AUT.840-2022-2023-1
Industrial Informatics
Assignment 2

Jukka Hirvonen [H218618],

Valtteri Nikkanen [282688],

Jouni Kokkonen [151091078]

Table of contents

Link

https://green-joulutorttu-321.azurewebsites.net/dashboard/?nID=

Introduction

This assignment was divided into two separate tasks. Task 1 was to produce a simple python application that received REST messages, reformatted them and forwarded the information to a topic in MQTT form.

The purpose of Task 2 was to design a cloud-deployed online SCADA system capable of saving information from MQTT messages received with a subscription into a database and deducing equipment states from the data. The information was to be displayed in a webpage UI format using HTML and JavaScript.

The design phase was implemented by creating UML diagrams. Based on the diagrams, programs were created. The task 1 program was first created with the Postman REST Client, then by running one instance as the subscriber and one to receive the subscribe messages and finally live in the FASTORY Lab. In each test the forwarded messages were verified using the web interface of the HiveMQTT web UI.

Task 2 UI was first tested with a dummy python app with the Flask library to ensure basic connectivity. Then, the whole task, including the Python backend, was tested live in the web browser using actual MQTT event feed from the assignment broker.

After testing bugs and problems were located and fixed to find reliable solution. And once the front- and backends were connected, without issues and the whole SCADA system was working it was uploaded to Azure cloud and started.

Functionality/Idea of the code

For Task 2 the idea was to read data from a database and show the data on a webpage. As the assignment was refined only a dashboard presenting the current state and a historical data which would present all the data were created for the webpage. The backend receives new information from the robots via MQTT messages which it saves into a database. The frontend then requests the correct data from the database via HTTP request which it then presents in the HTML page.

The dashboard updates automatically the current selected robot's status on the page. We ended up with a user interface where robots are selected by pressing a button, and dates are

selected using a selector form. On the Dashboard page, the State text changes colour according to the state of the robot. ("READY-PROCESSING-EXECUTING" = green, "READY-IDLE-STARVED" = yellow and otherwise red). Also, the state of the robot is shown as is when the latest message from the robot was received.

The historical data however does not update itself and only refreshes the data when the page is loaded by selecting a new/the same robot again. All the received messages for a robot can be viewed by just selecting a robot. The page normally loads all the received messages for the chosen robot. This historical data can also be viewed while filtering it by a date range when the messages were received. Both dates need to be selected and there needs to be a robot selected for the filtering to work.

The program only shows historical data by listing it all underneath each other. Originally some KPI parameters were supposed to be calculated but the assignment was lightened, and they were no longer necessary, so these were not implemented. The same goes with the alarms which also were excluded from the assignment.

For Task 2 multiple python libraries were used. The Flask library was used for communicating between the back- and front ends. Threading was used for starting the program in the cloud. Paho-mqtt library was used for subscribing to MQTT messages from industrial equipment. Sqlite3 was used for the database in the backend. JSON library was used for reading the MQTT messages to the database. And finally, datetime library was used for parsing the timestamp in the MQTT messages.

Use Case Diagrams

Use case diagrams that are shown in Figure 1 and Figure 2 were created in the design phase of the project and still hold true after the project was completed. We found two possible actors and the possible use cases for them.

Figure 1 Use case diagram for Task 1

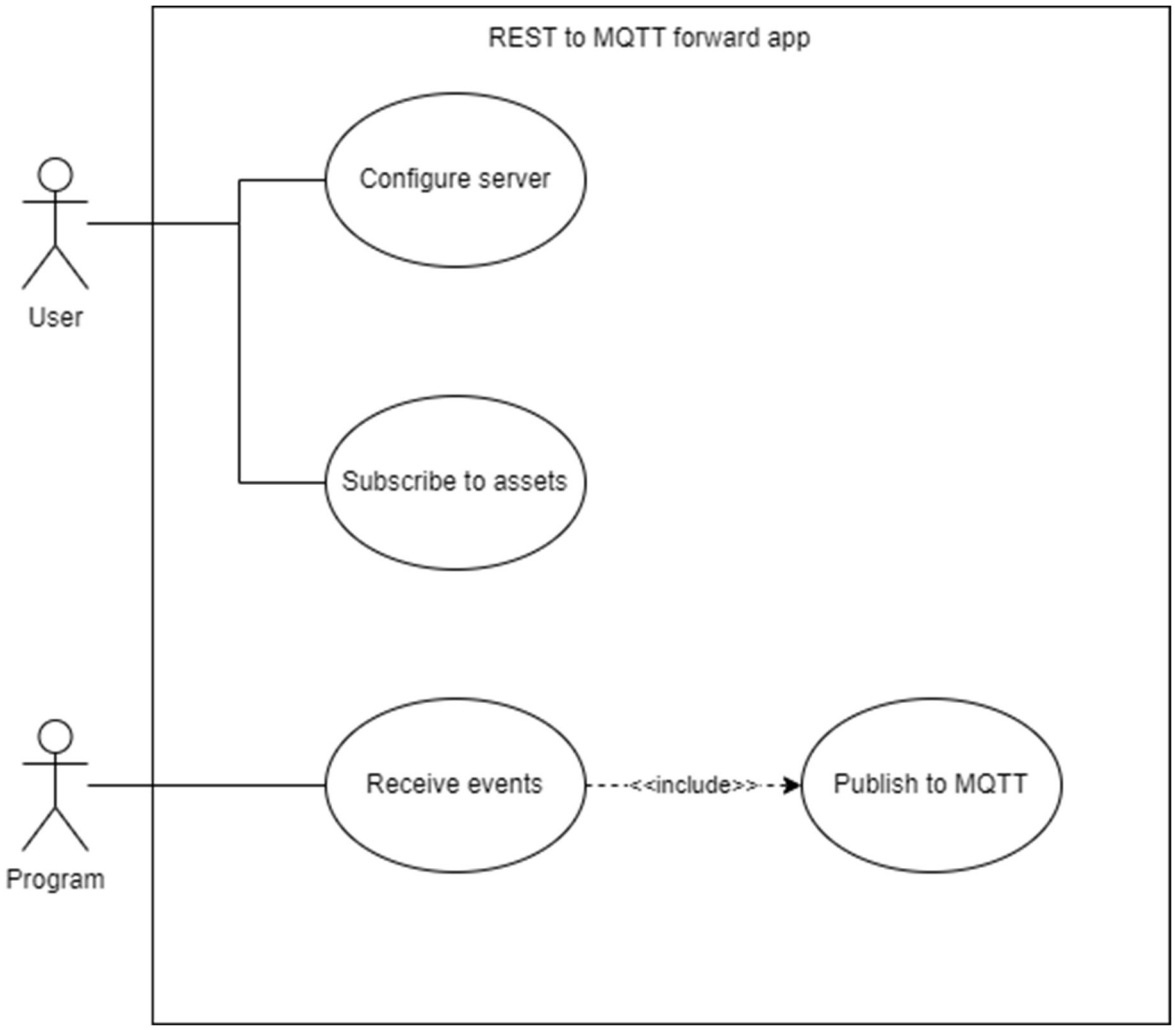
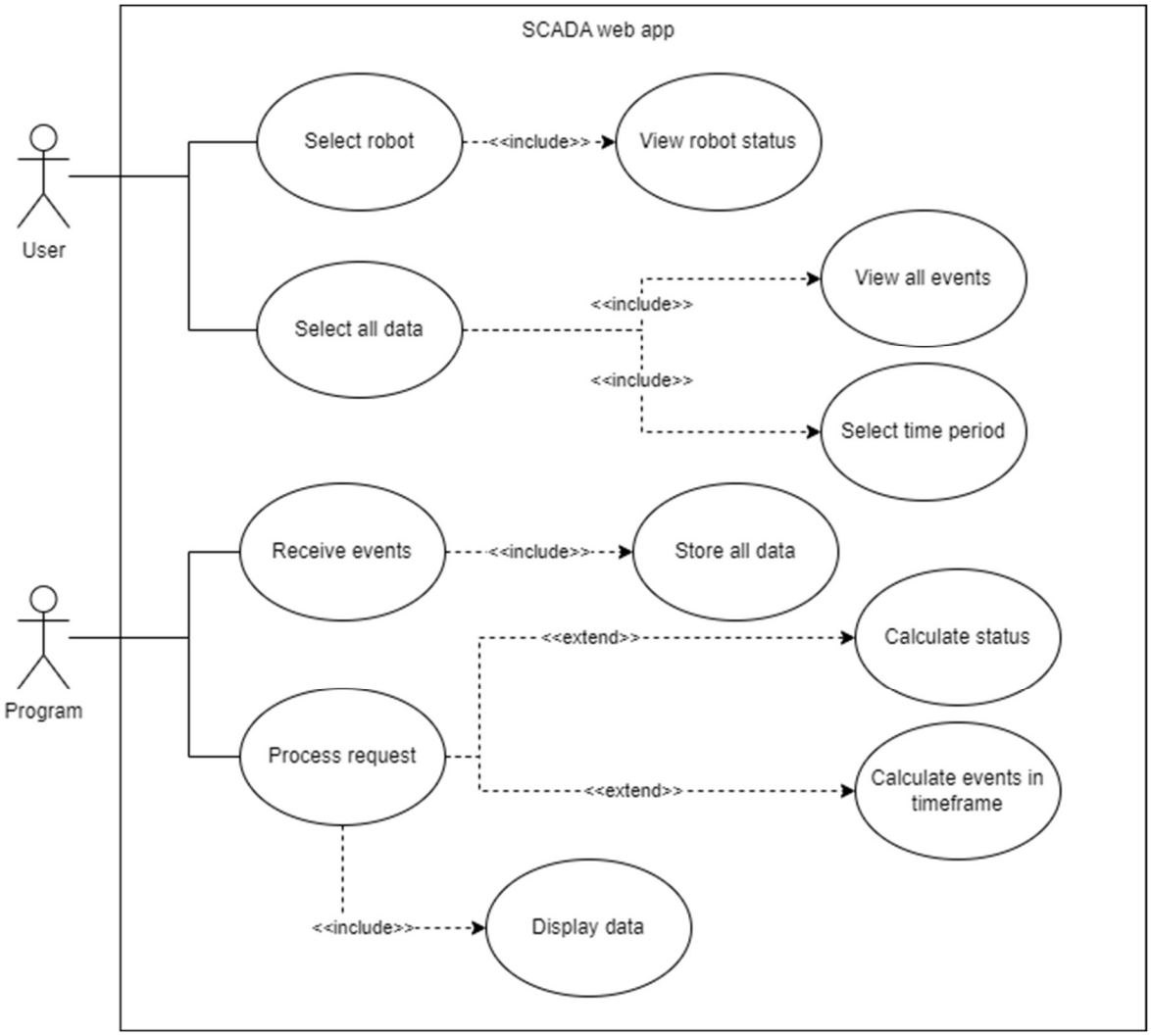


Figure 2 Use case diagram for Task 2

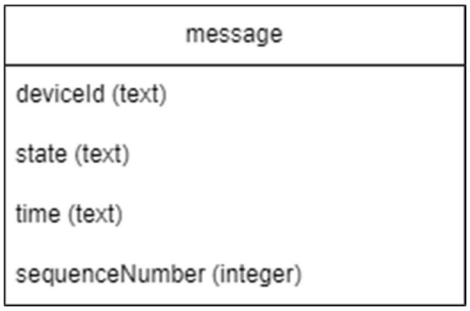


The finished diagrams were not only useful in the planning phase, but also effective at visually describing all the features of the program.

Database Diagram

Due to the simplicity of the program, a single table was sufficient to provide the functionality and performance required. In a more complex project, it would be wise to weigh the frequency of read and write events to decide how many tables to split the database into.

Figure 3 Database entity diagram for task 2



The resulting solution is elegant in its simplicity. The database was kept in memory rather than a file, so it would reset whenever the application was relaunched.

Sequence Diagrams

Figure 4 shows the process of finding and displaying a robot's current status through the system. The state is refreshed automatically for as long as the UI is open in the web browser.

Figure 3 Sequence diagram for displaying a robot's current status

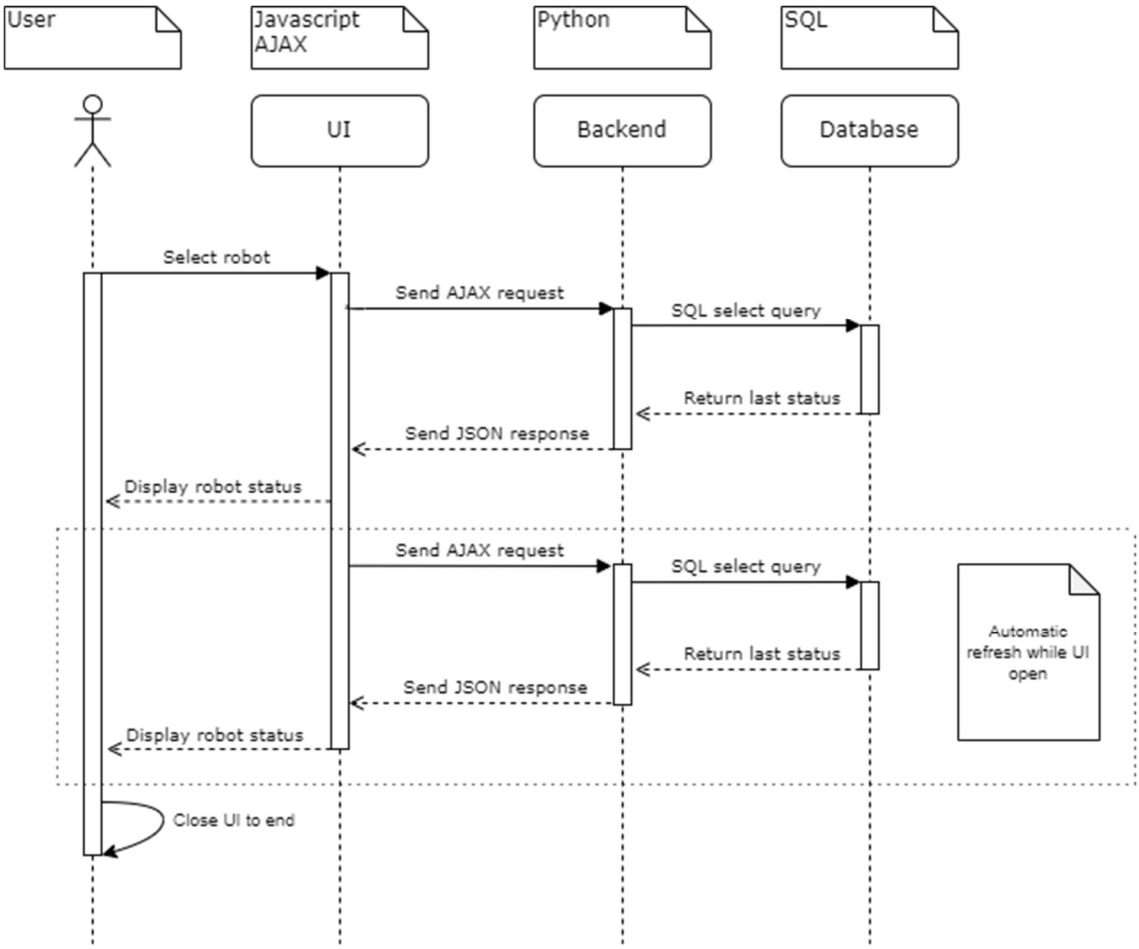
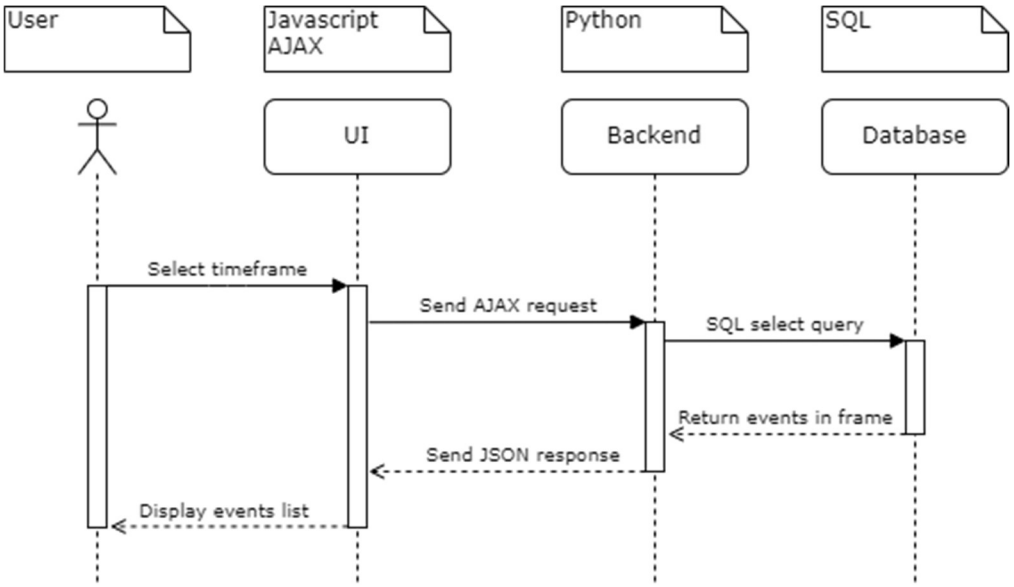


Figure 5 shows how the list of events occurring in a certain timeframe is found in the program. The user selects the timeframe in the UI, which contacts the backend with an AJAX request, which finds the answer in the database via an SQL query.

Figure 4 Sequence diagram for displaying a list of events in selected timeframe



The sequence diagram format was well suited to describing the main use cases of the task and effectively represents the way messages and responses pass through the different levels of the program.

Encountered Problems & Solutions

Several problems were encountered in the implementation of the user interface. There were challenges in designing functional Ajax get requests and the JavaScript was modified several times. Various selectors were tried, but finally we ended up with a button solution when choosing a robot. During the implementation phase, someone posted their own JSON messages to the same endpoint url that caused backend crashes. The problem was fixed by modifying the backend.

Some small problems with SQLite queries were encountered especially with getting the entries between the dates. However, once the AND keyword was figured out correctly the problems were quickly solved.

Getting the correct datetime format for the dates also took some figuring out. But once it was figured out that datetime library can’t comprehend nanoseconds and the last 3 digits were cut off the end of the time and so the nanoseconds were converted to microseconds everything started working correctly.

Result

Task 1 result was a single offline python file. In addition to formatting and passing the subscribed messages to the MQTT topic, it also provides a text-based menu for configuring the program ip/port and subscribing to assets.

We implemented the task 2 solution by using MVC-pattern. We made the user interface “dummy” that doesn't make choices and all decisions take place in the backend. With the simplification of the assignment thanks to some technical difficulties the final solution is simple, and the backend doesn’t need to do a lot of work aside from reading and sending data from the database.