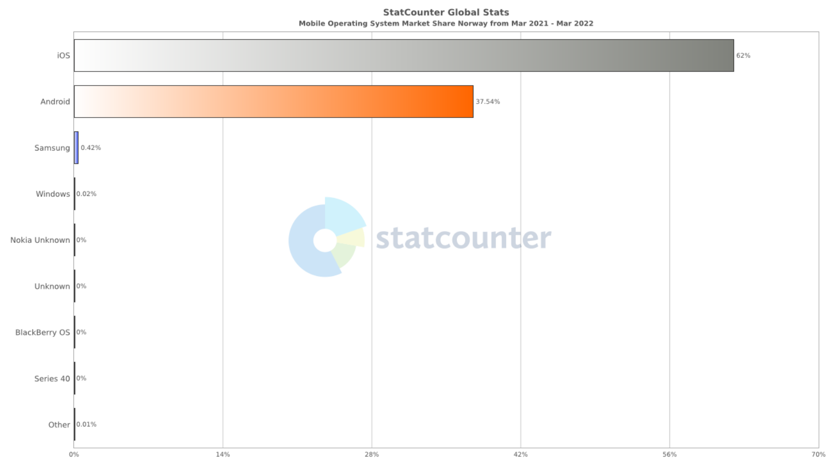


Figure 3.2: Mobile market share Norway [8]

In Norway iOS is more popular than android, but worldwide Android is more common than iOS. Leaving one of those two platforms out will greatly reduces the number of users able to use your application.

3.2 React

React is a free and open source JavaScript front-end library for building user interfaces. React was created by Meta (formerly known as Facebook) and launched in in 2013. Since its launch it has grown in popularity and is today the most used web frameworks among developers worldwide as illustrated in figure 3.3

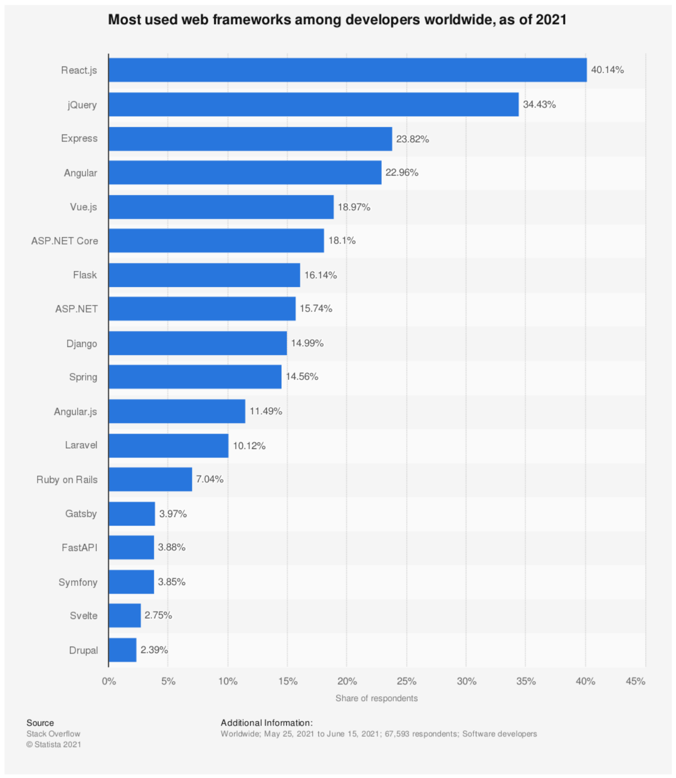


Figure 3.3: Most used web frameworks for developers [9]

React utilizes JSX (JavaScript XML) which is a JavaScript syntax extension in favour of regular HTML. This allows developers to combine markup language and logic into something called components as opposed to having to separate files for markup and logic. Although React does not require JSX being used it is highly recommended as it makes it easier to write or add HTML.

React has a component architecture. A React component is a independent and a reusable bit code that is responsible for a part of the user interface. Components have an internal state and returns JSX. There are two types of React components available, class components and functional components. Ever since React introduced hooks which gave functional components access to states and all of the features exclusive to class components. There are no real differences between the two except from syntax where functional components are more straightforward and concise.

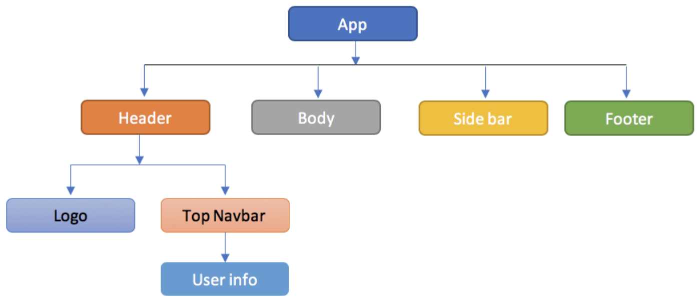


Figure 3.4: React component architecture [10]

Figure 3.4 illustrates how a user interface might be broken down into different components. Components are denoted by first letter uppercase while regular JSX uses lowercase. React components might look something like the example given below.

```
function Header() {
    return <h2>This is the header component!</h2>;
}
```

```
function Footer() {
    return <h2>This is the footer component!</h2>;
}
```

Here we have two separate and independent React components, each responsible for displaying different parts of the user interface. All that needs to be done to combine these components together to display the full UI is the following.

```
import Header from './components/Header';
import Footer from './components/Footer';

function App() {
    return (
        <Header />
        <Footer />
    );
}
```

3.3 Expo & React Native

3.3.1 React Native & Cross-platform

React Native is a cross-platform JavaScript framework developed by Meta and was open-sourced in 2015. The framework was built based on React, a JavaScript library developed earlier by Meta that was highly popular at the time of React Native's release. Ever since its release react native has grown in popularity and is behind the development of many world leading mobile apps such as Facebook, Instagram, Skype, Tesla and Discord. Having a background in JavaScript makes it easy using react native to start developing mobile apps for iOS and android [11].

The power behind the mobile framework is that it allows developers to build apps that supports various platforms from just a single codebase. The benefits of utilizing a cross-platform framework is the ability to target a large audience, cut down cost and time to market. A company developing a cross-platform app may now only hire one team instead of hiring an android developer team using Java or Kotlin and a separate iOS developer team using Swift or objective C.

A drawback to using a cross-platform framework is that it requires expertise in more than one platform and makes code design harder as it needs to be responsive to various devices and platforms. A native developer only needs to concern themselves with the ins and outs of their platform, whereas a cross-platform developer needs to worry about how a change might impact other platforms as well [12].

React Native enables hot reloading which results in fast iteration speed. Hot reloading allows you to save changes while running your project and the changes will appear immediately as opposed to having to save, run, compile the project and then navigating to the page in question to see your changes [11]. There are a lot of similarities between React and React Native such as JSX, components, props and state. There are some differences however. React Native uses native components as building blocks whereas React uses web components. Examples of standard JSX components in React are `<Div></Div>` or `<P></P>` which equates to `<View></View>` or `<Text></Text>` in React Native. Here the view component renders a container and the text component allows us to render text. You are also able to import hooks exactly like what you would have done using React.

3.3.2 Expo

When creating a React Native app there are two ways you can go. You can get started using the Expo command line interface (CLI) or the React Native CLI. Expo is a framework used to build React Native apps. It is a set of tools and services built around React Native that helps you develop, build, deploy and quickly iterate on IOS and Android from the same JavaScript codebase. Expo also provides hot reloading, debugging and logging features to simplify the development process [13].

Expo lets developers create iOS apps without the need for a computer running on macOS which lowers the barrier to entry and allowing windows and Linux users to also develop apps for iOS. Expo also offers over the air updates which means you don't have to wait for apple store or google play to review your updates each time you push new code, which can take weeks [13].

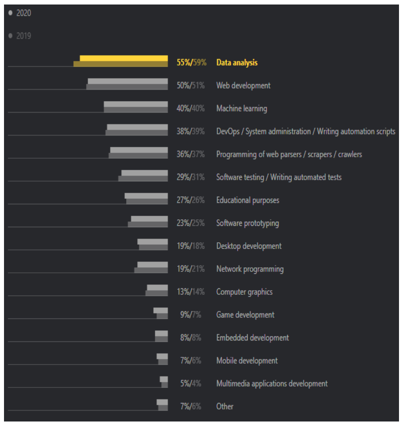
Although Expo is great for getting started quickly with cross-platform development it does have its limitations. Not all iOS and Android APIs are available like Bluetooth or Geolocation. In the Expo documentation there are lists of all the available native APIs and those available are satisfactory for the scope of this project [13]. Expo is supported by the the React Native team and is even included in the React Native documentation [11].

When developing an app using Expo you can view your project either through a physical device (Android or iOS) via their Expo Go app or through an emulator. The group had access to an iOS device and an Android device. This helps the mobile cross-platform development process as you are able to view the application through different platforms and keep up to date whether the app still has that native feeling on their respective platforms.

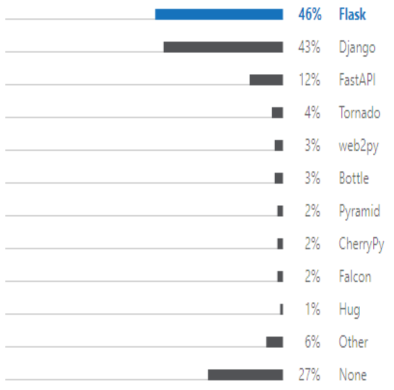
3.4 Python

Python is a high-level, general purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation [14]. Python is relatively simple to get started with due to easy to read syntax. Python is a popular choice when it comes to developing web applications as illustrated in a survey performed by Jetbrains[15]. In figure 3.5a, 50% of the 28.000 python developers surveyed responded that their main use of the programming language was for the intention of web development.

Python has a mature and supportive community as its been around for over 30 years and there are thus a ton of python frameworks and libraries to work with. This section will cover the three main libraries/frameworks included in this project.



(a) What do you use python for [15]



(b) Most used python web frameworks [15]

Figure 3.5: Caption for this figure with two images

When developing web applications with python there are a lot of frameworks to choose from. In the survey[15] mentioned previously the respondents were also asked which web framework they preferred when using python. The two most popular python web frameworks according to this survey ended up being Flask and Django as shown in figure 3.5b. For this project Flask ended up as the web framework of choice due to previous experience in courses like DAT250 and DAT310. The supervisor from IMS also had some limited experience using Flask which solidified this choice.

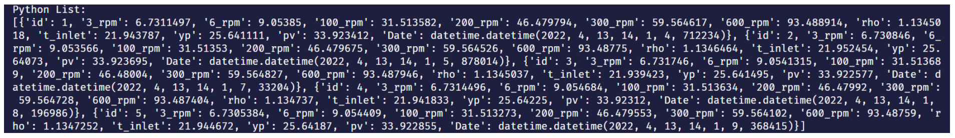
3.4.1 Flask

Flask is a python micro web framework released 12 years ago in April 2010. it is not classified as a framework because it does not require any tools or libraries. Flask does not have any database abstraction layer or any other components that third-party libraries already provides. In the documentation Flask claims that "Flask aims to keep the core simple but extensible."[16]. This way you only import what you need and nothing more.

3.4.2 Pandas

Pandas is a Python library designed for data manipulation and analysis. Pandas was first open sourced in 2009 and is widely used by data scientist/data analysts and also for machine learning purposes. Pandas introduces something called a DataFrame, their main and most commonly used data structure [17]. A DataFrame is a two-dimensional data structure that consists of columns and rows.

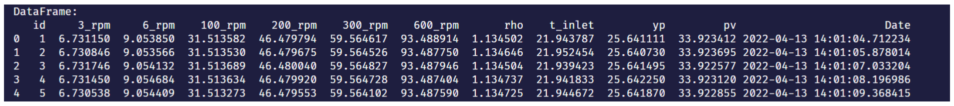
Figure 3.6 shows a standard Python list containing five dictionaries of various petroleum readings. Reading and working with data in such a format can be challenging and even impossible once the data set reaches a certain size. This is a situation where the application of the Pandas DataFrame would be useful.



```
Python List:
[{'id': 1, '3_rpm': 6.7311497, '6_rpm': 9.05385, '100_rpm': 31.513582, '200_rpm': 46.479794, '300_rpm': 59.564617, '600_rpm': 93.488914, 'rho': 1.13450
18, 't_inlet': 21.943787, 'yp': 25.641111, 'pv': 33.923412, 'Date': datetime.datetime(2022, 4, 13, 14, 1, 4, 712234)}, {'id': 2, '3_rpm': 6.730846, '6_
rpm': 9.053566, '100_rpm': 31.51353, '200_rpm': 46.479675, '300_rpm': 59.564526, '600_rpm': 93.48775, 'rho': 1.1346464, 't_inlet': 21.952454, 'yp': 25.
64073, 'pv': 33.923695, 'Date': datetime.datetime(2022, 4, 13, 14, 1, 5, 878014)}, {'id': 3, '3_rpm': 6.731746, '6_rpm': 9.0541315, '100_rpm': 31.51368
9, '200_rpm': 46.48004, '300_rpm': 59.564827, '600_rpm': 93.487946, 'rho': 1.1345037, 't_inlet': 21.939423, 'yp': 25.641495, 'pv': 33.922577, 'Date': d
atetime.datetime(2022, 4, 13, 14, 1, 7, 33204)}, {'id': 4, '3_rpm': 6.7314496, '6_rpm': 9.054684, '100_rpm': 31.513634, '200_rpm': 46.47992, '300_rpm':
 59.564728, '600_rpm': 93.487404, 'rho': 1.134737, 't_inlet': 21.941833, 'yp': 25.64225, 'pv': 33.92312, 'Date': datetime.datetime(2022, 4, 13, 14, 1,
8, 196986)}, {'id': 5, '3_rpm': 6.7305384, '6_rpm': 9.054409, '100_rpm': 31.513273, '200_rpm': 46.479553, '300_rpm': 59.564102, '600_rpm': 93.48759, 'r
ho': 1.1347252, 't_inlet': 21.944672, 'yp': 25.64187, 'pv': 33.922855, 'Date': datetime.datetime(2022, 4, 13, 14, 1, 9, 368415)}]
```

Figure 3.6: Raw Data

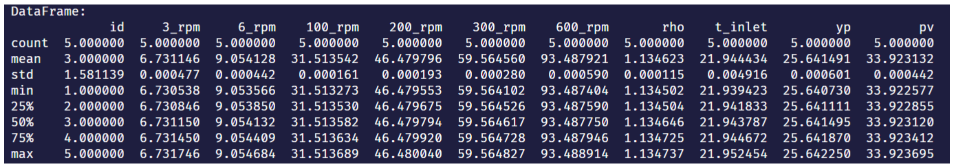
Figure 3.7 displays the same data as the previous figure, but instead using a DataFrame object to display the data. The power behind the DataFrame object is not only due to the improved visualization. It also opens up for a lot of methods to be called on the object that are useful for data analysis like percentiles, min, max and many more.



```
DataFrame:
   id     3_rpm     6_rpm    100_rpm    200_rpm    300_rpm    600_rpm       rho    t_inlet         yp         pv                       Date
0   1  6.731150  9.053850  31.513582  46.479794  59.564617  93.488914  1.134502  21.943787  25.641111  33.923412 2022-04-13 14:01:04.712234
1   2  6.730846  9.053566  31.513530  46.479675  59.564526  93.487750  1.134646  21.952454  25.640730  33.923695 2022-04-13 14:01:05.878014
2   3  6.731746  9.054132  31.513689  46.480040  59.564827  93.487946  1.134504  21.939423  25.641495  33.922577 2022-04-13 14:01:07.033204
3   4  6.731450  9.054684  31.513634  46.479920  59.564728  93.487404  1.134737  21.941833  25.642250  33.923120 2022-04-13 14:01:08.196986
4   5  6.730538  9.054409  31.513273  46.479553  59.564102  93.487590  1.134725  21.944672  25.641870  33.922855 2022-04-13 14:01:09.368415
```

Figure 3.7: Same data converted into a Pandas DataFrame object

Although collecting such statistical data is not impossible to do without Pandas, it does make it a lot easier and less time consuming to do so. Figure 3.8 shows how you can retrieve key statistical data points using the previous data set with the `describe()` method.



```
DataFrame:
             id     3_rpm     6_rpm    100_rpm    200_rpm    300_rpm    600_rpm       rho    t_inlet         yp         pv
count  5.000000  5.000000  5.000000   5.000000   5.000000   5.000000   5.000000  5.000000   5.000000   5.000000   5.000000
mean   3.000000  6.731146  9.054128  31.513542  46.479796  59.564560  93.487921  1.134623  21.944434  25.641491  33.923132
std    1.581139  0.000477  0.000442   0.000161   0.000193   0.000280   0.000590  0.000115   0.004916   0.000601   0.000442
min    1.000000  6.730538  9.053566  31.513273  46.479553  59.564102  93.487404  1.134502  21.939423  25.640730  33.922577
25%    2.000000  6.730846  9.053850  31.513530  46.479675  59.564526  93.487590  1.134504  21.941833  25.641111  33.922855
50%    3.000000  6.731150  9.054132  31.513582  46.479794  59.564617  93.487750  1.134646  21.943787  25.641495  33.923120
75%    4.000000  6.731450  9.054409  31.513634  46.479920  59.564728  93.487946  1.134725  21.944672  25.641870  33.923412
max    5.000000  6.731746  9.054684  31.513689  46.480040  59.564827  93.488914  1.134737  21.952454  25.642250  33.923695
```

Figure 3.8: The `describe()` method being called upon the previous DataFrame object

The improved data sorting capabilities Pandas provides is also useful in combination with other Python libraries, one of which will be covered in the following subsection.

3.4.3 Matplotlib

Matplotlib is a comprehensive Python data visualization library that has been around for 19 years [18]. Most of Matplotlib's utilities such as creating various charts fall under the pyplot sub module. Pyplot enables developers to create and customize their plots as they see fit. Supporting a ton of different methods to do so [18]. This library was therefore included as the necessary charts for this project required heavy customization. Matplotlib works great with the Pandas library and is even included in their own documentation [17].

3.5 Microsoft Azure

Microsoft Azure commonly knows as just Azure, is Microsoft's cloud computing platform. As cloud services has become more prevalent in the industry over the years there are now a multitude of technology companies that offer such services. Major tech companies like Google and Amazon, each offer their own cloud platform. Azure however, does have data a center located in Oslo, unlike Google or Amazon [19]. This might in part explain Azure's prominence in the Norwegian industry compared to its competitors. As stated previously, IMS have chosen Microsoft Azure as their cloud provider and for this project we will do the same. This section will introduce and give you a basic understanding of the Azure services used in this project.

3.5.1 Azure SQL Database

Azure offers numerous different ways to store your data like storage and database services. We reached out to the supervisor at IMS to know more about the how they choose to store their data on the cloud. This information proved to be useful when choosing which service would best fit the needs of this project. The group ended up selecting the Azure SQL Database since all of the group members had previous experience with SQL and because of the information gathered from IMS.

3.5.2 Logic App

For the projects daily reporting system we were in need of a service that would be able to periodically run a function. This is where Azures Logic Apps come in handy. The Logic Apps resource lets you create and run automated workflows. In Azure, a workflow is a process were there is an event followed by one or more actions [20]. Azure opens up for a lot of options and customization for what these events and actions may be.

3.5.3 App Service

The Azure App service resource lets developers build and host web apps, mobile back ends and APIs. App Service support most programming languages like Node.js, PHP, Python and many more [21]. The documentation for the App Service is extensive and getting started is fairly simple. This resource allows for deployment via Git through VS Code. Once your backend/app is deployed to the App Service you gain access to a variety of features and data, like the amount of requests made to the application.

Chapter 4

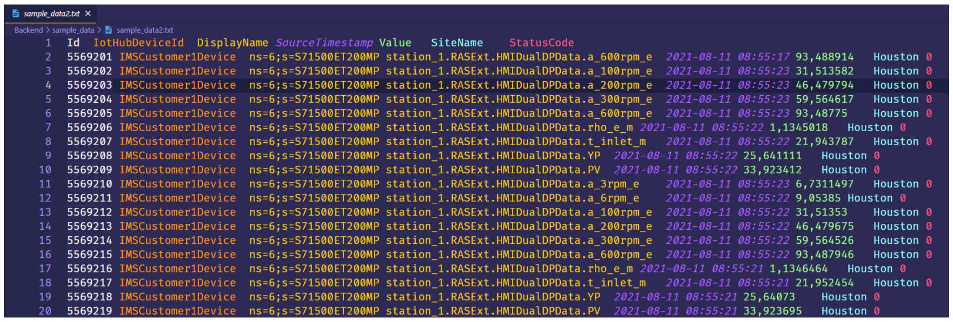
Backend Development

4.1 Setting up a development environment

The first thing we did when starting to work on the project was setting up a development environment. We did not have access to the clients production database nor was there an API to access the test data in the database. When creating the graphs we could have worked with hard coded test data, but seeing as IMS would also like a live demo site we would eventually have to get data from the database anyways.
By far the most common way to retrieve data from a database is by using an API as discussed in chapter 3.
Our plan for creating an API is to first create a database that we populate with data that can be used to test our graphical presentation.
We decided that the best course of action was to populate the database with data that resemble real data from a Rheosense, as this would help us in understanding how the data would be perceived by the end user. Thus our first goal was to simulate a Rheosense.

4.2 Simulating The Rheosense

When simulating the Rheosense we do not intend to create a full simulation but rather a script that can populate our database in real time with data that is similar to real rheology data.
Our first instinct was to determine a range of possible values for each data point and a range for their delta, and then generate new values randomly. This approach however is both time consuming and complicated, and when we started looking at Rheosense data we quickly found a better solution.
IMS provided us with a set of sample data from one of their Rheosense tests. Looking at that data we saw that it was very stable and that each data point often only had a delta of a few decimal points, and only occasionally had any significant changes.
To replicate this would be more complicated than just creating two ranges and use them to create random data points, this would most likely create a data set with far to much noise.



```
Id  IotHubDeviceId  DisplayName SourceTimestamp Value   SiteName    StatusCode
5569201 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.a_600rpm_e  2021-08-11 08:55:17 93,488914   Houston 0
5569202 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.a_100rpm_e  2021-08-11 08:55:23 31,513582   Houston 0
5569203 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.a_200rpm_e  2021-08-11 08:55:23 46,479794   Houston 0
5569204 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.a_300rpm_e  2021-08-11 08:55:23 59,564617   Houston 0
5569205 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.a_600rpm_e  2021-08-11 08:55:23 93,48775    Houston 0
5569206 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.rho_e_m 2021-08-11 08:55:22 1,1345018   Houston 0
5569207 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.t_inlet_m   2021-08-11 08:55:22 21,943787   Houston 0
5569208 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.YP  2021-08-11 08:55:22 25,641111   Houston 0
5569209 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.PV  2021-08-11 08:55:22 33,923412   Houston 0
5569210 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.a_3rpm_e    2021-08-11 08:55:23 6,7311497   Houston 0
5569211 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.a_6rpm_e    2021-08-11 08:55:22 9,05385 Houston 0
5569212 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.a_100rpm_e  2021-08-11 08:55:22 31,51353    Houston 0
5569213 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.a_200rpm_e  2021-08-11 08:55:22 46,479675   Houston 0
5569214 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.a_300rpm_e  2021-08-11 08:55:22 59,564526   Houston 0
5569215 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.a_600rpm_e  2021-08-11 08:55:22 93,487946   Houston 0
5569216 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.rho_e_m 2021-08-11 08:55:21 1,1346464   Houston 0
5569217 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.t_inlet_m   2021-08-11 08:55:21 21,952454   Houston 0
5569218 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.YP  2021-08-11 08:55:21 25,64073    Houston 0
5569219 IMSCustomer1Device  ns=6;s=S71500ET200MP station_1.RASExt.HMIDualDPData.PV  2021-08-11 08:55:21 33,923695   Houston 0
```

Figure 4.1: Example of the sample data provided.

As we already had a set of real data points we decided that a far simpler and more accurate approach would be to simply use a set of real Rheosense data.
Our plan then became, build a script that would take a set of Rheosene data provided by IMS and loop over it while putting each data point in our development database, allowing us to test with real data without accessing the production database.
Python would be a suitable language for this as we were both comfortable with it, and it is suited for working with large data sets.

Code 4.1: liveData.py Imports

```python
import time
import requests
from datetime import datetime
```

The simulation script does not need many external packages, we only use time and datetime for handling date and time alongside requests to make request to the API we made.
For us to loop over the sample data we would first have to clean it a bit, removing any duplicates as there should not be any duplicates in the final version of the Rheosense.
The sample we are working with is a also just a fragment of the data taken from the database, therefore not all data types had an entry for each time stamp. The way the Rheosense unit works is that for a set time interval all the sensors sends a reading, this means that for each timestamp there should be an unique reading for each data type.
There appeared to be some bugs regarding the upload to the database with some timestamps having duplicate readings of a datatype or missing some dataype values.
These readings had to be removed, such that we would end up with data that resembled the intended behaviour of a Rheosense unit.
To achieve this we created a set of python functions to clean the data before inputting the resulting list into our data generator class.

4.2.1 Cleaning the data

These functions listed in Appendix A.1 and A.2 uses list comprehension and basic string manipulation techniques to extract the relevant data points while checking for duplicates. We also had some functions to determine the amount of different data types in the data set. Having determined the amount of data types we then split the list of data points into a list with the length equal to the number of data points, with each index of the list containing a new list with every data point of said value.

Code 4.2: liveData.py Data splitting support

```python
def split_data(data,datatypes):
    splitData = []
    print(datatypes)
    for _ in datatypes:
        category = []
        splitData.append(category)
    for datapoint in data:
        index = find_datatype(datapoint, datatypes)
        splitData[index].append(datapoint)
    return splitData
```

The data generator

With the data now in the Right format and ready to be put into the database, to do this we created a data generator class. We chose to make a it a class to make it usable in other script as well as make it easy to change parameters. This we we could change the data source, interval and push target easily.

Code 4.3: liveData.py Datagenerator

```python
class DataGenerator:
    def __init__(self,sample, timeToWait,target):
        self.target = target
        self.sample = sample
        self.counter = 0
        self.timeToWait = timeToWait
        self.limit = self.find_limit()-1
    def wait(self):
        time.sleep(self.timeToWait)
```

Here we see the class `DataGenerator` being defined, an init function is defined to initialize a new object of the `DataGenerator` class. The init function defines the objects attributes. With target being the API endpoint the data should be pushed to. Sample is the data set to be looped trough. `Counter` keeps track of which timestamp should be pushed. `timeToWait` is time time to wait between each generation and push. This value is then used in the class method `wait`. The `wait` method utilizes the time package to pause the execution of the script for a specified amount of time.

The `limit` attribute is used to ensure that when looping over the arrays the `counter` does not exceed the length of the shortest array. This `limit` is then found upon initialization by another class method detailed in the appendix A.3.

Code 4.4: liveData.py Methods for generating data

```
def generate_data(self):
        while True:
            datapoint = self.generate_datapoints()
            self.push_datapoints(datapoint)
            self.update_counter()
            self.wait()


    def generate_datapoints(self):
        datapoints = []
        for i in range(len(self.sample)):
            datapoint = self.sample[i][self.counter]
            datapoints.append(datapoint)
        return datapoints
```

In the code snippets above we see the main method of the DataGenerator and the method used for looping trough the sample data.
The `generate_datapoints` method uses the sample and counter attributes to loop over the sample and create a list containing the data values for each datatype. Counter keeps track of which time stamp to get the data values from.
The `generate_data` method then uses the `generate_datapoints` method to create a set of data points corresponding to a timestamp. Updates the counter before pushing the data to the target API endpoint, theses methods are shown in the appendix A.4, A.5, **??**. Finally it uses the `wait` method to wait for amount of time specified by the timeToWait attribute before continuing the loop.

When creating the data generation script we strove to follow good object oriented programming principles for the data generator class, with each function doing only one thing at a time. Hopefully making it easy to reason about the overlaying flow of the function and what it does even tough some of the list comprehension in the individual functions may be difficult to understand at first.

4.3 Building the database

Our goal when creating the database was to end up with a database that would be similar to the IMS production database. With a database schema a copy would be achievable, we were however not able to get such a schema. We therefore had to create the database based on the sample data we were given.But most likely a 1 to 1 transfer for our database with the IMS production database is not possible, as most likely the column names and even database names would be different. The SQL queries would most likely have to be rewritten to function with the production database.

Although we could not create an exact replica we could use the same technologies, namely Microsoft SQL-Server and Azure. As we were mainly developing using WSL creating a local MSSQL server was not straight forward, we therefore used a sqlite3 database during the early phase of development. This was done to get a prototype up and running fast, as we had previous experience with sqlite3 and it is used in the flask documentation.
Once we were ready to host on azure we migrated to a MSSQL server, only a few changes in the sql queries was necessary.

4.3.1 Creating the database schema

Working from the sample data we received we could then create a simple database schema to use for our development database.
Looking at the figure 4.1 we can see that each data entry has 7 columns, for our database we are only concerned with the value and time columns. Seeing as all the sensors send the data at the same time, the time column should be equal for all rows for the given sensor reading.
We decided to then keep all the sensor readings for one interval on the same row, resulting in the following schema listed in appendix A.8.
We use real as the type for all the values as they are decimal numbers. To make it easy to work with time, date is set as a datetime object. This makes the date easy to work with in both python and JavaScript. When creating

the database we also made queries for adding and retrieving data. We used these queries to create helper functions for the api to call.

4.3.2 Connecting Python with the database

For python to be able to communicate with the SQL Server database a database driver is needed. Microsoft recommends pyodbc in their documentation[22]. Python can then connect and run queries to the database with a connection string. The details of creating a connection string and using it to establish a connection to the SQL Server will be covered in chapter 4.2.2.

4.3.3 Inserting data

With the database connection sql queries can be ran to add data to the database.

Code 4.5: Adding data to the database

```python
def insert_data(conn, data1, data2, data3, data4, data5, ...
    data6, data7, data8, data9, data10):
    sql = ''' INSERT INTO RheosenseData(rpm_3, rpm_6, ...
        rpm_100, rpm_200, rpm_300, rpm_600, rho, t_inlet, ...
        yp, pv, Date)
                VALUES(?,?,?,?,?,?,?,?,?,?,?) '''
    try:
        cur = conn.cursor()
        cur.execute(sql, (data1, data2, data3, data4, ...
            data5, data6, data7, data8, data9, data10, ...
            datetime.now()))
        conn.commit()
    except pyodbc.Error as e:
        print("Error: {}".format(e))
        return -1
```

The function takes the connection to run the query on and data to be inserted as parameters. Then a string representation of the SQL query is made to specify where to insert the data. Placeholder values in the form of ? are used, this tells the database driver to escape the values. Escaping the

values is important as it protects against sql injections by ensuring that the values can not be interpreted as SQL commands.
Using the execute command sup lied with the SQL string and values on the connection then runs the sql query and returns the result in the cursor. The query is wrapped in a try except block to allow error handling. In this case the error is printed to the console and -1 is returned to signify a failed attempt.

4.3.4 Querying data

A similar approach is used when creating functions for querying data. When querying data we are interested in the data between to points in time.The same query method will be used to get the newest data since a specific time, historic data from a point in time and a snapshot of historic time.
The query function will take two points of time as input and the different API endpoints will supply those points in time. When querying data to reduce the amount of API calls we will always query for all the columns of data. seeing as they are on the same row this will not be much slower than getting only one column.
This results in the input of the query method being, the connection, a start time and an end time.

Code 4.6: Querying the database

```
def select_interval(conn,startDatetime, endDatetime):
    cur = conn.cursor()
    cur.execute(
        "SELECT * FROM RheosenseData WHERE Date BETWEEN ? ...
            AND ?",(startDatetime,endDatetime))
    datalist = []
    for (id, data1, data2, data3, data4, data5, data6, ...
        data7, data8, data9, data10, data11) in cur:
        datalist.append({
            "id" : id, "3_rpm" : data1, "6_rpm" : data2, ...
                "100_rpm" : data3,
            "200_rpm" : data4, "300_rpm" : data5, ...
                "600_rpm" : data6, "rho" : data7,
            "t_inlet" : data8, "yp" : data9,"pv" : ...
                data10,"Date" : data11    })
    return datalist
```

Since the SQL string is a bit shorter it is written directly in the execute function, the end result is the same. The SQL statement is ran on the connection with the result being returned in the cursor.
The SQL statement works by selecting all the data from the table specified that corresponds to the condition set by `WHERE`. In this case being all the rows with a date value between the two supplied time values from the RheosenseData table.
To prepare the result for the API a list of dictionaries is made by looping trough the contents of the cursor `cur`. This list is then returned, the format makes it simple for the API to return it as a JSON object.

4.3.5 Stats query

To implement some of the features in the front end we require the average, minimum and maximum values of the different columns. There exists inbuilt functions in SQL to achieve this. This allows us to obtain these stats directly from the database without handling the values with python. One problem we ran into was that the name of the column to run these built in commands on could not be supplied by placeholder values.
we therefore had to create individual functions for each column and build a switch to direct the API request to the correct functions. This switch is simply an else if method and can be found in the appendix A.3.

Code 4.7: Querying for stats

```
def select_stats_3rpm(conn,category,start,end):
    cur = conn.cursor()
    cur.execute("SELECT avg(rpm_3),min(rpm_3),max(rpm_3)  ...
        FROM RheosenseData WHERE Date BETWEEN ? AND ? ...
        ",(end,start))
    result = cur.fetchall()[0]
    stats = {
        "name" : category,
        "average" : result[0],
        "min" : result[1],
        "max" : result[2]
        }
    return stats
```

Here is an example of such a function, this function retrieves the stats for

the 3 rpm column.
It functions similarly to the query for data values but instead of returning all the values as denoted by Select * it now returns the requested stats as denoted by the `SELECT avg(rpm_3)...` function. These functions can not be ran on * or ? making it neccessary to create individual functions for each column.
Again the result is prepared for the API by inserting it into a dictionary before returning the dictionary.

4.4 Creating The APIs

After developing the Rheosense simulator and building the database we need something to allow them to communicate together, an API. Using the Python micro-framework Flask we can develop these APIs and allow for communication between the simulator and the database. The API's for this project also needs to allow the frontend, both mobile and web to be able to communicate with the database.

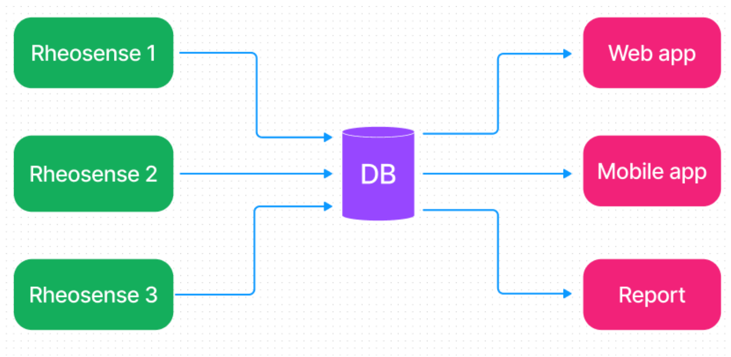


Figure 4.2: Flow of communication for the application.

Figure 4.2 illustrates the flow of data for this project, where the blue arrows represents the APIs. The Rheosense simulator will continuously push data to the database and the fronted applications will be able to retrieve this data. This section will cover the various APIs developed to make that possible.

4.4.1 Configuring the app

To get this project going we have to be able to add data from the Rheosense simulator into the database. As mentioned in chapter 3 we are going to use Flask to implement the necessary APIs. To get started we are first going to need to install Flask. This can be done through the terminal using pip3.

Code 4.8: app.py Flask Installation using pip3

```
pip3 install flask
```

Once Flask has been installed we created a new file named app.py. This is the file that is going to contain everything to do with the APIs. To start off we are going to import the following:

Code 4.9: app.py Imports

```
from flask import Flask, g, request
import json
from flask_cors import CORS
from decouple import config
import pyodbc
from azure_db import insert_data, select_data, ...
    select_interval, add_user, get_hash_for_login, ...
    get_user_by_email
```

In the first line we start off by importing `Flask` into our project including g and request. Flask provides us with two contexts. The application context and the request context. Here `g` is a special object associated with the current application context that is used to store data that can be accessed by multiple functions during a request. The application context is created before a request comes in and is torn down (deleted) after every request is finished. The `request` object is associated with the current request and is required to retrieve the request data [23]. For this project Pythons built-in package `json` will be used to work with JSON data. It includes a variety of methods for interchanging data formats, but this project will primarily utilize the `json.dumps` method [24].

Flask by default does not allow cross-origin requests. This means that all requests made to the server outside the server will receive a `Cross-Origin Request Blocked` error. To enable cross-origin requests we first have to install the Flask-CORS extension [25]. The following command shows how to install the extension using pip3, where the -U option will upgrade all the packages after the installation is complete.

Code 4.10: app.py Flask-CORS Installation using pip3

```
pip3 install -U flask-cors
```

Once the installation is complete, we can go ahead and import the CORS class from the extension as shown in line 3 in code snippet 4.9.
The application will need access to sensitive information like the database connection string, secret keys and both email username and password. To store these in a safe manner we are going to import the `config` object which is part of the `decouple` Python package [26].

As it is not a built-in package we will have to install it using pip3:

Code 4.11: app.py python-decouple Installation using pip3

```
pip3 install python-decouple
```

To be able to connect and work with the Azure SQL database in python a database driver is needed. Microsoft recommends using the pyodbc driver when working in python [22]. The driver can be installed using pip3:

Code 4.12: app.py pyocdbc Installation using pip3

```
pip3 install pyocdbc
```

At last we import all the database queries from the `setup_db` module covered in section 4.3. Now that all of the necessary imports are completed we can start configuring our application as shown in code snippet 4.13.

Code 4.13: app.py Configuring the app

```python
## Finds the secret key stored in a .env file
secret_key = config('secret_key')

## Creates and Configures the app
app = Flask(__name__)
CORS(app)
app.debug = True
app.secret_key = secret_key
```

1. `secret_key = config('secret_key')` sets the secret_key variable to the secret key value stored in the .env file using the config object imported from decouple.
2. `secret_key = config('secret_key')` creates an instance of the Flask class where the `__name__` parameter is the current module, which tells the app its path.
3. `CORS(app)` enables CORS on all domains and routes [25].
4. `app.debug = True` sets Flask in debug mode. This enables the debugger and hot reloading, which is useful during development.
5. `app.secret_key = secret_key` sets the secret key of the app to the secret key stored in the .env file. This key will primarily be used to securely sign the session cookies.

4.4.2 Establishing a connection with the database

As the app has been properly configured it now needs to be able to communicate with the database. Information needed to establish a connection to the database is found on the Azure Portal. This information is then put into the .env file mentioned previously which will be hidden. We can then use the method used previously to access the necessary values inside .env file as shown in code snippet 4.14.

Code 4.14: app.py Configuring the DB connection string

```python
## Azure Database Connection Configuration
server = config('server')
database = config('database')
username = config('username')
password = config('password')
```

Once the required values have been retrieved they can be put together into a complete connection string:

Code 4.15: app.py Completing the connection string.

```python
# Construct connection string
conn_string = 'DRIVER={ODBC Driver 17 for SQL Server};
SERVER='+server+';DATABASE='+database+';UID='+username+
';PWD='+ password
```

Now that the connection string is constructed we can go ahead and create a function `get_db()` that will return a connection to the Azure SQL database as shown below.

Code 4.16: app.py Creates the DB Connection.

```python
1 def get_db():
2     if 'db' not in g:
3         g.db = pyodbc.connect(conn_string)
4 
5     return g.db
```

In line 2 of the code snippet 4.16 the application context `g` checks whether a connection to the database has been made during the request. If it has it will return the current connection, and if it has not, it will return a new connection. This makes sure that only one connection to the database is made per request, thus reducing redundancy. This is especially important as connections to the database are quite expensive to make [27].

4.4.3 Closing the connection to the database

The database connection should also be closed after every request due to safety concerns, such as memory leakage. It could also lead to errors where there are too many connections open at once. To counteract this we want our connection to be closed after every request. For this purpose Flask provides the `teardown_appcontext()` decorator. Functions marked with this decorator will be called every time the application context is torn down, which is after every request [23]. The code below shows how we can use the Flask decorator to close the database connection if it exists.

Code 4.17: app.py Closes the DB Connection

```python
@app.teardown_appcontext
def teardown_db(exception):
    ##Closes the database at the end of the request.
    db = getattr(g, 'db', None)
    if db is not None:
        print("close connection")
        db.close()
```

Here the `tearrdown_db(exception)` function will be called whenever any request is finished and will close the connection if there is any. Decorators in Python starts with an `@` and are often used to add functionality or modify the behaviour of functions.

4.4.4 Creating the first API

Now that the app is configured and there are functions to help us connect and disconnect to the Azure database we are finally able to start developing the APIs required for this project. Following the Flask documentation we can utilize the `@app.routes()` decorators [28]. The decorator takes two parameters:

- `rule` - the URL string which will trigger the function below.

- `options` - are optional arguments e.g methods like POST or GET, where GET is the default argument.

For this project the Flask special routes decorator will be used to set up and specify our APIs. To start off building the first API we need to specify the URL and the methods to be used. As we want to push data from the Rheosense simulator into the database an intuitive URL name would be `/Add`. Secondly as we are pushing data from the client to the server, the method for this route has to be `POST`. A general rule of thumb is that when you are adding/updating data you should use the `POST` method, and when you are retrieving data you should use the `GET` method [29]. Now that we have established what our route decorator is going to look like we can go ahead and create the function it will trigger. The `/Add` API function should:

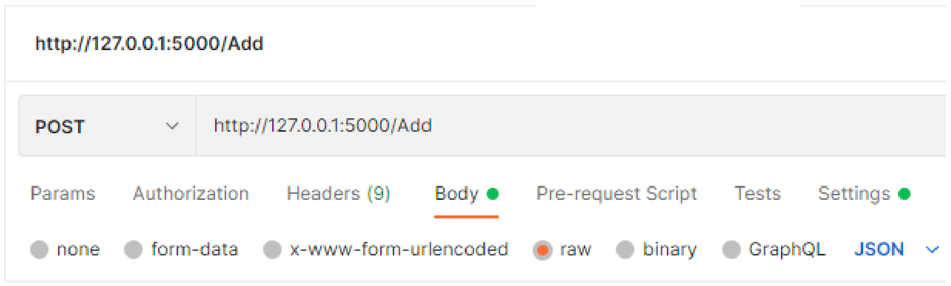
1. connect to the database
2. collect the data its been sent
3. insert the data into the database
4. return a `successful` message if it was indeed successful.

Code listing 4.18 shows how step 1-4 can be achieved. Note that we wrap the returning dictionary inside the `json.dumps()` method which changes the format from a python dictionary to JSON.

Code 4.18: app.py Add data to database API

```python
@app.route("/Add", methods=['POST'])
def push_to_db():
    conn = get_db()
    data = request.json
    insert_data(conn, data...)
    return json.dumps({'msg' : 'successful'})
```

Now that our Flask app is set up and we have created the first API, we can go ahead and run the application and check whether its working or not. To do this we are going to use the API platform Postman. Postman is a tool that lets developers send HTTP request to test and iterate their APIs. Before we can send a HTTP request we first have to run our application. This can be done by running the `python3 app.py` command in the terminal. Once its up and running Flask lets us now its running on localhost port 5000. We can now go ahead and enter the correct URL and method for our API as shown in figure 4.3.

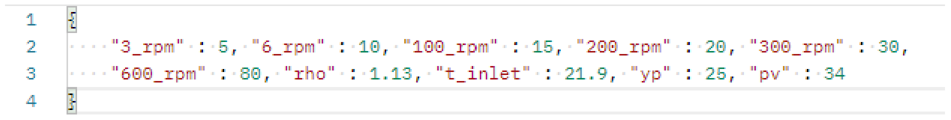


```
http://127.0.0.1:5000/Add

POST    http://127.0.0.1:5000/Add
```

Figure 4.3: Postman Configurations

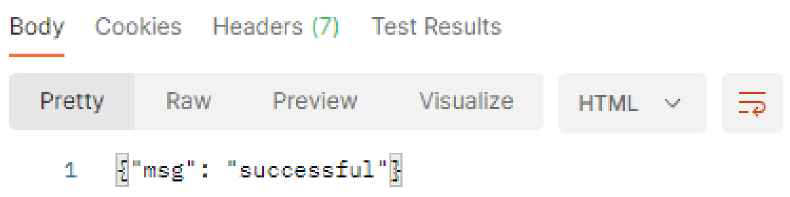
For our API to work correctly we also have to send it some data it can push to the database. Postman lets us send data, and we choose to do so in the JSON format as shown below:



```
{
    "3_rpm" : 5, "6_rpm" : 10, "100_rpm" : 15, "200_rpm" : 20, "300_rpm" : 30,
    "600_rpm" : 80, "rho" : 1.13, "t_inlet" : 21.9, "yp" : 25, "pv" : 34
}
```

Figure 4.4: Postman dummy data

Now that everything is set up and configured we can go ahead and send the HTTP request and wait for the response. The response from the Flask application returns a successful message as shown in figure 4.5.



```
{"msg": "successful"}
```

Figure 4.5: Postman HTTP response

Now the that the API is functional and running the Rheosense simulator is able to continuously add data every second to the database. Once the data has successfully entered the database it needs to be retrieved in order to create the desired graphical interfaces.

4.4.5 Get and GetInterval API

Developing the APIs responsible for retrieving data will follow a similar structure as the previous API. The main difference is that we are retrieving the data and not adding any, and will therefore not utilize the `POST` method. There are two ways we would like to collect the petroleum data for the frontend:

- `Get` which returns the entire data set.
- `GetInterval` which returns data between a specified timestamp.

The `Get` API utilizes the `select_data` query which retrieves all the records in the database. This API will primarily be called when a user wants access to historical petroleum data. It is fairly straightforward to implement this function and we will therefore focus on the `GetInterval` API. The API can be accessed by the URL `/GetInterval` using the `GET` method and requires an argument `Delta`. Here `Delta` is defined as the difference in seconds between two timestamps.

This argument can then be passed into the `select_interval` query together with the database connection to retrieve all records between those timestamps. Code listing 4.19 shows how this could be accomplished.

Code 4.19: app.py GetInterval API

```python
@app.route("/GetInterval", methods=['GET'])
def retrive_interval():
    conn = get_db()
    Delta = request.args.get("Delta")
    return json.dumps(select_interval(conn, Delta))
```

4.4.6 GetStats API

The last API necessary to create the graphical interface is the `GetStats` API. This API requires two arguments, `Delta` and `Column` where `Column` is either the name of one of the petroleum data points or all. These arguments are then passed to the `select_stats` query which returns key statistical data points.

4.4.7 Securing the APIs

Since we are going to host all of our APIs on Azure its important that only people with access are allowed to use them. We are therefore going to implement something called API keys. An API key is a key that is used to authenticate projects, clients or users that make a request to the API in question [30].

It is used to restrict access, which in our case is important as we are working with sensitive data. The API responsible for sending emails is also a prime target for abuse as spamming the email sender API will result in the users inbox being bombarded. To counteract this we are going to define a new custom decorator as shown in code snippet 4.20.

Code 4.20: app.py Custom decorator

```python
# Custom decorator function
def require_apikey(f):
    @wraps(f)
    def decorated_function(*args, **kwds):
        if request.args.get('api_key') and ...
            request.args.get('api_key') == config('api_key'):
            return f(*args, **kwds)
        else:
            abort(401)
    return decorated_function
```

To create our own custom decorator we import the `wraps` decorator from the the functools module. The syntax for creating the decorator is inspired by the functools documentation [31]. It is the code inside the `decorated_function` that is being run when the decorator is called. The function checks whether there is an `api_key` argument supplied and if it is the same as the one stored in the .env file. It will return a 401 internal server error if either is false. We can now add the custom decorator to the APIs that require authentication like so: `@require_apikey`.
For creating the key value stored in the .env file we used the follow script:

Code 4.21: app.py Create key

```python
from base64 import b64encode
from os import urandom
## Creates the secret key that will be stored in a .env file
random_bytes = urandom(64)
token = b64encode(random_bytes).decode('utf-8')
print(token)
```

4.5 Login and registration

Allowing users to login and register for our applications requires user registry. We will be creating a new database table, `Users`, that will serve as such a registry. The table should store the user's name, email, hashed password, role, permissions and the date the account was created. The code listing for the `Users` table can be found in appendix A code listing A.9.

4.5 Login and registration

Once our table has been created we want to create some useful queries we can run on the table. The following queries needed are:

- `select_users` query which will retrieve all users.
- `add_user` query that is going to add a new user to the database.
- `get_user_by_email` will return a user based on its email.
- `get_hash_for_login` used to validate login attempts.

All of these queries function in the exact same way as the ones in section 4.3 do. The major difference in the `Users` table is the `hashedpassword` column. This will store a hashed value of the users password as it is unsafe and considered bad practice to store the password as clear text. We will be importing `generate_password_hash` and `check_password_hash` from `werkzeug.security` to help us hash and verify passwords. When a user register a new account we will be using the `generate_password_hash` function to hash the password before adding it to the database.

Similarly when a user attempts to login we will use the `check_password_has` function to check whether the hash of the password being sent in during the login attempt matches the hash of the user stored in the database. The function below shows how we use the queries created and the imported functions to validate a login attempt from a user.

Code 4.22: app.py Validate_login function

```python
def validate_login(email, password):
    print(password)
    conn = get_db()
    hash = get_hash_for_login(conn, email)
    if hash == None:
        return False
    return check_password_hash(hash, password)
```

The API endpoint for login and register will be `/Login` and `/Register`, and both APIs will be using the `POST` and `GET` method. This is because both endpoints will be updating/adding data and returning data.

4.6 Deployment to Azure

In this section we are going to cover the Azures App Service and how we utilized that resource to host our backend APIs.

4.6.1 Creating the App Service on Azure

When creating the App Service resource we followed the procedure detailed by the Microsoft documentation [32]. These are the key configurations set for our App Service resource:

1. **Resource group** → rg-bachelor-development.
2. **Name** → bachelor-demo-ims.
3. **Runtime stack** → Python 3.8.
4. **Region** → Norway East.

We selected the App Service Plan B1 (basic plan) as recommended by Microsoft [32]. We then hit create and after a couple of minutes the App Service is successfully deployed.

4.6.2 deploying the Flask app to Azure

There are several ways of deploying your application to Azure. For this project we decided to deploy our application using Visual Studio Code. To be able to deploy via VS Code we first have to install the following extensions shown in figure 4.6.

Once you have the extensions installed you are able to login to your Azure account through the code editor. When you are logged into your Azure account you are able to view all your Azure Services through the editor.

Before selecting the option to deploy the application you first have to add a requirements.txt file containing all the packages used in the project. This needs to be done in able for the App Service to successfully run your app. This can be done by running the `pip3 freeze > requirements.txt` command in your terminal as long as you are in the correct directory. We can now go ahead and deploy the Flask app to the App Service created earlier. Once its successfully deployed you get access to key information about your apps performance on the Azure Portal website such as traffic in and out, requests made per minute, time per request etc.

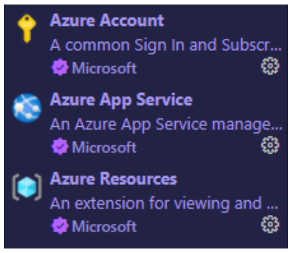

Azure Account
A common Sign In and Subscr...
Microsoft
Azure App Service
An Azure App Service manage...
Microsoft
Azure Resources
An extension for viewing and ...
Microsoft

Figure 4.6: VS Code extensions.

4.7 Automated Reporting Feature

This section will cover one of the main objectives for this project. An automated email reporting feature. This feature requires the use of different tools and can be divided into four steps:

1. Collecting the data for the last 24 hours.
2. Plotting the data in a satisfactory manner and saving the plot as a PDF.
3. Sending an email containing the PDF to all registered users.
4. Automatically repeat steps 1-3 every 24 hours.

Before we start collecting the data we are going to create a new route decorator like so:

Code 4.23: Send Report route decorator

```
@app.routes("/SendReport", methods=["GET"])
@require.apikey
def send_email():
```

Now every time the `/Send_email` URL is hit the `send_email()` function below will trigger if there is a valid API key supplied. It is inside this function that the report will be generated and sent off to all the registered users emails.

4.7.1 Collecting data

The first step to creating our plot is to gather the petroleum data for the last 24-hours. To do that we are going to first establish a connection with the database using the `get_db()` function. With the connection established we can go ahead and use the `select_interval` query to retrieve the desired records. Before executing the query we need to convert 24-hours into seconds, which gives us `Delta = 86400`. Now that the data has been successfully collected we are going to need to create a satisfactory chart to send to registered users.

4.7.2 Creating the plot

The code for generating the plot is going to take up a lot of space. In order to keep the code clean we are going to create a new file in the same directory as `app.py` that we are going to call `generate_report.py`. To start off we are going to import the following:

Code 4.24: generate_report.py Imports

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
```

Numpy and Pandas are going to assist in the handling of data while matplotlib is going to be the library responsible for plotting the charts. After importing the necessary libraries we are going to define a function `plot_report(datapoints)`.

The function has one required argument which is the data points that will be supplied by the `select_interval` query mentioned in the previous subsection. Code listing 4.25 shows the first couple line of codes inside the function.

Code 4.25: generate_report.py plot_report()

```
1 def plot_report(datapoints):
2     df = pd.DataFrame(datapoints)
3     ax = plt.subplots(figsize=(15,12))
4     ax1 = plt.subplot2grid((1,5), (0,0),
5           rowspan=1, colspan = 1)
6     ax2 = ax1.twiny()
```

We start off creating a DataFrame of the received data points. We then continue creating the figure that will contain the subplots where the figure's width and height is 15 and 12 inches respectively. Next is setting up the plot axis as shown in line 4-6.

When configuring the plot axis we specify how many there are (1,5) and its position in the figure (0,0). The last line of code twins the second y-axis to the first one. There are 10 such axis in total and they all follow the same structure as above with shifting positions to the right. As all the axis and subplots are in place we can now continue to plot our 24-hour data set onto the individual axis.

Code 4.26: generate_report.py plotting the axis

```
ax1.plot("3_rpm", "Date", data=df, color = "green",)
ax1.set_xlabel("3 rpm")
ax1.set_ylabel("Date")
ax1.xaxis.label.set_color("green")
ax1.invert_yaxis()
```

The first line plots the `3_rpm` and the `Date` column from the DataFrame to the x and y axis respectively. Further on the x and y label is set and colored. To finish off we invert the y-axis as reading petroleum data from top to bottom is preferred by most petrophysicists. This is because it intuitively resembles the depth of any particular well.

In addition to setting x and y labels we also define the bounds of the x-axis and x-ticks. This is not included in the code listing above as it takes up a lot of space and will be covered in detail in chapter 5. The rest of the axis follows a similar structure as `ax1`. Towards the end of the function we utilize the syntax shown in line 1 code listing 4.27 to make changes to multiple axis at once.

Code 4.27: generate_report.py applying style changes

```
1 for ax in [ax1, ax3, ax5, ax7, ax9]:
2         ax.grid(which='major', color='lightgrey',
3                 linestyle='-')
4         ax.xaxis.set_ticks_position("top")
5         ax.xaxis.set_label_position("top")
6         ax.spines["top"].set_position(("axes", 1.02))
```

in the code snippet above we configure the grid style of the subplots as well as configuring the top spines position and visibility. Here spine refers to the lines surrounding the subplot. The even number axis also have their top spines visible and are position on top of the odd numbered axis spines.
To finish off we only want the outer left- and right-hand side subplot to show their y-axis ticks. This is done by removing the y-axis ticks of axis 3-8 like so:

Code 4.28: generate_report.py applying style changes

```
for ax in [ax3, ax4, ax5, ax6, ax7, ax8]:
        ax.set_yticklabels([])
```

We end the function by applying the `tight_layout()` method which adjust the padding of the subplots and save the figure as PDF using the `savefig()` method. Passing some sample data into the function returns the following PDF containing the figure 4.7.

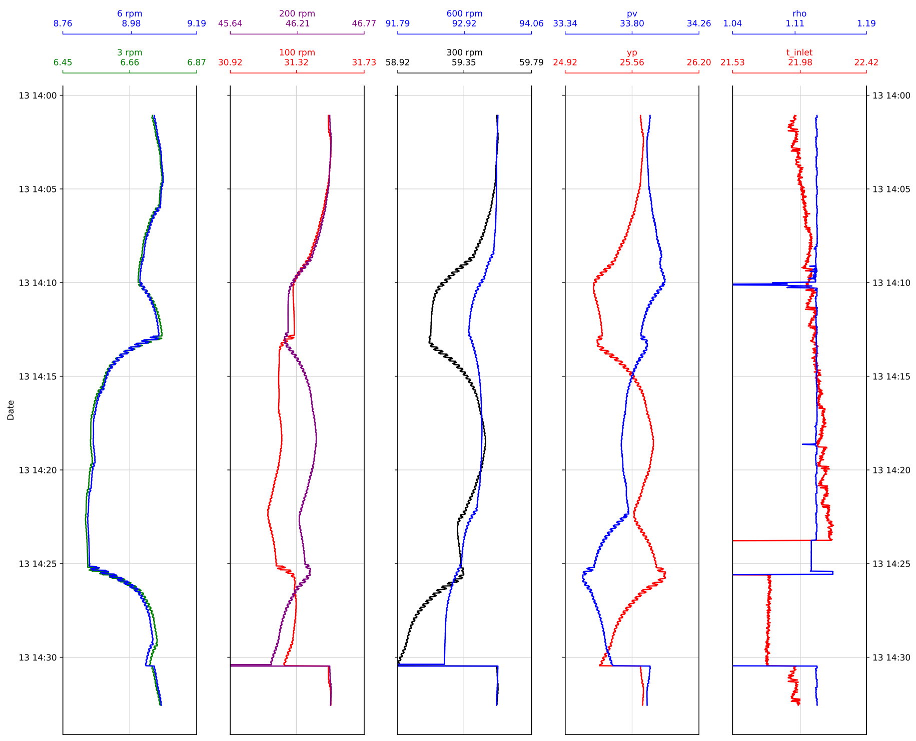


Figure 4.7: Plot produced by the plot_report function.

We can now import the `plot_report()` function into the `app.py` file where we will be completing the API responsible for sending the emails.

4.7.3 Sending emails to users

This project included the Flask-Mail extension to be able to send emails via the Flask application. Flask-Mail is not a built-in Flask extension, but can be installed using pip3. Upon completing the installation we want to import the Mail and the Message class. We now want to make the following configurations to allow our application to send emails through a Gmail account:

Code 4.29: app.py Flask app configurations for sending email via Google

```python
## Flask mail configurations
app.config['MAIL_SERVER']='smtp.gmail.com'
app.config['MAIL_PORT'] = 465
app.config['MAIL_USERNAME'] = config('email_username')
app.config['MAIL_PASSWORD'] = config('email_password')
app.config['MAIL_DEFAULT_SENDER'] = config('email_username')
app.config['MAIL_USE_TLS'] = False
app.config['MAIL_USE_SSL'] = True
```

`MAIL_SERVER` is set to Googles SMTP server for Gmail services, allowing us to connect to our own Gmail account using Flask [33]. In order for Flask to login to our Gmail account we have to set the `MAIL_USERNAME` and `MAIL_PASSWORD` configuration to our username and password.

These values are accessed through a .env file instead of being hard coded directly. The `MAIL_DEFAULT_SENDER` configuration is set to the senders email which makes the code responsible for sending the email cleaner as you wont have to specify the sender email each time. Now that our configurations are set we can create an instance of the Mail class as shown below.

Code 4.30: app.py Creates an instance of the Mail class

```python
mail = Mail(app)
```

Before sending any emails we first have to call the `select_users` query in order to collect all registered users and their emails. As the API will be sending the same email to several recipients we will be using the `mail.connect` method to send our emails as detailed in the Flask-Mail documentation [34]. Code snippet 4.31 shows the part of the code inside the `SendReport` API responsible for retrieving the users and sending them the mail containing the generated well report PDF shown in figure 4.7.

Code 4.31: app.py Part of the code responsible for sending the actual emails.

```
1  users = select_users(conn)
2  with mail.connect() as mail_conn:
3      for user in users:
4          msg = Message('Well Report',
5                      recipients = [user.email])
6          with app.open_resource("Well-Report.pdf") as fp:
7              msg.attach("Well-Report.pdf", ...
                   "application/pdf", fp.read())
8          mail_conn.send(msg)
```

The first line of code in the listing above retrieves the users while the second line establishes a connection with the Google SMTP server. It thereafter loops through every user and creates a new instance of the `Message` class and is supplied with a subject string and the users email. Before the email gets sent in line 8, the Well-Report PDF is attached. We can now use Postman to trigger the API to test whether it works or not. When calling the `SendReport` API we also have to remember to include the necessary API key for it to go through as it uses the `@require.appkey` decorator.

After sending a successful request to the API via postman, we can sign in to our Gmail account and check our inbox for the incoming email. The email has been sent successfully and the result is shown in figure 4.8.

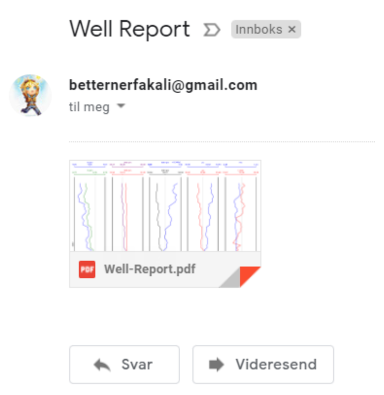


Figure 4.8: Email received with PDF.

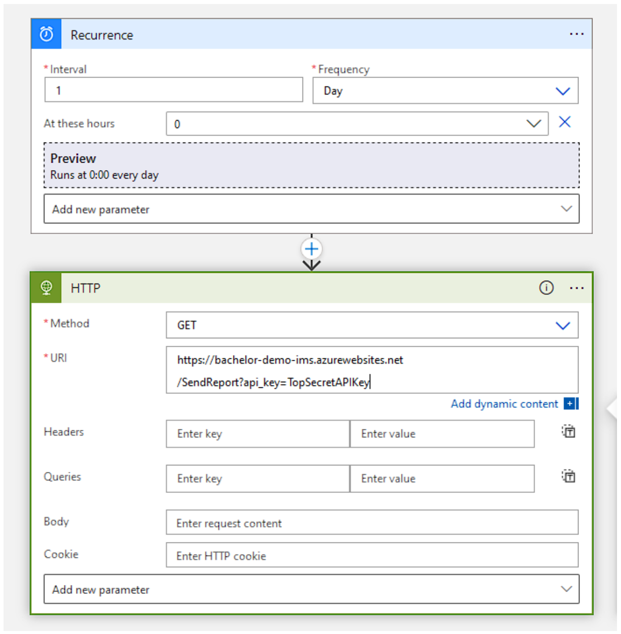
Here we can see that the subject name of the email is identical to the subject string in code listing 4.31 and that the Well-Report.pdf file is attached.
The PDF attachment can be opened for preview or downloaded directly from the Gmail inbox.

We now have an API that generates a plot containing petroleum data over the last 24-hours and sends that plot via an email.

In order to complete this feature we need to be able to trigger this API at time 00:00 every 24-hours. This is where Azures Logic Apps resource comes in handy.

4.7.4 Azure Logic Apps

The Logic App resource is a cloud-based platform for developing automated workflows. By workflow Azure simply means a set of triggers followed by a set of actions [20]. The specifics regarding the triggers and actions are customizable and up to the developer. The figure 4.9 below shows the configuration chosen for our Azure Logic App.



Recurrence

*Interval: 1

*Frequency: Day

At these hours: 0

Preview
Runs at 0:00 every day

Add new parameter

HTTP

*Method: GET

*URI: https://bachelor-demo-ims.azurewebsites.net
/SendReport?api_key=TopSecretAPIKey

Add dynamic content

Headers: Enter key | Enter value

Queries: Enter key | Enter value

Body: Enter request content

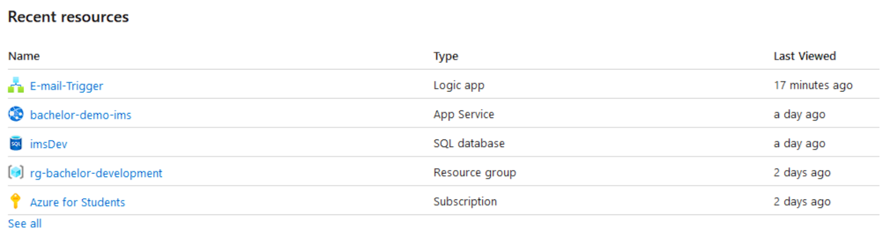
Cookie: Enter HTTP cookie

Add new parameter

Figure 4.9: Configurations for the Azure Logic App

The Recurrence tab specifies the automated workflow's trigger. Here we set the interval equal to 1 and frequency equal to the day. We want the trigger to occur at 00:00 and the preview reveals that the current configurations will run at 00:00 every day. Now that the trigger is configured correctly we want to configure the following action. We specify the action to be a

HTTP request using the GET method and the URL shown in the tab. Here the API key argument being passed is a placeholder for the actual key that is being used in production. We hit save and the resource is now up and running.



Recent resources

Name	Type	Last Viewed
E-mail-Trigger	Logic app	17 minutes ago
bachelor-demo-ims	App Service	a day ago
imsDev	SQL database	a day ago
rg-bachelor-development	Resource group	2 days ago
Azure for Students	Subscription	2 days ago

See all

Figure 4.10: List of all the Azure Resources used in this project.

Figure 4.10 shows all the Azure resources used in this project under the rg-bachelor-development resource group.

Chapter 5

Web Development

5.1 Structure

There is not a strict way to structure react an application according to the react documentation [35]. Although a common way of structuring your files is to group by functionality to avoid several layers of nested folders. This is the structure we chose to follow, creating folder for assets, components and helper functions. Assets are any pictures or other media that needs to be accessed. Components will contain all the components used in the project. The helper folder contains all the helper functions needed. Helper functions are JavaScript functions used by the components. Components themselves are also JavaScript functions. The key difference between a helper function and a component boils down to components returning JSX or a state while helper functions modify data.

5.1.1 Routing

React is a framework primarily used to make Single Page Applications, SPA‘s. This is also the case for our application. SPA‘s are web pages that are only loaded once from the server and then the content is dynamically changed by using JavaScript. To allow such applications to have different

pages a router is needed. A router intercepts https connections containing the URL for a new route, then shows that route. For the user it looks like a new web page has been retrieved form the server, but in actuality the same page is still shown but with new content. This reduces the amount of requests made to the server.
A popular library used for routing is React Router, which is used by companies like Microsoft and Netflix [36].
To match incoming URLs with the right content the react router has to maintain its own history stack that is based on how a web browser creates a history stack. This allows the user to navigate with the backwards and forwards pages as one would on a normal website [37].
To serve the right content to the user, React router will match the location path of the URL with the router configuration and find a match [37]. The location path of the URL is the part denoting which page it is looking for example the URL, https://www.website.com/page, here /page is the location path.
The router configuration is the route tree defined inside the react application, it contains all the routes and its corresponding components [37].

Code 5.1: Route tree

```
<Router>
  <div className="App">
    <NavBar />
      <div className='Content'>
        <Routes>
            <Route path="/" element={<LiveChart />}></Route>
            <Route path="/historic" ...
                element={<HistoricChart />}></Route>
            <Route path="/LogIn" element={<LogIn />}></Route>
         </Routes>
     </div>
    </div>
    </Router>
```

Above is the route tree for our prototype application. The router component wraps the entire application while the routes are defined inside the Routes tag. This allow for components defined outside the routes tag to be the same for all the routes, useful for headers, footers and navigation bars. When the location has been matched, the component mapped to the location will be rendered [37].

5.1.2 Rendering

In computer science rendering is used to describe the transformation of data into a graphical representation. The use of rendering in react is similar. It refers to the creation of the web page. One can think of it as the action of translating the code into the visual representation of the web page. To understand the implications of this we need to understand how a react page functions.
In a regular web page the blueprint for the page is called a Domain Object Model abbreviated to DOM. The DOM is a collection of object representations of the website. This allows programming languages like JavaScript to interact with the website, changing and even adding content to the web page [38].
A react page however maintains a Virtual-DOM. React's virtual-DOM is a collection of the react components as JavaScript objects. When the page is being rendered react works through the virtual-DOM collecting all states, props and runs any functions.
The entire virtual-DOM is not rendered on each re-render, only the objects below the object that triggered the re-render. This has major performance effects. Best case scenario being an object far down in the object tree changes, only it and those objects below it re-renders. On the flip-side if one is not careful this can cause a minor change at the top of the tree to trigger a re-render of the whole page.
There are tools like memoization in the form of the UseMemo Hook to prevent designated calculations from being re-run on each render.

5.1.3 State and props

There are two main tools for handling component data in react, state and props.
A general rule of thumb is that state is data that lives inside a component while props are passed into the component. Changing the state will also trigger a re-render of the page, unlike props that can be changed without causing a re-render.

Managing re-rendering is a key part of developing react components. A common pattern is to pass a state from one component into another as a prop. This is the pattern of state management that is used in this project. It is easy to reason about in a small application with a unilateral data flow. For applications with a more complex data flow, a more advanced pattern is optimal, like the redux pattern.
When working with state in functional components one has to use something called React hooks.

5.1.4 Hooks

Hooks are a novel concept in React. Definition from the react documentation reads as follows; "Hooks are functions that lets you hook into React states and lifecycle features from function components."[39].
React provides a set of built in hooks to do common tasks, while also allowing developers to create their own custom hooks. In this project the hooks used are the useState, useEffect and useMemo hooks.
The useState hook is a hook used to work with states. When using useState it creates the state and a function to update the state.

Code 5.2: example of useState

```
const [time, setTime] = useState(1800);
```

Above is an example of useState from the project. The state called time is declared and initialized with the value 1800 and with a function setTime to update the state.

The useEffect hooks is a hook that provides much of the same functionality as onMount and similar methods from regular JavaScript. It allows the running of code based of the life-cycle of objects in the Virtual-DOM , often referred to as side effects in React [39].

Code 5.3: example of useEffect

```
useEffect(() =>{
  fetch(`urlString`,{method: "GET",})
  .then(res => res.json())
  .then((data) =>{
      setStats(data);
  })
}, []);
```

Above is another example from our project where a useEffect hook is used to fetch data from our API on the initial render of the page. Here we also see the setStats method being used to update the state called Stats.
React hooks will run on every render of the application and by design is not allowed to be run conditionally. Every render of the page must run the same amount of hooks, one can therefore not put a hook inside an if statement [39].
This can cause performance problems, especially when fetching data from an API. This default behaviour would cause every change on the page to run all the effects and calling the API for every re-render. React provides a tool to solve this problem. A second parameter can be added to the use-Effect. This parameter is often referred to as a dependency array. React will not run the useEffect if the contents of the dependency array has not changed.
In the above example an empty array is passed as this conditional argument. This means that the contents will never change and the effect will only be run on the initial page load.

Another complication of every hook being ran on each render of the page is when an expensive computation is required. This computation gets calculated for each re-render even if its inputs has not changed.
The useMemo hook is reacts tool for solving this problem. Much in the same way as in useEffect, a conditional argument is passed to the useMemo hook. This conditional argument will be an array and checks for changes.

Code 5.4: example of useMemo

```
const data = useMemo(() => ...
    CreateStatsData(stats,categories),[stats,categories]);
```

Above we see the useMemo hook used when creating a stats table in the project. The result of the useMemo hook is memoized, meaning that that value will be returned on future renders, unless the dependency array has changed.

5.2 Charting

One if the main objectives of this project is improving the graphical presentation of the rheology data gathered by the RheoSense unit. To do this we need to create a chart that represent the data in a recognizable and useful form for mud engineers.
One of the main requirements is that time is represented as descending on the y-axis with the data represented vertically. This is to intuitively represent how the data changes as the drill bit moves down the well.

How deltas (variation) in the data value is represented must also be kept in consideration. The data being represented is used to describe the properties of the drilling mud. In section 4.2 we discussed how the values are quite stable. This stability has to be represented in the graph. This means that the domain of the x-axis must be small enough to show changes but not to small such that a change of a few decimal points is shown as a large fluctuation.

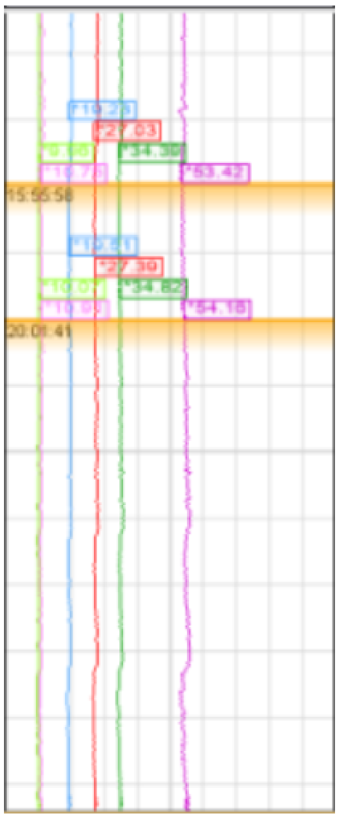


Figure 5.1: Sample Graph provided by IMS

With this in mind we set out to find a suitable charting library. Recharts was a natural choice as it is a charting library based on d3.js made specifically for React.

It uses configurable components to build charts. It is also released under the MIT licence allowing it to be used freely in a commercial product.

5.2.1 Chart component

When developing the graph we started by hard coding in each graph and making changes in different graphs allowing us to compare designs easily.
We worked with our supervisor from IMS to come up with a working design using this method. We determined that grouping two values in each graph created a view that was space efficient and readable.
To allow for suitable domains on the x-axis stacking two x-axes was needed.
To improve the readability of the graph we also determined that each line should not be centered on the middle of the x-axis, but rather shifted to separate sides.
This lead us into the problem of defining the x-axis domains.

Determining the x-axis

In most cases the x-axis can simply be determined automatically by the charting library. This approach is problematic for live data representation, as the domain will be calculated differently as the data changes.
This causes the x-axis ticks to shift, making it difficult to spot changes in the values by observing the line. This problem is most observable in short time representations.
We decided that the best cause of action would be to define a static x-axis.
This created desirable results. When working with hard coded graphs this was easy as we could look at the graph and create the domain.
However for a general graph component we need a way to determine the domain based on the values supplied to it. We decided that the most consistent way of determining the axis was to look at the historical data.
To do this we require statistical data for the historical data. This lead to the creation of the GetStats API endpoint detailed in 4.4.6. With the introduction of stats we created several helper functions to create x-axes from the supplied stats.

Code 5.5: The SetAxis function

```
export function SetAxis(stats,index){
    let axis = CreateAxis(stats);
    let amount = 0.125
    if (isEven(index)){
       return ModifyAxis(axis,"left",amount);
    }
    return ModifyAxis(axis,"right",amount)
}export default SetAxis;
```

The function above is responsible for creating the desired x-axis. It enables two x-axis to be stacked on the same graph as it shifts the first axis to the right, and the second axis to the left. The CreateAxis, ModifyAxis and isEven functions can be found in the appendix A.10, A.11, A.12.
When determining the axis in CreateAxis we look at the difference between the average value and maximum value. Then a value equal to 3 times this difference is added to the average to create the upper bound of the axis domain. To find the lower bound we subtract the same value from the average.
It was necessary to use this convoluted method due to the presence of some data readings with the value 0. These values should not be present in the final Rheosense version as 0 values should be filtered out. In the future a simpler function looking at the max and min values should be sufficient.
For cases where the difference is lower than 1, we set 1 as the domain, subtracting 0.5 and adding 0.5 to the average for the respective start and end points of the domain.
Then for even index numbers the axis is shifted to the left, moving the line to the right of the original center. For odd index number the opposite is the case. When these axes domains are provided it results in a graph with two lines that are of center to different sides, making the graph easier to read.
With the problem of creating generalized axes solved we could then start to create a generalized chart component.

General chart component

The goal of a General chart component is to create a component that creates a chart based on the given data values. Due to the x-axis constraints the stats are also needed. We also want the chart to contain multiple lines so

it needs to handle being supplied with more than 1 set of data.
After receiving the data from the API it is in the form of a JSON object. When processed it then takes the form of a list of dictionaries.
The dictionaries has a key that is the name of the column and a value that is a list of all the values in the column. When inputting the whole list to the chart and setting the key for a line equal to one of the column keys this creates a line with the values from the value list.
This allows us to create the following component:

Code 5.6: Chart.js - Generalized chart component

```
<LineChart className='charts' width={325} height={900} ...
    data={items} margin={{MarginSpecifications }} ...
    layout='vertical'>
   {categories.map((category,index)=>{
       return (<>
        <Line
           xAxisId={index}  key = {"Line"+index} ...
               dataKey={category}
           type="monotone" stroke={colours[index]}
           isAnimationActive={false} dot={false} ...
               allowDataOverflow={true}
        />
        <ReferenceLine
           xAxisId={index}  key = {"refLine"+index}
           x={stats[getStatsIndex(category,stats)].average}
           stroke={colours[index]} strokeDasharray="3 3"
        />
        <XAxis
           xAxisId={index}  key={"axis"+index}type="number"
           domain={
               SetAxis(stats[getStatsIndex(category,stats)],
               index)}
           orientation="bottom" dataKey={category} ...
               allowDataOverflow={true}
        />
    </>);
       }
    )}
     <CartesianGrid stroke="#ccc" strokeDasharray="5 5" />
     <YAxis dataKey="Date" type="category" width={100}/>
     <Tooltip />
     <Legend verticalAlign="top" />
    </LineChart>
```

There is a lot to unravel in this component, it follows the recharts pattern of creating a chart. A rechart chart is a component containing the component parts of a chart with options passed in as props.

- `<LineChart />`, this is the main component containing all the parts options configuring the layout is given as props alongside the data for the chart.
- `categories.map((category,index)`,
  Here we see the power of JSX allowing us to write a JavaScript function mapping over the supplied array of columns names
  creating a `<line />`, `<ReferenceLine />` and `<Xaxis />` component for each category.
- `<Line />`, the line component takes in all its settings as props.
  Of particular interest is the `xAxisID` and `key` props, they are present in all of the mapped components.
  `xAxisID` ensures that the lines belong to the correct x -axis.
  `Key` is strictly not necessary but the browser complains in the console if mapped items does not have unique ids, making troubleshooting difficult. `dataKey` declares which dictionary is going to provide the values for the line.
- `<ReferenceLine />`, creates a line denoting the average value for the last 24h.
  the value x is average value that is found from the stats, like with data all the stats are in a array. here the `getStatsIndex` function gets the correct index for the category being mapped, this index is then use to get the corresponding stats.
- `<XAxis />`, the XAxis component uses the same props as the lines to with some additional props for configuration. Here the `SetAxis` detailed earlier is used to define the axis domain. The setting
  `allowDataOverflow` is set to true, this is to handle the 0 value points. If the 0 points are removed in a future version this should be set to false.
- The next components are pretty self explanatory, for the `<YAxis />` Date is supplied as the datakey and it is set to be a category. This

> means that the y axis is not a true representation of time if there are missing data points. How this can be solved in the future will be covered in chapter 7

Now that we have a working chart component, for us to utilize it to make live and historic charts we need a way to retrieve the correct data.

5.3 Stats

Having collected statistics about the average, minimum and maximum values for the previous 24 hours to create the charts. We decided that it would be relevant and easy to show them to the user.
We created a stats component that gets passed the stats as a prop and returns a form displaying the stats prop.
This component contains quite a bit of boilerplate code and parts has been shown when describing useMemo. Thus it will only be shown in the appendix A.14.
In short a list of dictionaries is defined to create a coupling between data and table fields. Then the table component is mapped over the data and this dictionary. The code written here mostly follows the documentation[40].

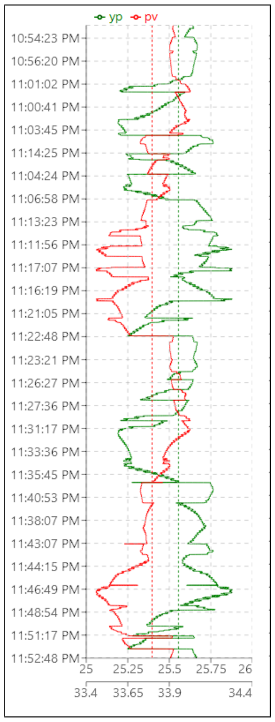


Figure 5.2: Chart made by chart component

5.4 Obtaining data

We are making both a live and historic chart. Retrieving data for the historic chart is fairly trivial when we have created an API. We can use JavaScript‘s fetch API to make calls to our API endpoint specifying the time interval we want the data from. Since we are using the previous 24 hours stats,

we can use this method to get stats for both historic
and live charts. When creating a live chart however
each time one wants to get updated data a connection to the server needs to be made. If this needs to happen in real-time sending an API call every few seconds can be performance intensive for the server. For real-time updates using a websocket connection is an alternative. If some delay is acceptable buffering can be used instead. We did consider exploring this option, as creating a websocket connection with flask can be done easily with socket.io. But after talking with IMS and considering how the charts would be used it was decided that real-time representation of the data was not that important. Mud engineers want to see the data for the last 24,12 and 4 hours, the second to second change is not that important. IMS‘s future RheoSense algorithms will represent the data as 5 or 1 minute points. This means that instead of an API call every few seconds a call every minute is made, this reduces the performance impact on the server and we can use the same API endpoint for both historic and live data.

5.4.1 Fetching data

To get the data needed for our charts we need to create 3 different fetches.

- A fetch for getting historical data, needs to run on page load.
- A fetch for getting live data, this fetch needs to run on page load and at a set interval.
- A fetch for getting stats from the previous 24 hours data.

Utilizing useEffect we can achieve all 3 of the fetches. this results in the following useEffects;

Code 5.7: HistoricChart.js - fetch hook for historic data

```
useEffect(() =>{
    fetch(`BaseURL/GetInterval?∆=${encodeURIComponent(time)}`
    ,{mehtod: "GET",})
    .then(res => res.json())
    .then((result) =>{
        setIsLoaded(true);
        let formattedResult = Format(time,result)
        setHistoricItems(formattedResult);
    })
  }, [ time]);
```

This useEffect functions in the same way as the useEffect shown in section5.1.4. The differences is the API endpoint and that the `Format()` function is used to format the date. This function is shown in the appendixA.13. Before using setHistoricItems to set the HistoricItems state. Since the stats fetch has been discussed earlier it wont be repeated here, we will instead focus on the more interesting interval fetch hook instead.

Code 5.8: LiveChart.js - fetch hook for livec data

```
useEffect(() => {
  const interval = setInterval(async () => {

    await fetchData();
  }, fetchInterval);
  return () => {
    clearInterval(interval);
  }
}, [liveitems], time);
```

To make the code easier to read the fetch has been moved into the fetchData function. We use the built in JavaScript method setInterval to run the fetch function in an interval. This interval function is set on mount by the useEffect, fetch will not keep running in an interval.
To ensure that the interval stops when the component is unmounted a cleanup function is required. This is done by returning a function to remove the interval. This function will then run when the component is unmounted.

Code 5.9: LiveChart.js - function for interval fetch

```
const fetchData =  async () => {
    let fetchDelta = time
    if(liveitems.length>0){
      fetchDelta = fetchInterval/1000;
    }
    console.log(fetchDelta);
  await fetch(`baseURL/GetInterval?Δ=$
  {encodeURIComponent(fetchDelta)}`,{
  method: "GET",
})
    .then(res => res.json())
    .then(
      (result) => {
        setIsLoaded(true);
        let formattedResult = Format(time, result);
        if(liveitems.length>0){
          let mergedResult = MergeResult(liveitems, ...
              formattedResult);
          setLiveItems(mergedResult);
        }
        else{
            setLiveItems(result);
        }
      },
      (error) => {
        setIsLoaded(true);
        setError(error);
      }
  )
}
```

The fetch works similar to the historic fetch showed earlier with two additions to improve performance. First an if condition checks the length of the liveitems array containing the data, if this list is empty it means that this is the first API call. For the first API call we need the data for the entire period we intend to visualize, we then use this period as the the delta. If the data array has a length that is greater than zero, this means that data from previous calls are already in the array. We can now fetch only the data that has arrived from the previous call, we do this by setting the delta equal to the interval.
The interval is set in milliseconds and the delta in seconds, hence the fetchInterval/1000. The MergeResult helper function is then use to append the new data points to the liveitems array, and maintaining the length by re-

moving the items no longer part of the time period.
For example in the case of a 12 hour visualization with 1 minute fetch interval; the first call retrieves the data for the whole 12 hour period, while following calls only retrieve data for the 1 minute interval between fetches.

5.5 Creating Pages

With routing, charts and fetches sorted we can now create the pages for visualizing live and historic data. Since react router assigns components to an URL these pages are made as components.
These components are rather large and the important code snippets are shown above, we will therefore not show them in the text but they will be present in the appendix A.15A.16.

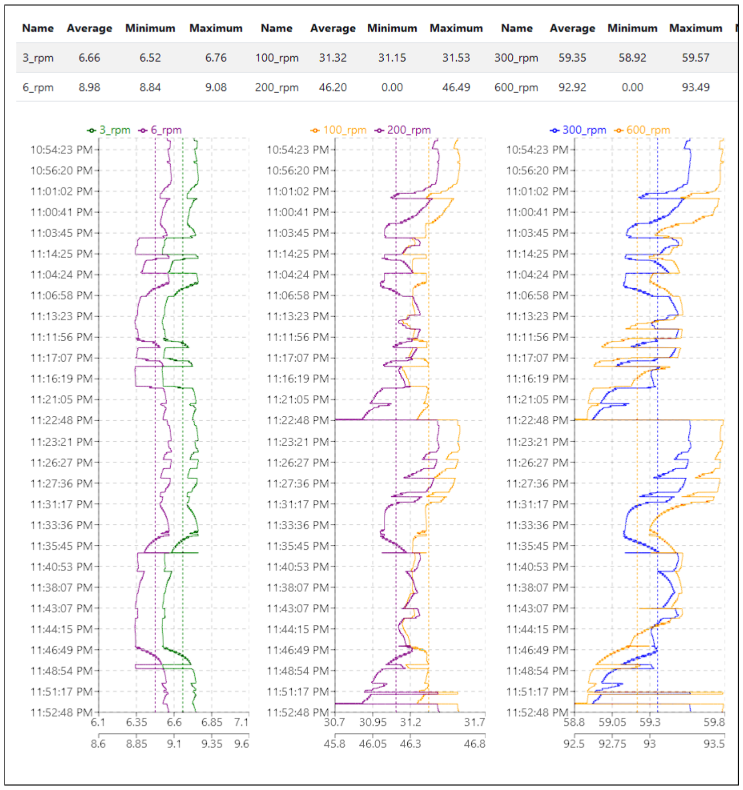
Code 5.10: HistoricChart.js - fetch hook for historic data

```
 const HistoricChart = () =>{
    const [time, setTime] = useState(14400);
   //...repeated for remaining states
    const handleClick = (time) => {
        setTime(time) }
useEffect(Stats);
useEffect(historicfetch);
if(!isLoaded ){    return(<>Loading....</>)
}else if(!StatsLoaded){ return(<>Loading....</>) }
else{
return(
<> <div className='category-container'>
             <Stats stats = {stats} ...
                 categories={["3_rpm","6_rpm"]}/>
            <div className='chart-combiner'>
            <Chart items={HistoricItems} stats={stats} ...
                isLoaded={isLoaded}
             error={error} categories= {categories} ...
                 colours={colours} />
            </div>
         #... repeated for desired categories
         <button className='timeOption' onClick={ () => ...
             handleClick(1800) }> 30 Minutes </button>
         #... repeated for desired time options </>)}}
```

Above is a shortened representation of the HistoricChart component, the LiveChart component follows the exact same structure. In fact with some further development this can be generalized to a single component receiving the fetch method as a prop.
In both components the following 5 steps are performed;

- First the required states are defined, we see the time state being defined. The time state defines the time period to visualize data for.
  States to hold the data and stats are also defined, the use of them has been shown earlier.
  Additionally 3 states controlling the rendering of the component are defined. These are error, isLoaded and statsLoaded. These states are set in their respective fetches. They are necessary due to one of the rule of react, namely that all components must be rendered. This means that the Chart page components must be rendered before any of the stats has been fetched. Thus if the stats are not loaded a loading message will be shown instead, the same is true for error encountered in the fetching.
- Then a function to set the time period by clicking on a button is defined.
- Following this functon are the two useEffects used to retrive stats and data
- Before JSX is returned the control states has to be checked to determine which JSX to return
- Lastly the JSX is returned. We see that the JSX returned when everything has been loaded uses the Stats and Chart components shown earlier in addition to buttons allowing the user to specify the time period.

The next page will display what the resulting charts will look like on the web page.



Name	Average	Minimum	Maximum	Name	Average	Minimum	Maximum	Name	Average	Minimum	Maximum
3_rpm	6.66	6.52	6.76	100_rpm	31.32	31.15	31.53	300_rpm	59.35	58.92	59.57
6_rpm	8.98	8.84	9.08	200_rpm	46.20	0.00	46.49	600_rpm	92.92	0.00	93.49

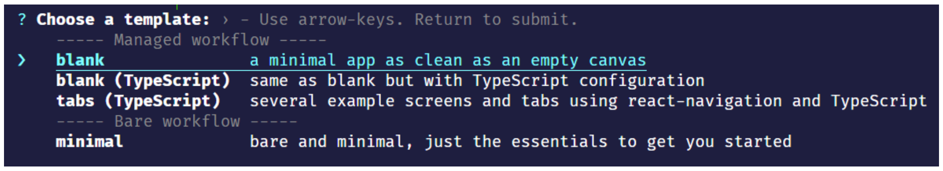
Figure 5.3: A part from The Historical Data Page, showing 3 generated charts

Chapter 6

Mobile Development

6.1 Structure

Expo will be the framework used to build the React Native app as detailed in section 3. Expo can be installed using the package manager for JavaScript, npm. Scaffolding a new project in expo can be done by running the command `expo init project-name` where project-name is replaced by the actual name of the project. When Scaffolding a new project Expo will serve you a prompt where you are asked to select one of the following templates:

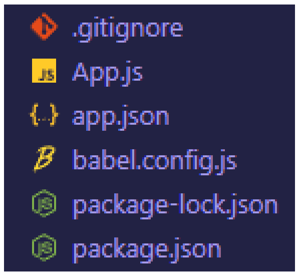


```
? Choose a template: › - Use arrow-keys. Return to submit.
    ----- Managed workflow -----
❯   blank                 a minimal app as clean as an empty canvas
    blank (TypeScript)    same as blank but with TypeScript configuration
    tabs (TypeScript)     several example screens and tabs using react-navigation and TypeScript
    ----- Bare workflow -----
    minimal               bare and minimal, just the essentials to get you started
```

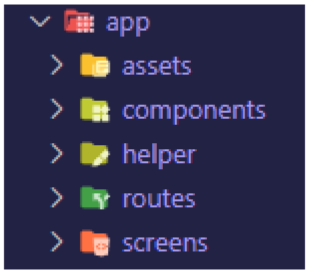
Figure 6.1: Selecting a template for the app

For this project we will select the blank template as we want to start fresh and add tools and libraries as we go. Once the template has been selected Expo will install all the packages and dependencies needed for the project.

When the installation is complete the files shown in figure 6.2a including some Expo specific configuration files will appear in the project directory. This includes a .gitignore and App.js file. App.js is the file that is being executed when the app starts.



(a) App.js and configuration files



(b) app folder structure

Figure 6.2: File structure for the app

For this app we want to add the folders shown in figure 6.2b. The folders will contain the following:

1. **assets** → images and other media resources.
2. **components** → style components used to create custom JSX components.
3. **helper** → helper functions similar to the one in chapter 5.
4. **routes** → will contain the stack navigator component.
5. **screens** → JavaScript files for displaying different screens.

Before diving into the code responsible for each individual screen we are going to take a closer look at the `styles.js` file inside the components folder. This file will be responsible for exporting style components to the entire app.

6.1.1 Custom styled components

When creating our own custom style components we are going to import the following:

Code 6.1: styles.js Imports

```
1 import {View, Text, Image, TextInput, TouchableOpacity} ...
      from 'react-native';
2 import styled from 'styled-components';
3 import Constants from 'expo-constants';
4 const StatusBarHeight = Constants.statusBarHeight;
```

We start off by importing the default JSX components from the React Native library that has been covered briefly in chapter 3. These are the components we will apply styling to. We then want to install styled-components using npm and import it as shown in line 2 in the code snippet above. The last import required for our custom components are the expo-constants. expo-constants provide useful information about the system the app is being run on. This information lasts throughout the lifetime of your app's install and supports all devices and the web [41]. We will the system information to find the height of the status bar on the device running the app. The status bar of any given iOS or Android device is the top section of the screen displaying thing like battery life and time as shown in the figure below:

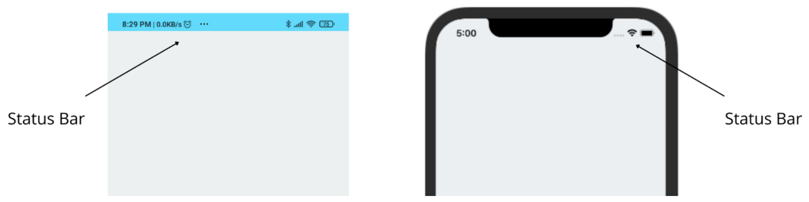


Figure 6.3: Status bar [42]

Finding the height of the device's status bar is important as you don't want any of your components to end up at the top of your screen overlapping with the status bar.

To the styling process smoother we will go ahead and define a set of colors for our mobile app as such:

Code 6.2: styles.js const Colors

```
export const Colors = {
    primary: "#ffffff",
    secondary: "#E5E7EB",
    tertiary: "#1F2937"
};
```

Now we have everything we need to create our first custom styled component. When creating our components we will be following the styled-components documentation [43]. The first custom component we want to create is a styled container that is based off the standard view component.

Code 6.3: styles.js First custom style component.

```
export const StyledContainer = styled.View`
    flex: 1;
    padding: 25px;
    padding-top: ${StatusBarHeight}px;
    background-color: ${primary};
`;
```

Here `flex: 1` means that the container will grow in proportion to the screen size. `padding-top` is set to the height of the device's status bar in pixels and the `background-color` is set to primary. This container can now be imported and used in all JavaScript files in the project.

There are several custom styled components like this in the project and they all follow a similar structure, but with different styling depending on their usage. As we now have the desired styling for the app we can now go ahead and create our first screen.

6.2 Login screen

The code responsible for creating the login screen will be put into a new file. We will call this file `Login.js` and it will be placed inside of our screens folder. To start off we are going to import `React` and `useState` to be able to create our login component as shown in code listing 6.4. Component states in React Native functions the same way it does in React which is detailed in section 5.1.3.

Code 6.4: Login.js Login component.

```
import React, {useState} from 'react';

const Login = ({navigation}) => {
    const [hidePassword, setHidePassword] = useState(true);
    const [message, setMessage] = useState();
    const [messageType, setMessageType] = useState();
    const [googleSubmitting, setGoogleSubmitting] = ...
        useState(false);
};
export default Login;
```

In the login component there are four states that we need to keep track of. These states will be used to update and change the UI based on their values. The four states introduced are responsible for keeping track of:

- `hidePassword` is a boolean that will be used to toggle password visibility on or off. It should update every time the user presses a toggle password button.
- `message` is the state responsible for returning a message to the user regarding their login attempt. The state should update every time the user attempts to sign in.
- `messageType` keeps track of whether the login attempt was successful or not. It should also update every time the user attempts to sign in.
- `googleSubmitting` is a boolean that updates every time the google sign in button gets pressed. This state should update every time the user attempts to sign in via Google.

6.2.1 Creating the form

The login screen must include a form to allow the user to enter their credentials. There exists external libraries for creating forms using React Native. This project includes the Formik open source library to assist in creating forms for our app. A common problem when dealing with form on mobile devices is that the keyboard that appears tend to cover the input section. To counteract this we will be importing the react-native-keyboard-aware-scroll-view component to our project. This component allows the app to automatically scroll down if the appearing keyboard overlaps with the text input field being used. Both these libraries needs to be installed via npm before they can be imported into the project.
Code listing 6.5 shows how the login screen will be structured.

Code 6.5: Login.js Scroll view implementation.

```
return (
    <KeyboardAwareScrollView>
        <StyledContainer>
            <InnerContainer>

            </InnerContainer>
        </StyledContainer>
    </KeyboardAwareScrollView>
);
```

Here the `KeyboardAwareScrollView` component wraps around our custom styled components imported from `styles.js`. It is inside the `InnerContainer` component we will create our form using Formik. We want our login form to contain a text input field for the users email and two text input fields for entering and verifying their password. At the end of the form there should be two buttons. One for logging in with the input credentials and one for signing in via Google.

The official Formik documentation offers guides and insight into creating forms like the one needed for this app. The following code listing is inspired by the Formik documentation [44].

Code 6.6: Login.js Creating a form using Formik.

```
1  <Formik
2  initialValues={{email: '', password: ''}}
3  onSubmit={(values, {setSubmitting}) => {
4      if (values.email == "" || values.password == "") {
5          handleMessage("please fill in all the fields!");
6          setSubmitting(false);
7      } else {
8          handleLogin(values, setSubmitting);
9      }
10 }}
11 >
```

We start off by passing empty strings as the initial values for the email and the password field. In line 3 the `onSubmit` function is run whenever the form is submitted. It is supplied the values from the the input fields as well as the `setSubmitting` method. This method is required as it needs to be set to false at some point to finish the submission cycle as detailed in the documentation [45].

The function then goes on to validate if both the email and password field is left empty. If either one of them are, the `handleMessage` function will be called with a string argument attached and the submission cycle will end. The `handleMessage` function will update both the `message` state and the `messageType` state. This will cause the DOM to re-render and show the "Please fill in all the fields!" message to the user after the failed login attempt.

If the email and password field is not empty however, the `handleLogin` function will be called with the field values and the `setSubmitting` method. This is the function responsible for carrying out the HTTP request to the `/Login` API and will be covered the next subsection.

There are a total of three text input fields used for the form in the login screen. We will take a closer look at how the email field works and how it is structured. Code snippet 6.7 shows the options selected for the text input component for the form.

Code 6.7: Login.js Email field.

```
<MyTextInput
    label="Email Address"
    icon="mail"
    placeholder="example@gmail.com"
    placeholderTextColor={Colors.darkLight}
    onChangeText={handleChange('email')}
    onBlur={handleBlur('email')}
    value={values.email}
    keyboardType="email-address"
/>
```

The options `label`, `icon`, `Placeholer` and `PlaceholderTextColor` are all related to styling. The other options all serve some functionality that we will take a closer look at.

- `onChangeText` function updates the email value to whatever is being typed into the field.
- `onBlur` is useful when keeping track of which input has been touched.
- `value` sets the field value equal to email.
- `keyboardType` is going to set the keyboard type that will appear when tapping the input field to the native email keyboard type. This makes it easier for the user to enter their emails as the layout is modified to include symbols like the at-sign as shown in figure 6.4.

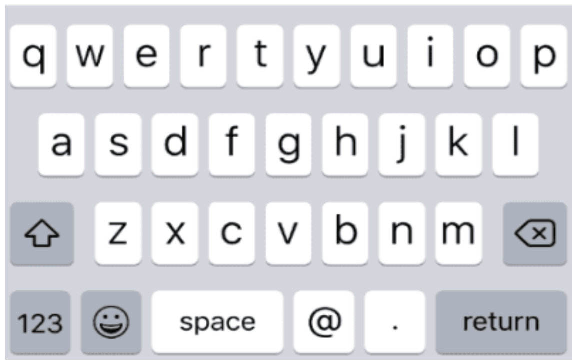


Figure 6.4: Email keyboard type [46]

The remaining two input fields, password and verify password both follow the same structure as the input field in code listing 6.7 apart from some styling options. Coming towards the bottom of the screen we are going to want a JSX component to display error messages to the user and a login button to submit the form. Code listing 6.8 introduces a new React concept called conditional rendering.

Code 6.8: Login.js Error message and login button.

```
1 <MsgBox type={messageType}>{message}</MsgBox>
2 {!isSubmitting && (
3 <StyledButton onPress={handleSubmit}>
4     <ButtonText>
5         Login
6     </ButtonText>
7 </StyledButton>
8 )}
```

Here `isSubmitting` is a boolean state that tells us whether the form is in submission or not. Line 2 reads as follows; if the form is not submitted, render the following. This is whats called conditional rendering and `&&` is the React syntax for doing so. This also means that once the button is pressed `isSubmitting` will be set to true and the condition in line 2 will not be met and the button will therefore not render. This is considered good practice as the button press will go on to call an asynchronous function when sending the credentials to the login API. During this time the user can tap the login button 5 more times and cause a whole host of issues. What we are going to do instead is to create a second button to be rendered as shown in the code snippet below.

Code 6.9: Login.js Second login button.

```
1 {isSubmitting && (
2 <StyledButton disabled={true}>
3     <ActivityIndicator size="large" color={Colors.primary} />
4 </StyledButton>
5 )}
```

As shown in the first line, this button will render if `isSubmitting` is set to true. We want to disable this button as it serves no purpose outside of the visual. To finish off we add an `ActivityIndicator` component which renders a smooth "loading" graphic inside the button, letting the user know the login attempt is being processed.

Figure 6.5: Activity Indicator, iOS and Android [47]

The buttons for signing in via Google functions the exact same way as the regular login button except for the boolean that is responsible for the conditional rendering. Instead of using the `isSubmitting` Formik state we will be using our own `googleSubmitting` state.

6.2.2 Sending a login request

When the form is submitted it should send a HTTP request to the backend login API. We are going to create a function `handleLogin` that will be responsible for sending the request. The function should accept the form credentials and the `setSubmitting` method as arguments. The login credentials will be sent to the backend for validation and the `setSubmitting` method will be used to end the submission cycle when the request is finished. We will be using React's built in function `fetch` to send our request using the URL below.

Code 6.10: Login.js Login URL

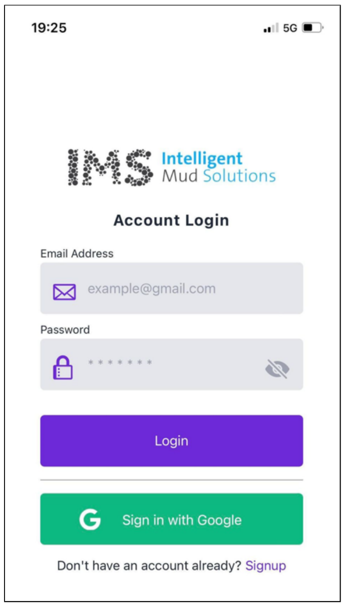
```
const url='https://bachelor-demo-ims.azurewebsites.net/Login'
```

We will use the method `POST` when sending the request and add the login credentials supplied to our request body as shown in the code snippet below.

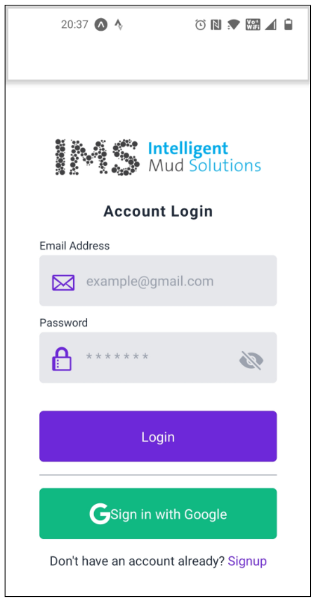
Code 6.11: Login.js Request body

```
body: JSON.stringify({
                email: credentials.email,
                password: credentials.password
              })
```

The `JSON.stringify` method will convert the JavaScript values into the JSON format. It then goes on to catch the response from the server. An error message will be shown to the user if the request was not successful and if it was successful it will redirect the user to the home screen. The process for navigating between screens will be covered in section 6.5. At the end of the request, whether it was successful or not `setSubmitting` will be set to false to end the submission cycle. To make the login screen more appealing we will add an IMS logo, page title and icons where it makes sense. Figure 6.8 shows what the login screen looks like on both iOS and Android.



(a) Login screen for iOS



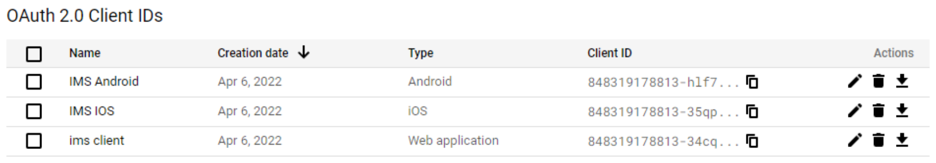
(b) Login screen for Android

Figure 6.6: Login screen for both iOS and Android.

The screen looks mostly identical between iOS and Android except for the Google button text. A bug appeared where the button text would disappear on Android whenever padding was applied. As a temporary solution we deployed platform specific code that would not add this padding to the button text if the device was an Android.

6.2.3 Sign in via Google

To be able to utilize Google OAuth 2.0 in our app we need to first obtain the OAuth 2.0 client ID's for both iOS and Android. Following Google's documentation[48] we were able to create the necessary client ID's as shown below:



OAuth 2.0 Client IDs

☐	Name	Creation date ↓	Type	Client ID	Actions
☐	IMS Android	Apr 6, 2022	Android	848319178813-hlf7...	
☐	IMS IOS	Apr 6, 2022	iOS	848319178813-35qp...	
☐	ims client	Apr 6, 2022	Web application	848319178813-34cq...	

Figure 6.7: Activity Indicator, iOS and Android [47]

We can now go ahead and install and import the `expo-google-app-auth` module as it provides Google authentication integration with our Expo app [49].

Code 6.12: Login.js Imports

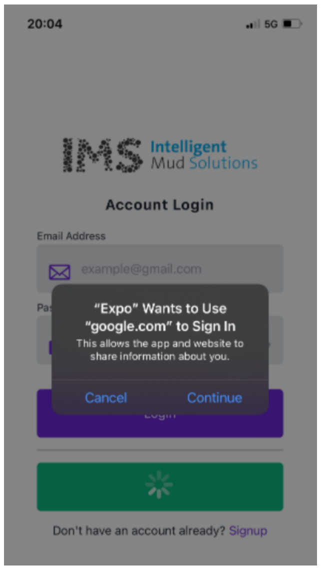
```javascript
import * as Google from 'expo-google-app-auth';
```

Once the module has been imported into the project we want to create a new function `handleGoogleSignin`. This function will be responsible for handling the authentication via Google. Inside our function we want to declare a new constant `config` that is going to store our iOS and Android client ID's. We can then use the `logInAsync` function from the imported Google module to start the authentication process. The function requires the `config` constant to be passed as an argument. We will take a close look at how the `logInAsync` works in code listing 6.13.

Code 6.13: Login.js Authentication process

```
1  Google
2      .logInAsync(config)
3      .then((result) => {
4          const {type, user} = result;
5
6          if (type == 'success') {
7              const {email, name} = user;
```

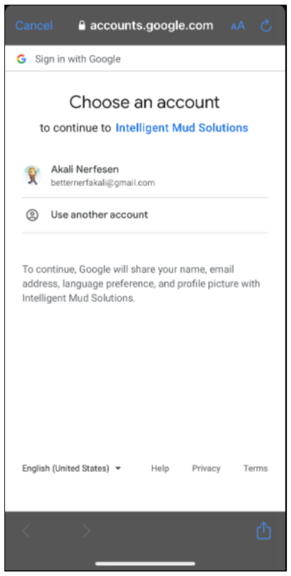
The asynchronous function returns a promise of type `Google.LogInResult`. If the promise gets rejected we will return an error and set `googleSubmitting` state to false. If, however, the promise gets fulfilled we will extract both type and user from the result as shown in line 3. Type has two potential values, `'cancel'` or `'success'`. If type is equal to `'success'` we continue to extract necessary information from the user. Here user refers to the authorized Google account and there a multitude of values you are able to collect. We are however only interested in the users name and email. Figure 6.8 illustrates the Google authentication process for our app.



"Expo" Wants to Use "google.com" to Sign In
This allows the app and website to share information about you.

Don't have an account already? Signup

(a) iOS native prompt



to continue to Intelligent Mud Solutions

To continue, Google will share your name, email address, language preference, and profile picture with Intelligent Mud Solutions.

(b) Google sign in

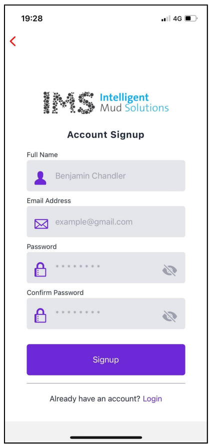
Figure 6.8: Google Sign in process

6.3 Registration screen

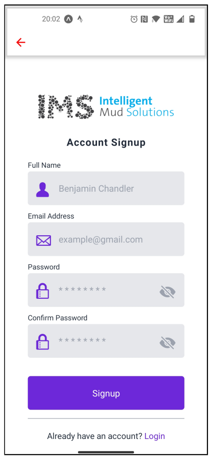
The registration screen should look almost identical to the login screen with just a few tweaks. It will therefore follow the same structure as detailed in the previous section. They key modifications made to the registration screen are:

- An additional input field for the user's name and only a single register button. The Google button should not be included.
- A function, `handleSignUp` responsible for carrying out the HTTP request made to the `/Register` API. This function works exactly like the `handleLogin` function in the previous section.

There are no new concepts introduced for the development process of the registration screen and the code follows an identical structure as the code in Login.js. We will therefore not spend much time detailing the code generating the registration screen. The Figure below shows what the sign up screen looks like for both iOS and Android.



(a) iOS



(b) Android

Figure 6.9: Registration screen for iOS and Android

6.4 Home screen

The home screen for the mobile app should be where the charts are displayed to the user. Selecting a charting library for React Native was quite difficult as the library used for the web application (Recharts.js) does not support React Native. We therefore had to find a new library to assist in creating our charts. After testing numeral libraries we ended up going with the victory-native charting library. We start off by creating the `Home` component as shown below.

Code 6.14: Home.js Home component

```
const Home = ({navigation, route}) => {
}
export default home;
```

The component receives two props, `navigation` and `route` which will be discussed in section 6.5. The goal of this component is to return a UI containing live charts. As victory-native does not access values the same way Recharts does we have to declare a state for each data point in our data set. The first step towards having a continuously updating chart is to obtain data continuously. We will therefore start by implementing a function, `getDataAsync` that is going to responsible for fetching data. We will also be importing and utilizing the `useEffect` hook for calling the `getDataAsync` function at a set interval. The process for fetching data in React Native is very similar to how its done in React. We will therefore not spend much time going through all the details as it's already been covered in section 5.4.

What is different however, is the victory-native library and how its used to create charts. Before importing victory-native into our project it first has to be installed using npm.

Code 6.15: Home.js victory-native Import

```
import { VictoryChart, VictoryLine, VictoryTheme } from ...
    'victory-native';
```

We import two components, `VictoryChart` and `VictoryLine`, and the `VictoryTheme` constant from the library. The two imported components lets us create the necessary lines and graphs while the constant gives us out of the box styling.

Code 6.16: Home.js VictoryChart and VictoryLine component

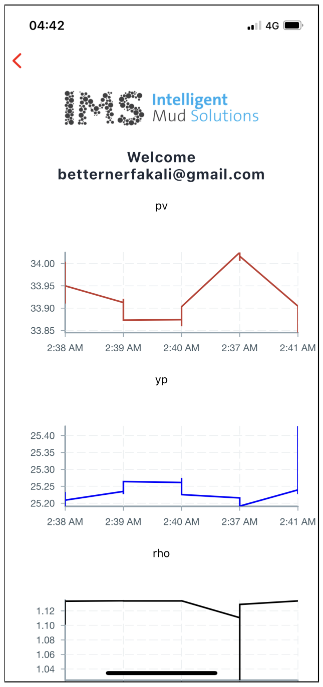
```
<VictoryChart
  scale={{x: 'time'}}
  theme={VictoryTheme.material}
  height={200}
  >
    <VictoryLine
        style={{
        data: { stroke: "#c43a31" },
        parent: { border: "1px solid #ccc"}
        }}
      data={chartPv}
    />
</VictoryChart>
```

Code listing 6.16 shows we can use the `VictoryChart` and `VictoryLine` components to create a chart containing a single line. We specify the theme of the chart using `VictoryTheme`. The data used for plotting the graph is accessed directly through the `chartPv` state which is of the format:

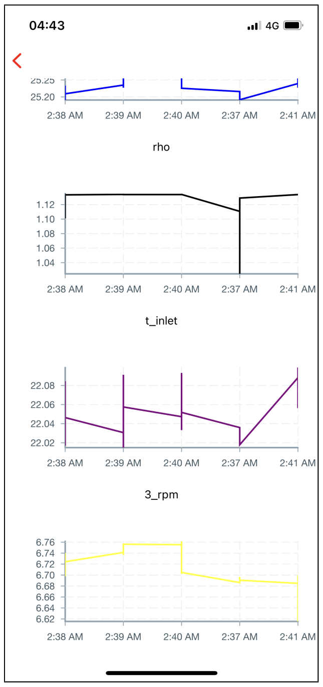
Code 6.17: Home.js chartPv format

```
chartPv = [{x: Date, y: pv}...]
```

The charts are set up to show one dta line. There are 10 data points in our data sets which means there will be a total of 10 charts visible on our homepage. As every chart is 200 pixels in height plus padding and margins, this is going to take up a lot of space. This will lead to charts ending up "outside" our screen. To counteract this we are going to wrap all of our charts inside a `ScrollView` component. The `ScrollView` component can be imported through the react-native library and enables scrolling on our screen. Figure 6.10 shows what the charts on the home screen looks like.



(a) VictoryChart top



(b) VictoryChart scrolled down

Figure 6.10: Home.js Illustrates some of the charts created by victory-native

6.5 Routing

As all the screens for our app are now assembled, we can go ahead and connect them together. To do this we are going to create a new file, `RootStack.js` inside of our routes folder. To be able to navigate to different screens we are going to import all of our components inside the screens folder and the following react navigation libraries:

Code 6.18: RootStack.js Imports

```
import { NavigationContainer } from ...
    '@react-navigation/native';
import { createNativeStackNavigator } from ...
    'react-native-screens/native-stack';
```

The stack navigator functions just like a normal stack. You start of with an inital screen at the bottom of the stack and every time you navigate to a different screen that screen gets put on the stack. Whenever you exit said screen it gets popped of the stack. To get started we will create a stack using the `createNativeStackNavigator` function. We then continue creating our `RootStack` component. Following the React Navigation documentation[50] we will implement the following into our component:

Code 6.19: RootStack.js RootStack component

```
<NavigationContainer>
    <Stack.Navigator initialRouteName="Login">
        <Stack.Screen name="Login" component={Login} />
        <Stack.Screen name="Signup" component={Signup} />
        <Stack.Screen name="Home" component={Home} />
    </Stack.Navigator>
</NavigationContainer>
```

We include the components we want to be able to navigate between, and set the initial screen to be displayed when opening the app as the login screen. To finish off we import our `RootStack` component in our App.js file. We have to do this because the `App` component gets rendered first as the app is launched.

Code 6.20: App.js Adding the RootStack component.

```
import RootStack from "./app/routes/RootStack";

export default function App() {
  return <RootStack/>;
}
```

Code listing 6.21 shows an example of how to navigate between screens once the RootStack has been created and configured properly. The app can be published to Expo with the command `expo publish`.

Code 6.21: Login.js Example of how to navigate between screens.

```
<TextLink onPress={() => navigation.navigate("Signup")}>
```

Chapter 7

Results, Conclusion and future development

To conclude the project we will summarize the result of our work and if we managed to achieve our goals set in the outline in section 2.5. Then we will outline how future development can improve the result.

7.1 Results

Starting with the minor objectives;

- We created a script that could push RheoSense data to our development Microsoft SQL-Server database that was hosted on azure.
- We designed a graph that displayed data in a format recognizable to mud engineers.
- we created an API for the database and hosted the API on azure. We were able to use this API in our other tools. With some changes to the SQL queries this API can be used with IMS‘s database.
- A log in system was created for the app, this log in system could be migrated to the website to enable authorization.

Our performance in the minor objectives directly impacted the results of our main goals.

- We created components for displaying the data in a satisfactory manor.
- With the components we created we were able to create a website displaying data received from our API. We did not host this website as we were not able to migrate our authorization system from the mobile application.
- We built a tool that could use our API to generate 24h reports. By hosting it on Azure we were able to automate the creation and sending of reports at set time intervals. We also protected the API endpoint with an API Key.
- We created and published a mobile application that had a login and registration system and some rudimentary charts created with data from our API. Due to limitations in the charting libraries for React-Native we were not able to replicate the charts from the website.

7.2 Discussion and further work

7.2.1 Web application

We are pleased with the charting components and the fetching of data for the we application.
One problem we encountered is that with live charting for 12 and 24 hour periods hovering over the charts to display the tool tips can be unresponsive. Likely due to the sheer amount of data points. As IMS‘s plans to move towards reporting the data as averages of say 1 or 5 minutes this would likely not be an issue. We can still do some more work making the chart pages more generalized. Mainly using mapping to create the individual chart components based on a list of categories.
Log in and authorization however is the pain point in the web application. We intended to use Google's OAuth 2.0 for the login section just like with the mobile application, as the client ID for the web is already configured. Since we had some difficulties with setting up Microsoft SQL Servers locally

we had to start development using PostgreSQL for mobile and Sqlite3 for the web application. This meant that we did not have time to migrate the login system and implement authorization via protected routes in flask and JSON Web tokens.
This is the first thing to be done for further development. The library used for creating the forms on mobile, Formik, is also support by React and can be used to implement the login and registration forms our web application.

Mentioned earlier was IMS‘s intent to represent data as averages of 1 or 5 minutes blocks instead. Included in this is also the option of then zooming in on each of these blocks to reveal the full data.
We believe this can be implemented in the application by using caching, Where the data is fetched and then cached on the client side. From this cache the display data is calculated and shown. Clicking on a point can then zoom the chart in on the raw data from the cache. The chart components we have made can handle this, all that is needed is caching and some middle-ware to calculate the averages.
We could also consider using the stats API to calculate this on the server side. Further investigation would be needed to determine the best course of action.
The most important part however is to get the log in and authorization working, after this the web application can be hosted on azure in the same manner as the API. It would then be ready for live demos using their test database. As such this is the first course of action when continuing development.

7.2.2 Mobile application

We are pleased with how the both the login and registration screen turned out for our mobile application. The charts however, is quite lackluster compared to the charts produced by both the email report system and the web app. This comes down to mainly two reasons. First off we had a different development cycle for the mobile application. Our plan was to start of with developing both the login and register screen first and then reuse much of the code for the live charting from the web app. This led to much of the time spent on the mobile app was used developing the login and registration and screen and left little time left for the home screen. The second reason for our charts not being quite satisfactory for the app is that the library used for the web app was not compatible with React Native. Quite a lot of time went into understanding, reading the documentation and developing using the Recharts library. Finding a new library that supports React Native and repeating the same process was both difficult and time consuming. We tested multiple libraries to find one that best suits the needs for our application. The library we ended up going with, victory-native, did not have everything we needed for our charts, but it was the best we could come up with given the time we had left. Our charts on the mobile app suffered as a result. Further development will go into exploring new options for charting with React-Native.

7.3 Conclusion

We set ourselves some ambitious goals when starting the project. In many ways we feel that we achieved most of them, while still leaving room for improvement in all parts of the project.
Even when focusing on technologies that we were familiar with we still had to research plenty of new technologies and libraries.
We had no previous experience hosting projects, and this required quite a bit of research. We still feel like there is plenty of room for improvements in this aspect.
We could possibly have benefited from cutting parts out of the projects. By cutting out the mobile part more care could have been put into the creation of the web application. Our intent behind using React and React-Native was to reuse code from the web application in the mobile app. Due to the difference in charting libraries for web and mobile this was not possible. To achieve this we would most likely have to create our own charting library which would be a massive undertaking.
By not creating a back end we could also have focused on creating all the front end visualization. Freeing up more time for bells and whistles, like buttons to control the axes as we already have the underlying functions. This would severely limit the possibility of creating the reporting tool and result in limited viability for live demos.
Our exploration of mobile development also provides insight in to the work required to represent the data as a phone app. Taking this into consideration we are happy with our choices and feel that the results of the project offer an excellent starting point for further development.

Bibliography

[1] D. Williamson, “Drilling-fluids.” https://www.slb.com/resource-library/oilfield-review/defining-series/defining-drilling-fluids. Accessed: 02.03.2022.

[2] Kirk-Othmer, “Rheology and rheological measurements.” http://www.123seminarsonly.com/Seminar-Reports/008/62345855-Rheology-One.pdf?msclkid=13f15e34c7a711ec8b38b34cdba91096. Accessed: 07.05.2022.

[3] Schlumberger, “Oilfield glossary.” https://glossary.oilfield.slb.com/en/terms/s/shear_stress. Accessed: 07.05.2022.

[4] Schlumberger, “Oilfield glossary.” https://glossary.oilfield.slb.com/en/terms/s/shear_rate. Accessed: 07.05.2022.

[5] Schlumberger, “Oilfield glossary.” https://glossary.oilfield.slb.com/en/terms/p/plastic_viscosity. Accessed: 07.05.2022.

[6] Schlumberger, “Oilfield glossary.” https://glossary.oilfield.slb.com/en/terms/y/yield_point. Accessed: 07.05.2022.

[7] “Gitflow-image.” https://www.bitbull.it/en/blog/how-git-flow-works/. Accessed: 14.04.2022.

[8] “Mobile-market-share.” https://gs.statcounter.com/os-market-share/mobile/norway/#monthly-202103-202203-bar. Accessed: 15.04.2022.

[9] “Web-frameworks-figure.” https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/#professional. Accessed: 15.04.2022.

[10] "React-components-image." `https://ixorasolution.com/blog/context-a-way-to-communicate-between-components-in-reactjs`. Accessed: 15.04.2022.

[11] "React-native-documentation." `https://reactnative.dev/`. Accessed: 25.04.2022.

[12] M. Budziński, "React-native." `https://www.netguru.com/glossary/react-native`. Accessed: 25.04.2022.

[13] "Expo-documentation." `https://docs.expo.dev/`. Accessed: 25.04.2022.

[14] "Python-documentation." `https://docs.python.org/3/`. Accessed: 25.04.2022.

[15] "Python-survey." `https://www.jetbrains.com/lp/python-developers-survey-2020/`. Accessed: 26.04.2022.

[16] "Flask-documentation." `https://flask.palletsprojects.com/en/2.1.x/foreword/`. Accessed: 26.04.2022.

[17] "Pandas-documentation." `https://pandas.pydata.org/`. Accessed: 26.04.2022.

[18] "Matplotlib-documentation." `https://matplotlib.org/`. Accessed: 28.04.2022.

[19] "Azure-data-center-regions." `https://azure.microsoft.com/en-us/global-infrastructure/geographies/#geographies`. Accessed: 29.04.2022.

[20] "Azure-logic-apps." `https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-overview`. Accessed: 01.05.2022.

[21] "Azure-app-service." `https://docs.microsoft.com/en-us/azure/app-service/`. Accessed: 02.05.2022.

[22] "Microsoft-pyodbc." `https://docs.microsoft.com/en-us/sql/connect/python/python-driver-for-sql-server?view=sql-server-ver15`. Accessed: 03.05.2022.

[23] "Flask-application-context." `https://flask.palletsprojects.com/en/0.12.x/tutorial/dbcon/`. Accessed: 03.05.2022.

[24] "Python-json." https://docs.python.org/3/library/json.html. Accessed: 03.05.2022.

[25] "Flask-cors." https://flask-cors.readthedocs.io/en/latest/. Accessed: 03.05.2022.

[26] "Python-decouple." https://pypi.org/project/python-decouple/. Accessed: 03.05.2022.

[27] "Database-connection-expense." https://quick-adviser.com/is-database-connection-expensive/. Accessed: 03.05.2022.

[28] "Flask-routes-decorator." https://flask.palletsprojects.com/en/2.1.x/api/#flask.Flask.route. Accessed: 03.05.2022.

[29] "Get-vs-post." https://lazaroibanez.com/difference-between-the-http-requests-post-and-get-3b4ed40164c1. Accessed: 03.05.2022.

[30] "Api-keys." https://cloud.google.com/endpoints/docs/openapi/when-why-api-key. Accessed: 05.05.2022.

[31] "Python-functools." https://docs.python.org/3/library/functools.html. Accessed: 05.05.2022.

[32] "Azure-app-service-documentation." https://docs.microsoft.com/en-us/azure/app-service/quickstart-python?tabs=flask. Accessed: 05.05.2022.

[33] "Google-smtp-server." https://pythonbasics.org/flask-mail/. Accessed: 04.05.2022.

[34] "Flask-mail." https://pythonhosted.org/Flask-Mail/. Accessed: 04.05.2022.

[35] "React file stucture." https://reactjs.org/docs/faq-structure.html. Accessed: 28.04.2022.

[36] "React router." https://reactrouter.com/. Accessed: 01.05.2022.

[37] "React router concepts." https://reactrouter.com/docs/en/v6/getting-started/concepts. Accessed: 01.05.2022.

[38] "Dom." https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. Accessed: 07.05.2022.

[39] "React hooks." https://reactjs.org/docs/hooks-overview.html/. Accessed: 07.05.2022.

[40] "Reacttable." https://react-table.tanstack.com/docs/examples/basic. Accessed: 07.05.2022.

[41] "Expo-constants." https://docs.expo.dev/versions/latest/sdk/constants/. Accessed: 05.05.2022.

[42] "Status-bar-image." https://www.educative.io/edpresso/how-to-add-a-status-bar-in-react-native. Accessed: 05.05.2022.

[43] "Styled-components." https://styled-components.com/docs. Accessed: 05.05.2022.

[44] "Formik." https://formik.org/docs/overview. Accessed: 05.05.2022.

[45] "Formik-submissions." https://formik.org/docs/guides/form-submission. Accessed: 05.05.2022.

[46] "Keyboard-type-email-image." https://infinitbility.com/how-to-set-keyboard-type-in-react-native. Accessed: 06.05.2022.

[47] "Activity-indicator-image." https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/activityindicator. Accessed: 06.05.2022.

[48] "Goole-oauth-2.0." https://support.google.com/cloud/answer/6158849?hl=en. Accessed: 07.05.2022.

[49] "Expo-google-authentication." https://docs.expo.dev/versions/v44.0.0/sdk/google/. Accessed: 07.05.2022.

[50] "React-native-navigation." https://reactnavigation.org/docs/getting-started/. Accessed: 07.05.2022.

Vedlegg A

Programlisting

A.1 LiveData.py

Code A.1: livedata.py Duplicate removal

```python
def clean_data(raw):
    data = []
    for line in raw:
        datapoint = []
        line = line.strip('\n')
        rawline = []
        for word in line.split('\t'):
            rawline.append(word)
        datapoint.append(rawline[2])
        datapoint.append(rawline[3])
        datapoint.append(rawline[4])
        data.append(datapoint)
    cleanedData = remove_duplicates(data)
    del cleanedData[0]
    return cleanedData
```

Code A.2: liveData.py Support functions

```python
def check_if_duplicate(data, newDatapoint):
    result = False
    for datapoint in data:
        if newDatapoint[0] == datapoint[0] and ...
            newDatapoint[1] == datapoint[1]:
            result = True
            print('Found a duplicate')
            break
    return result

def remove_duplicates(data):
    cleanedData = []
    for datapoint in data:
        if check_if_duplicate(cleanedData, datapoint):
            pass
        else:
            cleanedData.append(datapoint)
    return cleanedData
```

Code A.3: liveData.py Method for finding the limit

```
def find_limit(self):
        minrange = len(self.sample[0])
        for datatype in self.sample:
            length = len(datatype)
            if length < minrange:
                minrange = length
        return minrange
```

Code A.4: liveData.py Method for generating datapoints

```
def generate_datapoints(self):
        datapoints = []
        for i in range(len(self.sample)):
            datapoint = self.sample[i][self.counter]
            datapoints.append(datapoint)
        return datapoints
```

Code A.5: liveData.py Method for updating the counter

```
def update_counter(self):
        if self.counter == self.limit :
            self.counter = 0
        else:
            self.counter += 1
```

A.1 LiveData.py

Code A.6: liveData.py Method for pushing data

```
def push_datapoints(self,datapoints):
        response = requests.post(self.target, ...
            verify=False, json = {
        'rpm_3': float(datapoints[8][-1].replace(',','.')),
        'rpm_6':float(datapoints[9][-1].replace(',','.')),
        'rpm_100':float(datapoints[1][-1].replace(',','.')),
        'rpm_200':float(datapoints[2][-1].replace(',','.')),
        'rpm_300':float(datapoints[3][-1].replace(',','.')),
        'rpm_600':float(datapoints[0][-1].replace(',','.')),
        'rho':float(datapoints[4][-1].replace(',','.')),
        't_inlet':float(datapoints[5][-1].replace(',','.')),
        'yp':float(datapoints[6][-1].replace(',','.')),
        'pv':float(datapoints[7][-1].replace(',','.')),
        #'Date': date
        })
        print(datetime.now())
        if response.json()["msg"] == "ok":
            print("Successful")
        else:
            print("Unsuccessful call")
```

Code A.7: Switch for stats selection

```
def select_stats(conn,category,start,end):
    result = None
    if category == "all":
        result = select_all_stats(conn,start, end)
    elif category =="3_rpm":
        result = select_stats_3rpm(conn,category,start,end)
    elif category =="6_rpm":
        result = select_stats_6rpm(conn,category,start,end)
    elif category =="100_rpm":
        result = select_stats_100rpm(conn,category,start,end)
    elif category =="200_rpm":
        result = select_stats_200rpm(conn,category,start,end)
    elif category =="300_rpm":
        result = select_stats_300rpm(conn,category,start,end)
    elif category =="600_rpm":
        result = select_stats_600rpm(conn,category,start,end)
    elif category =="rho":
        result = select_stats_rho(conn,category,start,end)
    elif category =="t_inlet":
        result = select_stats_tinlet(conn,category,start,end)
    elif category =="pv":
        result = select_stats_pv(conn,category,start,end)
    elif category =="yp":
        result = select_stats_yp(conn,category,start,end)
    return result
```

Code A.8: Schema for database

```
sql_create_Rheosense_table = """ IF NOT EXISTS (SELECT * ...
    FROM sysobjects WHERE name='Rheosense' and xtype='U')
CREATE TABLE RheosenseData (
                                id INTEGER,
                                rpm_3 REAL,
                                rpm_6 REAL,
                                rpm_100 REAL,
                                rpm_200 REAL,
                                rpm_300 REAL,
                                rpm_600 REAL,
                                rho REAL,
                                t_inlet REAL,
                                yp REAL,
                                pv REAL,
                                Date datetime
                            );"""
```

A.2 Setup$_d b.py$

Code A.9: setup$_d b.pyUserstable$

```
def create_tables(conn):
    # Drop previous table of same name if one exists
    cursor = conn.cursor()
    cursor.execute("DROP TABLE IF EXISTS Users;")
    print("Finished dropping tables (if existed)")
    cursor.execute("""
        CREATE TABLE Users (
            id serial PRIMARY KEY,
            name TEXT NOT NULL,
            email TEXT UNIQUE NOT NULL,
            hashedpassword TEXT NOT NULL,
            role TEXT NOT NULL,
            permissions TEXT[],
            created TIMESTAMP

        );""")
    print("Finished creating table")
    conn.commit()
```

A.3 Helper Functions

Code A.10: CreateAxis.js Method for creating the axis

```
export  function CreateAxis(stats){
    console.log(stats)
    console.log(`creating axis for:${stats.name}`)
    const spread = stats.max-stats.average
    console.log(`spread is:${spread}`)
    if (spread<1){
        const start= ...
            parseFloat((stats.average-0.5).toFixed(1))
        const end=  parseFloat((stats.average+0.5).toFixed(1))
        const axis =[start, end]
        return axis
    }else{
        const start = Math.round(stats.average-spread*3)
        const end = Math.round(stats.average+spread*3)
        const axis =[start, end]
        return axis
    }
}
export default CreateAxis;
```

Code A.11: ModifyAxis.js Method for modifying the axis

```
export  function ModifyAxis(axis,direction,amount){
    console.log(`Shifting axis:${axis[0]},${axis[1]} ...
        direction:${direction} with a modifier of: ${amount}`)
    let axisLength = axis[1] - axis[0];
    function leftShift(axis,amount){
        axis[0]=parseFloat(
        (axis[0]-axisLength*amount).toFixed(1));
        axis[1]=parseFloat(
        (axis[1]-axisLength*amount).toFixed(1));
    };
    function rightShift(axis,amount){
        axis[0]=parseFloat(
        (axis[0]+axisLength*amount).toFixed(1));
        axis[1]=parseFloat(
        (axis[1]+axisLength*amount).toFixed(1));
    };
    function scaleAxis(axis,amount){
        axis[0]=parseFloat(
        (axis[0]-axisLength*amount/2).toFixed(1));
        axis[1]=parseFloat(
        (axis[1]+axisLength*amount/2).toFixed(1));
    };
    if (direction==="left"){
        leftShift(axis,amount)
    }else if(direction==="right"){
        rightShift(axis,amount);
    }else{
        scaleAxis(axis,amount)
    }
    console.log(`new axis is:${axis[0]},${axis[1]}`)
    return axis
}
export default ModifyAxis;
```

Code A.12: Method for determining if axis index is even

```
function isEven(index){
        if(index%2 ==0 ){
            return true
        }
        return false
```

Code A.13: Format.js - Code for formatting the date

```
import moment from 'moment';

export function Format(timeDelta, result) {

      if(timeDelta === 1800) {
        result.forEach(element => {
            console.log("formating date:")
            console.log(element.Date)
            element.Date = moment(element.Date).format("LT");
            console.log(element.Date)
        });
        return result
      }
      if(timeDelta === 30 || timeDelta === 120) {
          result.forEach(element => {
            element.Date = moment(element.Date).format("LTS");
        });
        return result
      } else {
        result.forEach(element => {
          element.Date = moment(element.Date).format("LTS");
        });
        return result
      }
}

export default Format;
```

A.4 Stats component

Code A.14: Stats.js - The stats component

```
export const Stats = (props) =>{
    const { stats } = props;
    const { categories } = props;
    const data = useMemo(() => ...
        CreateStatsData(stats,categories),[stats,categories]);

    const columns = useMemo(() =>[
        {
            Header:'Name',
            accessor:'name'
        },
        {
            Header:'Average',
            accessor:'average'
        },
        {
            Header:'Minimum',
            accessor:'minimum'
        },
        {
            Header:'Maximum',
            accessor:'maximum'
        }
        ],[]);

    const {
        getTableProps,
        getTableBodyProps,
        headerGroups,
        rows,
        prepareRow,
    } = useTable({
        columns,
        data
    });

    return(
    <div className="Stats-Table-Container">
        <table className="table table-striped" ...
            {...getTableProps()}>
         <thead >
            {headerGroups.map((headerGroup)=>(
                <tr {...headerGroup.getHeaderGroupProps()}>
                    {
                    headerGroup.headers.map( column =>(
                    <th {...column.getHeaderProps()}> ...
                        {column.render('Header')} </th>
                    ))}
                </tr>
                  ))}
        </thead>
        <tbody {...getTableBodyProps()}>
            {rows.map((row)=> {
                prepareRow(row)
                return (
                    <tr {...row.getRowProps()}>
                        {row.cells.map((cell) =>{
                         return  <td
                         {...cell.getCellProps()}>
                         {cell.render('Cell')}
                         </td>
                                })
                            }
                        </tr>
                    )}))}
            </tbody>
        </table>
    </div>
        );

        }
```

A.5 Page components

Code A.15: HistoricData.js - The historic Data page

```
const HistoricChart = () =>{
    const [time, setTime] = useState(14400);
    const [stats, setStats] = useState([]);
    const Δ = 14400;
    const [error, setError] = useState(null);
    const [isLoaded, setIsLoaded] = useState(false);
    const [StatsLoaded, setStatsLoaded] = useState(false);
    const [HistoricItems, setHistoricItems] = useState([]);
    const fetchInterval = 2000;

    const handleClick = (time) => {
        setTime(time)
      }




useEffect(() =>{
  fetch(`https://bachelor-demo-ims.azurewebsites.net/
  GetStats?collum=all&Δ=${encodeURIComponent(43200)})`,{mehtod: ...
      "GET",})
  .then(res => res.json())
  .then((result) =>{
        setStats(result)
        setStatsLoaded(true)
  })
}, []);




useEffect(() =>{
    fetch(`https://bachelor-demo-ims.azurewebsites.net/
    GetInterval?Δ=${encodeURIComponent(time)}`,{mehtod: ...
        "GET",})
    .then(res => res.json())
    .then((result) =>{
        setIsLoaded(true);
        let formattedResult = Format(time,result)
        setHistoricItems(formattedResult);
    })
  }, [ time]);
if(!isLoaded ){
    return(<>Loading....</>)
}else if(!StatsLoaded){
    return(<>Loading....</>)
}
else{
return(
<>
<div className="stats-container">
                <div className='category-container'>
                   <Stats stats = {stats} ...
                       categories={["3_rpm","6_rpm"]}/>
                   <div className='chart-combiner'>
                   <Chart items={HistoricItems} stats={stats} ...
                       isLoaded={isLoaded}
                   error={error} categories= ...
                       {["3_rpm","6_rpm"]} ...
                       colours={["green","purple"]} />
                         </div>
                    </div>
                    <div className='category-container'>
                       <Stats stats = {stats} ...
                           categories={["100_rpm","200_rpm"]}/>
                       <Chart items={HistoricItems} ...
                           stats={stats} isLoaded={isLoaded}
                       error={error} categories= ...
                           {["100_rpm","200_rpm"]} ...
                           colours={["orange","purple"]} />
                   </div>
                   <div className='category-container'>
                        <Stats stats = {stats} ...
                            categories={["300_rpm","600_rpm"]}/>
                       <div className='chart-combiner'>
                       <Chart items={HistoricItems} ...
                           stats={stats} isLoaded={isLoaded}
                       error={error} categories= ...
                           {["300_rpm","600_rpm"]} ...
                           colours={["blue","orange","purple"]} ...
                           />
                       </div>
                       </div>
                       <div className='category-container'>
                       <Stats stats = {stats} ...
                           categories={["yp","pv"]}/>
                       <div className='chart-combiner'>
                           <Chart items={HistoricItems} ...
                               stats={stats} isLoaded={isLoaded}
                           error={error} categories= ...
                               {["yp","pv"]} ...
                               colours={["green","red"]} />
```

```
                                </div>
                                </div>
                                <div className='category-container'>
                                <Stats stats = {stats} ...
                                    categories={["rho","t_inlet"]}/>
                                <div className='chart-combiner'>
                                <Chart items={HistoricItems} ...
                                    stats={stats} isLoaded={isLoaded}
                                error={error} categories= ...
                                    {["rho","t_inlet"]} ...
                                    colours={["blue","orange"]} />
                                    </div>

                                </div>
                            </div>
                            <button className='timeOption' ...
                                onClick={ () => handleClick(1800) ...
                                }> 30 Minutes </button>
                            <button className='timeOption' ...
                                onClick={ () => handleClick(3600) ...
                                }> 1 Hour </button>
                            <button className='timeOption' ...
                                onClick={ () => handleClick(14400) ...
                                }> 4 Hours </button>
                            <button className='timeOption' ...
                                onClick={ () => handleClick(43200) ...
                                }> 12 Hours </button>
                            <button className='timeOption' ...
                                onClick={ () => handleClick(86400) ...
                                }> 24 Hours </button>

</>)
}
}

export default HistoricChart;
```

Code A.16: LiveData.js - The live Data page

```
const LiveChart = () =>{
    const [time, setTime] = useState(1800);
    const [stats, setStats] = useState([]);
    const Δ = 1800;
    const [error, setError] = useState(null);
    const [isLoaded, setIsLoaded] = useState(false);
    const [liveitems, setLiveItems] = useState([]);
    const fetchInterval = 2000;


const fetchData =  async () => {
    let fetchDelta = time
    if(liveitems.length>0){
      fetchDelta = fetchInterval/1000;
    }
    console.log(fetchDelta);
  await fetch(`https://bachelor-demo-ims.azurewebsites.net/
  GetInterval?Δ=${encodeURIComponent(fetchDelta)}`,{
  method: "GET",
})
    .then(res => res.json())
    .then(
      (result) => {
        setIsLoaded(true);
        let formattedResult = Format(time, result);
        if(liveitems.length>0){
          let mergedResult = MergeResult(liveitems, ...
              formattedResult);
          setLiveItems(mergedResult);
        }
        else{
            setLiveItems(result);
        }

      },
      (error) => {
        setIsLoaded(true);
        setError(error);
      }
    )
}

useEffect(() =>{
  fetch(`https://bachelor-demo-ims.azurewebsites.net/
  GetStats?collum=all&Δ=${encodeURIComponent(43200)})`,{mehtod: ...
      "GET",})
  .then(res => res.json())
  .then((data) =>{
      setStats(data);
  })
}, []);

const handleClick = (time) => {
  setTime(time)
  setLiveItems([])

}
```

```

useEffect(() => {
  const interval = setInterval(async () => {

    await fetchData();
  }, fetchInterval);
  return () => {
    clearInterval(interval);
  }
}, [liveitems], time);

return(
<>
  <div className="panel-container">
      <div className='panel panel-primary'>
        <Stats stats = {stats} ...
            categories={["3_rpm","6_rpm"]}/>
        <div className='panel-body'>
          <Chart items = {liveitems} isLoaded = {isLoaded}
          error = {error} stats = { stats } ...
              categories={["3_rpm","6_rpm"]} colours = ...
              {['red','blue']}/>
        </div>
      </div>
      <div className='panel panel-primary'>
        <Stats stats = {stats} ...
            categories={["100_rpm","200_rpm"]}/>
        <div className='panel-body'>
        <Chart items = {liveitems} isLoaded = {isLoaded}
        error = {error} stats = { stats } categories = ...
            {["100_rpm","200_rpm"]} colours = ...
            {['green','orange','blue']} />
        </div>
      </div>
      <div className='panel panel-primary'>
        <Stats stats = {stats} ...
            categories={["300_rpm","600_rpm"]}/>
        <div className='panel-body'>
        <Chart items = {liveitems} isLoaded = {isLoaded}
        error = {error} stats = { stats } ...
            categories={["300_rpm","600_rpm"]} colours = ...
            {['green',"purple"]}/>
        </div>
      </div>
      <div className='panel panel-primary'>
        <Stats stats = {stats} categories={["yp","pv"]}/>
        <div className='panel-body'>
        <Chart items = {liveitems} isLoaded = {isLoaded}
        error = {error} stats = { stats } ...
```

```
               categories={["yp","pv"]} colours = ...
               {['blue','orange']}/>
            </div>
          </div>
          <div className='panel panel-primary'>
            <Stats stats = {stats} ...
                categories={["rho","t_inlet"]}/>
            <div className='panel-body'>
            <Chart items = {liveitems} isLoaded = {isLoaded}
            error = {error} stats = { stats } ...
                categories={["rho","t_inlet"]} colours = ...
                {['green','red']} />
            </div>
          </div>
      </div>
      <div>
        <button className='timeOption' onClick={ () => ...
            handleClick(1800) }> 30 Minutes </button>
        <button className='timeOption' onClick={ () => ...
            handleClick(3600) }> 1 Hour </button>
        <button className='timeOption' onClick={ () => ...
            handleClick(14400) }> 4 Hours </button>
        <button className='timeOption' onClick={ () => ...
            handleClick(43200) }> 12 Hours </button>
      </div>
    </>)
}

export default LiveChart;
```