Presentation

Requirements

MARTE

In the section Performance Requirement Modelling with MARTE we have created an activity diagram that is based on how we think the system will run. In the MARTE we have added estimated mean values of what each part of the systems service time is, then we have the response time for the whole system which we require to be at maximum 10 seconds. We have also estimated the probabilities whenever a user can make a choice.

We believe that a user visits our web server and then decides whether they want to sign up/login or exit the web server. Then when the user has logged in they either log in as an admin or user and depending on which they log in as they will have different functionalities. While logged in the user and admin will work with the database and then when they feel done they will log out and then exit the application.

UML requirements

This is a sample Activity Diagram of the AUTH-USER Requirement from our system.

The requirement description is as follows:

- “A user shall be the only one that can access their personal profile.”

AUTH-USER

In this sample activity diagram, we have a component called SignUp which contains a page with a SignUp button, which requests to open a pop-up window for google authentication.

Once the user has input their data, the values are collected from google and sent directly to our database where it is saved under a collection “ Users”. We do not store the password inside of our database, this is handled by Google authentication services. The reason for that is strictly for security purposes.

Once a user wants to login again they are connected to the database where the Google authentication once again will verify that the user has an account and if the user doesn’t exist in our database they are automatically added through the system.

Since we have stored the data inside Firebase we can fetch specific values and check for the identification of the user. If a user is logged in they will return their unique identification value, otherwise, they will return “null” or “undefined”. This value can then be checked in PrivateRoute.js to see if the user is logged in or not, and from there decide which component to display in the navigation bar or menu.

The same method can be used inside of other components to Redirect the user if they manually try to enter an URL in the address bar.

Performance modeling

In performance modeling, we have based the model of the system on the activity diagram that we created on D1 and we input the service time values that we estimated in that diagram as well. After that, we input the probabilities that were created in D1. Then we decided what performance indices we wanted to know for the system and chose the system's response time, each individual part of the system's throughput and utilization. We want these values to be able to see that we have a response time that is less than the required time we set at D1. We want to know how effective our system will be and how used each part of the system will be.
Finally, we run a simulation and find out the output on the performance indices. We got decent values, but they could have been better overall.

Results, response time around one second, decent value. Throughput 0.3-0.7 product/second not ideal would like a higher production rate, at the very least over one product/second. Utilization Authentication and UserAppServer used most of the system parts.

Design issue

We've encountered many design issues, and these are the two main ones.

The first issue we had was what kind of software design pattern we would choose. Our first thought was to implement Model, View, and controller (also called MVC). The reasoning behind our choice was purely based on our familiarity with the pattern. Very late on in the project, we realised that the MVC pattern was not suitable for the React framework. The most compatible design pattern for React is Flux. The discovery of this issue was found in the class hierarchy of our project. We couldn't fully apply the MVC pattern into our project since classes and interfaces needed to be in different places. It was complicated to match our original MVC design pattern into the project. We solved the issue by applying Flux, but since the discovery was found very late on in the project, we couldn't fully implement everything in our project.

The second issue we had was regarding the implementation of habits and categories into our code. We had some struggles to store the data from habits and categories in the right place since we wanted to give the user access to their individual habits and categories. The problem was solved by linking the created habit and categories to the user's id tag. This way, a user is given full access to them.

Software architecture

The software architecture we chose to base our project on is the client-server architecture. The reasoning behind this choice is since we are hosting a series of users we thought that this architecture would make the most sense.

We chose to go with the library called react and therefore went with their own model, which is called flux. We decided to go with flux since it is the best way of building client-side web applications with react since it complements reacts composable view. And that fitted perfectly with our vision of the project.

Components

This is the current state of our system architecture presented in a components diagram. It shows the correspondence between our components and how they talk with each other.

As an example of how our system works using FLUX as our design pattern would be “CreateHabit”. Say, for example, you want to work out for 30 minutes every other day, then you can submit this information inside the form, and this component will send that data directly to firebase as an action.

Firebase will then take this action data and store it under the specific user’s unique id value (*which we briefly explained previously in the presentation*) and store this new habit data in the collection “habits”.