PROJECT ONE

Week In My Life

Abstract

This document provides a glimpse into my day-to-day experience, tasks, and contributions during a typical work week as part of my apprenticeship

Alister David

alisterdavid03@gmail.com

Contents

Introduction:

My name is Alister, I am a software engineer apprentice at IQHQ. I have been working at IQHQ since the 6th of November 2023. This presentation provides a glimpse into my day-to-day experience, tasks, and contributions during a typical work week as part of my apprenticeship.

The Company:

IQHQ,
Defence & Security

IQHQ is one of the few waveform development companies within Europe. We specialise in supporting the communications, intelligence, surveillance & security markets by developing bespoke datalink and communication waveforms on custom-designed hardware. On top of implementing communication waveforms, IQHQ also tests and develops them rigorously. We leverage cutting-edge research and techniques, offering virtually limitless control and configuration, resulting in a true software defined radio or datalink when paired with the available hardware packages.

IQHQ breaks limitations of application specific integrated circuits by allowing the controlled configuration of every function within the firmware and hardware of their technology. This provides optimised profiles, resilient to weaknesses in each use case scenario. When considering interoperability, it also offers connectivity that provides the information needed, where needed and when needed under a variety of stressed link environments.

My Job Role:

As a Software Engineer Apprentice at IQHQ, I directly contribute to the dynamic software development team. I will be working on improving and fixing existing technology as well as adding new features. Software is an integral part of the company since it is used on every product, as well as the various systems to test and output the radio communication systems. I am actively involved in hands-on experiences with using languages such as JavaScript, TypeScript, Node.js, React & MUI for the UI. My learning experience extends to database management, specifically PostgreSQL for data persistence. In alignment with the broader project lifecycle at IQHQ, my responsibilities involve collaborating with cross-functional teams to

*Figure SEQ Figure * ARABIC 1 - Team Structure*

translate project requirements into actionable tasks, ensuring the quality, performance, and reliability of our software solutions. Through active participation in design discussions, coding, testing, and debugging processes, I contribute to delivering innovative software solutions that meet the needs of our customers and drive the success of our organisation.

Software Development Life Cycle:

Stages and Inputs/Outputs of the Software Development Life Cycle (SDLC):

Feasibility Study:

In the feasibility study stage, the project's feasibility is assessed in terms of technical, operational, and economic aspects.

Inputs: Inputs include project proposals, initial requirements, and market analysis.

Outputs: The main output is a feasibility report that determines whether the project is viable and worth pursuing.

Requirement Analysis:

Requirement analysis involves gathering, documenting, and validating the functional and non-functional requirements of the software.

Inputs: Inputs include stakeholder interviews, user surveys, and existing system documentation.

Outputs: The output is a detailed software requirements specification document (SRS) that serves as the basis for the design and development phases.

Design:

Design involves creating the architectural and detailed designs of the software based on the requirements identified in the previous phase.

Inputs: Inputs include the requirements specification document, design principles, and best practices.

Outputs: Outputs include high-level architectural designs, detailed design specifications, database schemas, and user interface mock-ups.

Code Development:

Code development is the implementation of the software based on the designs created in the previous phase.

Inputs: Inputs include the design documents, programming languages, and coding standards.

Outputs: The primary output is the executable code or software product, along with unit test cases and documentation.

Testing:

Testing involves verifying and validating the functionality, performance, and reliability of the software.

Inputs: Inputs include the software product, test plans, and test cases.

Outputs: Outputs include test reports, defect logs, and updated documentation. The goal is to ensure that the software meets the specified requirements and quality standards.

Deployment:

Deployment is the process of releasing the software to the production environment for end-users to access and utilise.

Inputs: Inputs include the tested and approved software product, deployment plans, and user training materials.

Outputs: Outputs include deployed software instances, configuration settings, user manuals, and training records.

Maintenance:

Maintenance involves ongoing support and enhancement of the deployed software to address issues and incorporate new features.

Inputs: Inputs include user feedback, bug reports, and change requests.

Outputs: Outputs include software updates, patches, bug fixes, and documentation updates. The goal is to ensure the continued functionality, security, and usability of the software over time.

Software Development Methodologies:

Agile:

Agile is a form of an iterative development method, which is a way of breaking down the software development life cycle of a large application into small chunks. In iterative development, feature code is designed, developed & tested in repeated cycles.

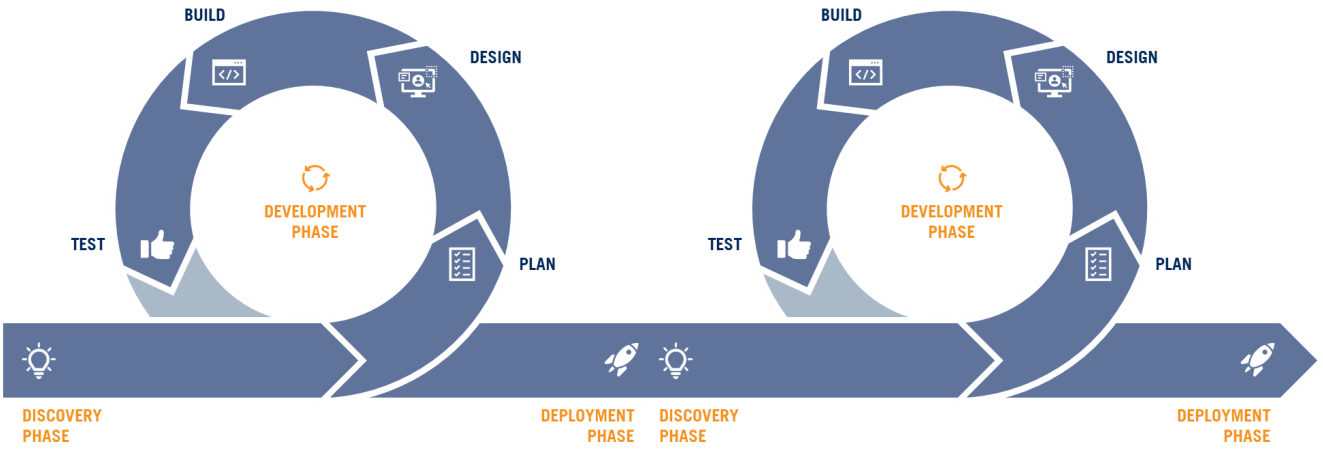


*Figure SEQ Figure * ARABIC 2 - Agile Model*

Agile allows for more flexibility when it comes to making changes. It is flexible, fast and aims for continuous improvements in quality. The agile methodology is a people focused, results focused approach to software development in a dynamic world. It is centred around adoptive planning, self-organisation & short delivery times. Some examples of Agile models include Scrum, Kanban, and XP.

Agile Manifesto:

- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.
- Individuals & interactions over process and tools.

Advantages of Agile:

Agile is a realistic approach for software development because it doesn't overload the project team with unrealistic demands. It enables rapid development and testing of functionalities, saving time and resources. Whether the requirements are fixed or

subject to change, Agile adapts well, allowing for early delivery of partially working solutions. With minimal planning required, Agile is easy to manage and provides flexibility to developers to adjust as needed. Overall, Agile streamlines the development process, making it efficient and practical for teams to deliver high-quality software products.

Disadvantages of Agile:

Agile, while advantageous in many respects, presents certain drawbacks. It may not handle complex dependencies well and might face difficulties in projects with many interconnected parts. This can lead to challenges in managing and coordinating various aspects of the development process. Moreover, there's a higher risk concerning the sustainability, maintainability, and extensibility of the software created under Agile methodologies. To overcome these challenges effectively, a detailed overall plan and strong Agile leadership become crucial. Additionally, strict delivery management is necessary to ensure that projects stay on track and meet deadlines. Furthermore, Agile heavily relies on customer interaction, which, while beneficial for understanding needs, can also introduce uncertainties and delays. Lastly, the approach often lacks thorough documentation, which can slow down knowledge transfer and future development efforts.

Waterfall:

Waterfall is a step-by-step approach where tasks follow a linear order. Each stage must finish before the next one begins. It's known for being well-documented with clear requirements that don't change much. The technology used is stable and doesn't change often. Waterfall projects usually have a fixed plan and use plenty of resources. They also tend to be shorter in duration.

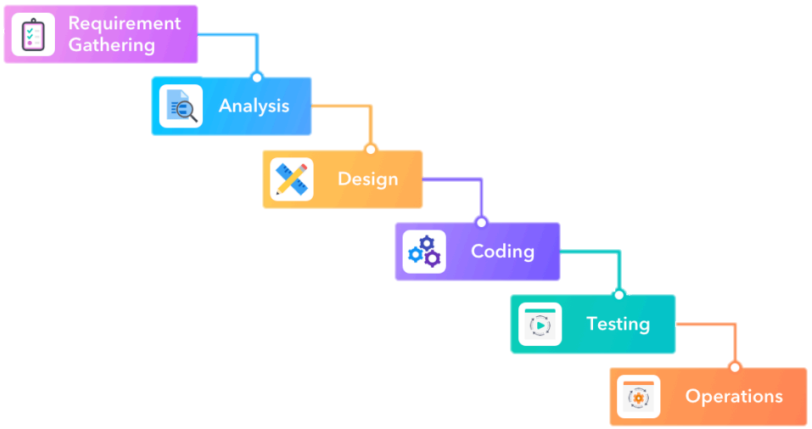


Figure 3 - Waterfall Model

Advantages of Waterfall:

Waterfall methodology offers several advantages, including the ability to departmentalise tasks and maintain control over the project's progression. By setting a schedule with deadlines for each stage, teams can effectively manage their time and resources. The strict sequential order of phases simplifies the process, making it easy to understand and implement, especially for smaller projects. Additionally, the well-defined milestones provide clear checkpoints for progress evaluation. This structure makes arranging tasks straightforward, ensuring a systematic and organised approach to project development. Overall, the Waterfall methodology facilitates efficient project management through its clear structure and manageable stages.

Disadvantages of Waterfall:

Waterfall methodology also presents several disadvantages. Firstly, because it follows a sequential approach, no working software is produced until late in the cycle, which can lead to delayed feedback and potential issues with meeting project objectives. Additionally, the high amount of risk and uncertainty is inherent in Waterfall due to the lack of flexibility to adapt to changing requirements or unforeseen challenges. There's limited room for reflection or revision throughout the process, making it challenging to address issues as they arise. Moreover, Waterfall is not well-suited for complex or object-oriented projects, nor for long-term or ongoing

development models, as it struggles to accommodate dynamic requirements and changes over time. Progress measurement within each stage can be difficult.

Agile & Waterfall Comparison:

Aspect	Agile	Waterfall
Approach	Iterative and flexible	Sequential and rigid
Feedback	Early and continuous	Late and limited
Risk Management	Embraces change and adapts easily	High risk due to inflexible structure
Requirements	Dynamic and evolving	Fixed and established
Project Control	Collaborative and decentralised	Centralised and departmentalised
Documentation	Minimal, focuses on working software	Comprehensive, emphasises documentation
Progress Measurement	Iterative, with regular checkpoints	Difficult, mainly at the completion of each stage
Suitable Projects	Well-suited for dynamic and complex projects	Better for simpler, well-defined projects
Integration	Continuous integration throughout the project	Integration at the end of the project
Flexibility	Offers flexibility to accommodate changes	Limited flexibility, changes may be difficult to implement
Communication	Regular communication and collaboration	Less emphasis on continuous collaboration
Time and Resources	Efficient use of resources, shorter timelines	Uses ample resources, longer timelines
Adaptability	Adapts well to changing requirements	Limited adaptability to changes during development

Figure 4 - Agile & Waterfall Comparison Table

Software Development Methodology At IQHQ:

At IQHQ, choosing the right software development methodology is an important part of completing our projects. Since each project tends to vary, we must adapt our software development methodology to accommodate the unique requirements and challenges of each project. While Agile methodologies such as Scrum and Kanban offer flexibility and adaptability to changing requirements, the Waterfall methodology may be preferred for projects with well-defined and stable requirements.

For example, in our research and development projects, where the requirements may constantly change, we often adopt Agile methodologies to accommodate iterative development cycles and frequent feedback from stakeholders. On the other hand, for production-focused projects with clearly defined requirements and timelines, we turn to the Waterfall methodology to ensure we reach our milestones and meet the deadline.

By carefully evaluating the similarities and differences between different software development methodologies, we can tailor our approach to suit the specific needs of each project, allowing us to be as efficient as possible and deliver on time.

The Roles Within IQHQ:

At IQHQ, our software development life cycle is driven by a diverse team of professionals, each playing a crucial role in delivering innovative solutions to our clients. From project management to technical leadership and software engineering, we have various roles that contribute to every stage of our development process.

Managing Director:

The Managing Director provides strategic direction and leadership for the organisation. They set overall goals and priorities, ensuring the effective operation of the company.

Project Manager:

The Project Manager oversees the planning, execution, and delivery of software development projects. They define project scope, allocate resources, manage budgets, and communicate project status to stakeholders.

Chief Engineer:

The Chief Engineer provides technical leadership and guidance for software development projects. They define technical standards and best practices, conduct design reviews, and mentor junior engineers.

Software Team Leader:

The Software Team Leader manages and coordinates the efforts of the software development team. They assign tasks, monitor progress, resolve conflicts, and provide technical guidance and support to team members.

Software Engineers:

Software Engineers design, code, test, and debug software solutions. They translate project requirements into software implementations, ensuring the quality, performance, and reliability of the final product.

Firmware Team Lead:

The Firmware Team Lead oversees the development of firmware solutions for hardware products. They define firmware requirements, architect firmware solutions, and coordinate firmware development efforts with other teams.

Hardware Engineers:

Hardware Engineers design and develop hardware solutions for products. They collaborate closely with software engineers to ensure compatibility and interoperability between hardware and software components.

Production Team:

The Production Team assembles and tests products. While not directly involved in software development, they ensure the quality and reliability of the final product through rigorous testing and quality control measures.

Quality Assurance Team:

Our QA team is responsible for ensuring the quality and reliability of our software solutions. They develop and execute test plans, identify and report defects, and collaborate with software engineers to resolve issues promptly. By maintaining rigorous testing standards, they help uphold our commitment to delivering robust and error-free software products.

Task 1: Adding Export Functionality to the Dashboard

Situation:

Throughout Monday and Tuesday of the week, my primary task was to implement an export functionality on the dashboard interface at IQHQ. This feature aimed to enable the export of database logs to customers post-demonstrations, enhancing our product's utility and customer satisfaction.

Task:

My objective was to integrate an export button into the dashboard UI, allowing users to export database logs to their devices.

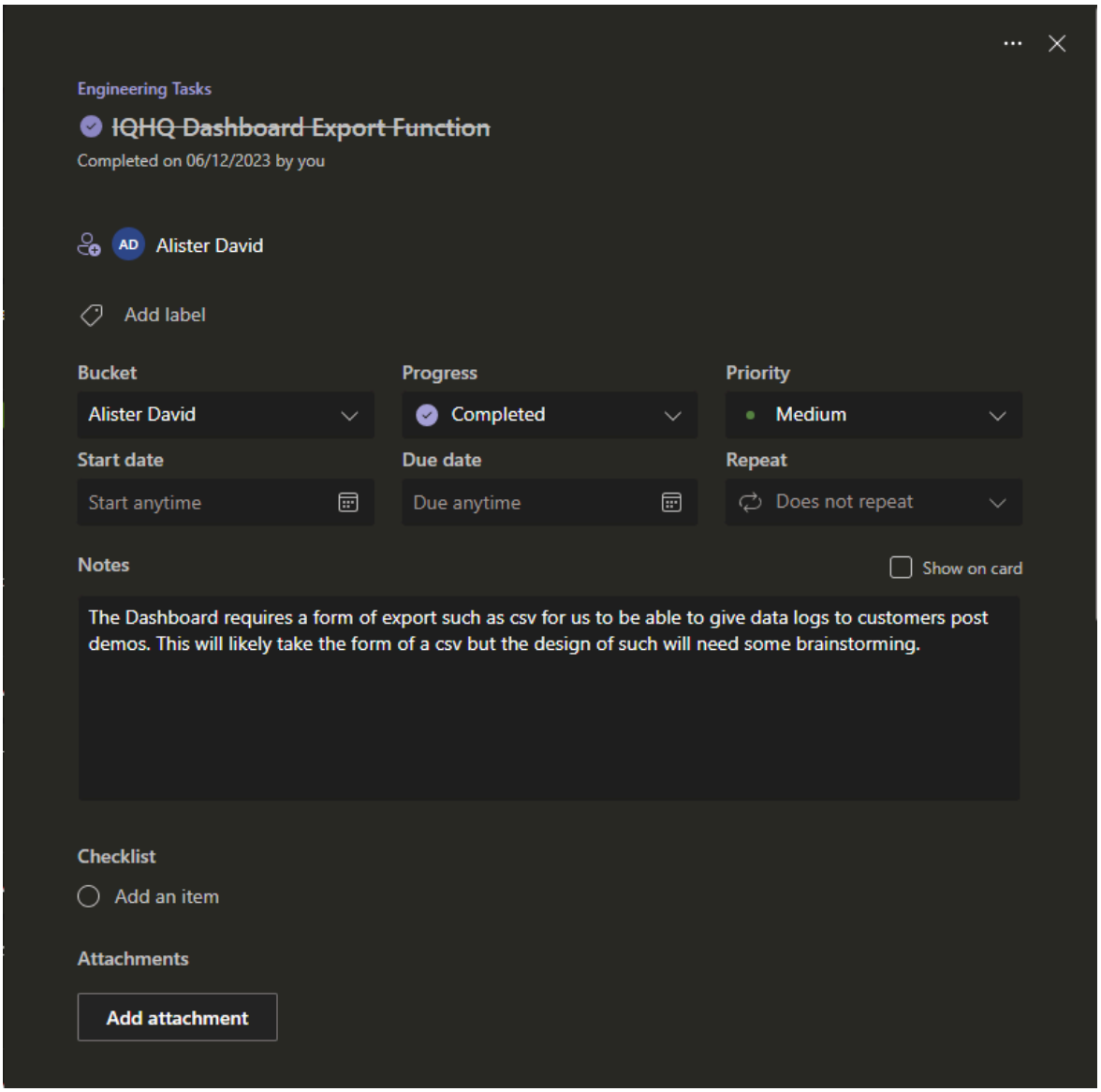


Figure 5 - Dashboard Export Function Task on Teams

Database Planning:

Before I began this task, it was crucial that I had a good understanding of the database structure and management. At IQHQ, we use PostgreSQL, a relational database management system (RDBMS), known for its robustness and flexibility. Relational databases organise data into tables, each with a predefined schema, and use structured query language (SQL) for data manipulation and retrieval. In contrast, non-relational databases, like MongoDB or Cassandra, store data in a more flexible format, such as JSON documents or key-value pairs. Understanding the differences between relational and non-relational databases is essential for choosing the right database solution for specific project requirements. If I needed to use an SQL database instead of PostgreSQL, I would have followed similar steps but adjusted the SQL queries accordingly.

Action:

My first step was to find out how to export the database using Docker. After identifying the relevant image name with the command ‘docker ps’, I researched and found the right command to download the database. I utilised the child_process module from Node.js to add a file called exportFunction, containing code to pg_dumpall the database. Simultaneously, I integrated an export button into the dashboard using React components, although it had no functionality initially, logging only ‘Button Clicked’.

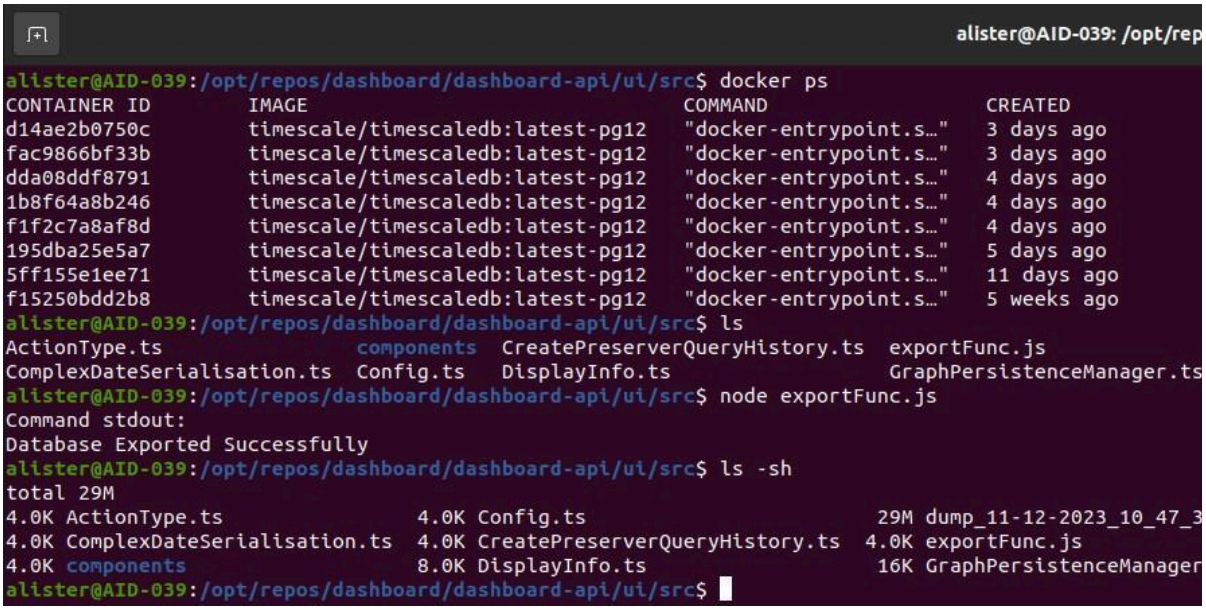


Figure 6 - Terminal running docker ps.

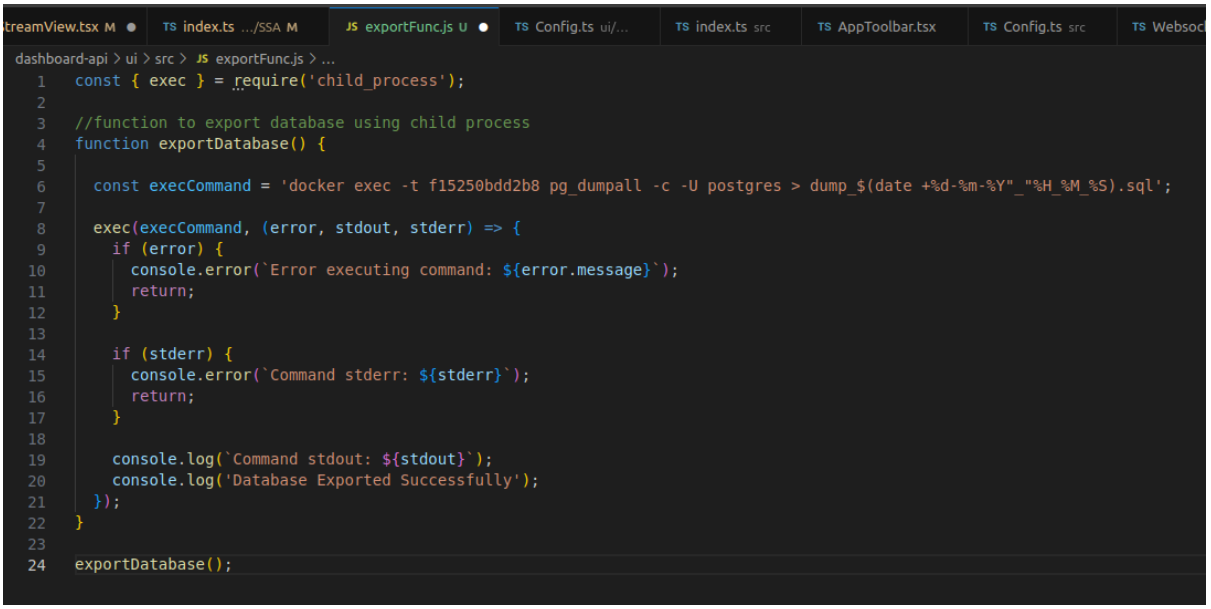


Figure 7 - exportDatabase Function.

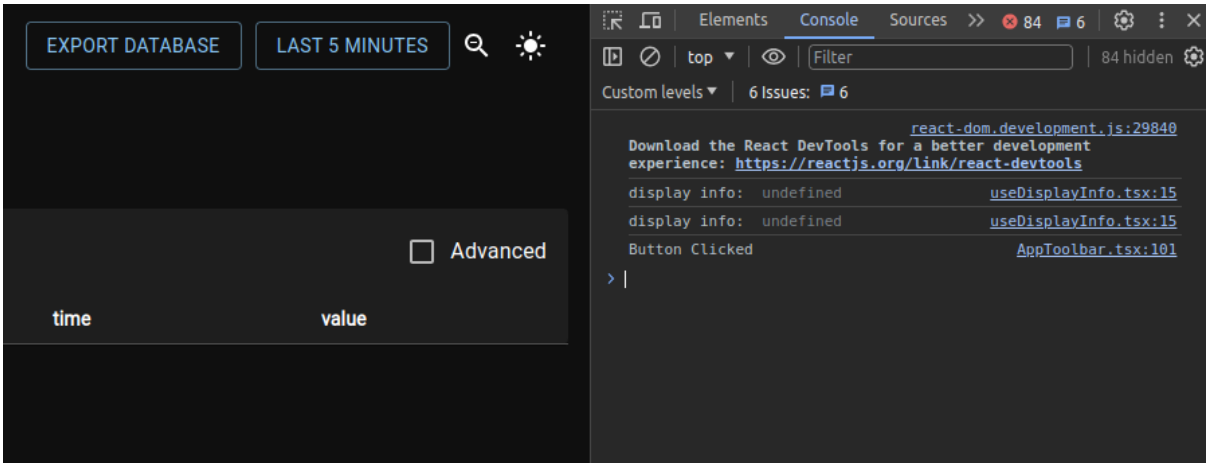


Figure 8 - React button logging 'Button Clicked'.

```
return (
    <Toolbar>
        <Grid container>
            {!props.isSidebarOpen
                && <Grid item><IconButton onClick={props.onSidebarOpen}>
                    <Menu />
                </IconButton></Grid>
            }
            <div style={{
                flexGrow: 1,
            }}/>

            <Grid style={{alignSelf: "center"}}>
                <HTTPTriggerButton
                variant="outlined"
                style={{marginRight: 10}}
                key={"Export"}
                method="post"
                path={"/export"}
                onComplete={(ok) => {
                    console.log('Button Clicked');
                }}
                >
                EXPORT DATABASE
                </HTTPTriggerButton>
            </Grid>
```

Figure 9 - Code for the react export button.

Issues arose when attempting to export the database, encountering errors while viewing the PostgreSQL table in the terminal. Additionally, the child_process module functioned exclusively on the server-side, hindering front-end execution. I realised each terminal command run created a new Docker container, leading to multiple restarting containers. Therefore, I need to implement some sort of filter in my exportFunction file that will only use the running container ID. To resolve these challenges, I focused on shifting the export function to the backend, collaborating with the software team to establish frontend-backend communication.

By the end of Monday, I was able to manually export the database and view its contents. I had added a React button as a placeholder for a function React button to export the database.

On Tuesday, my primary goal was to relocate the exportFunction to the backend. With assistance from the software team, I moved the code to a new file called ExportFunction.ts. Initially, I encountered difficulties retrieving the correct container ID, as multiple containers with the same imageName caused confusion. I created a function called getContainerIds, which takes the imageName to return the container ID of a running container.

```
// Function to return the running IDs using the image name: timescale/timescaledb:latest-pg12
function getContainerIds(imageName: string): string | null {
    try {
        const command = `docker ps -q --filter "ancestor=${imageName}" --filter "status=running"`;
        const containerId = execSync(command, { encoding: 'utf-8' }).trim();
        return containerId || null;
    } catch (error) {
        console.error(`Error getting container ID: ${error.message}`);
        return null;
    }
}

const imageName = 'timescale/timescaledb:latest-pg12';
const allContainerIds = getContainerIds(imageName);
```

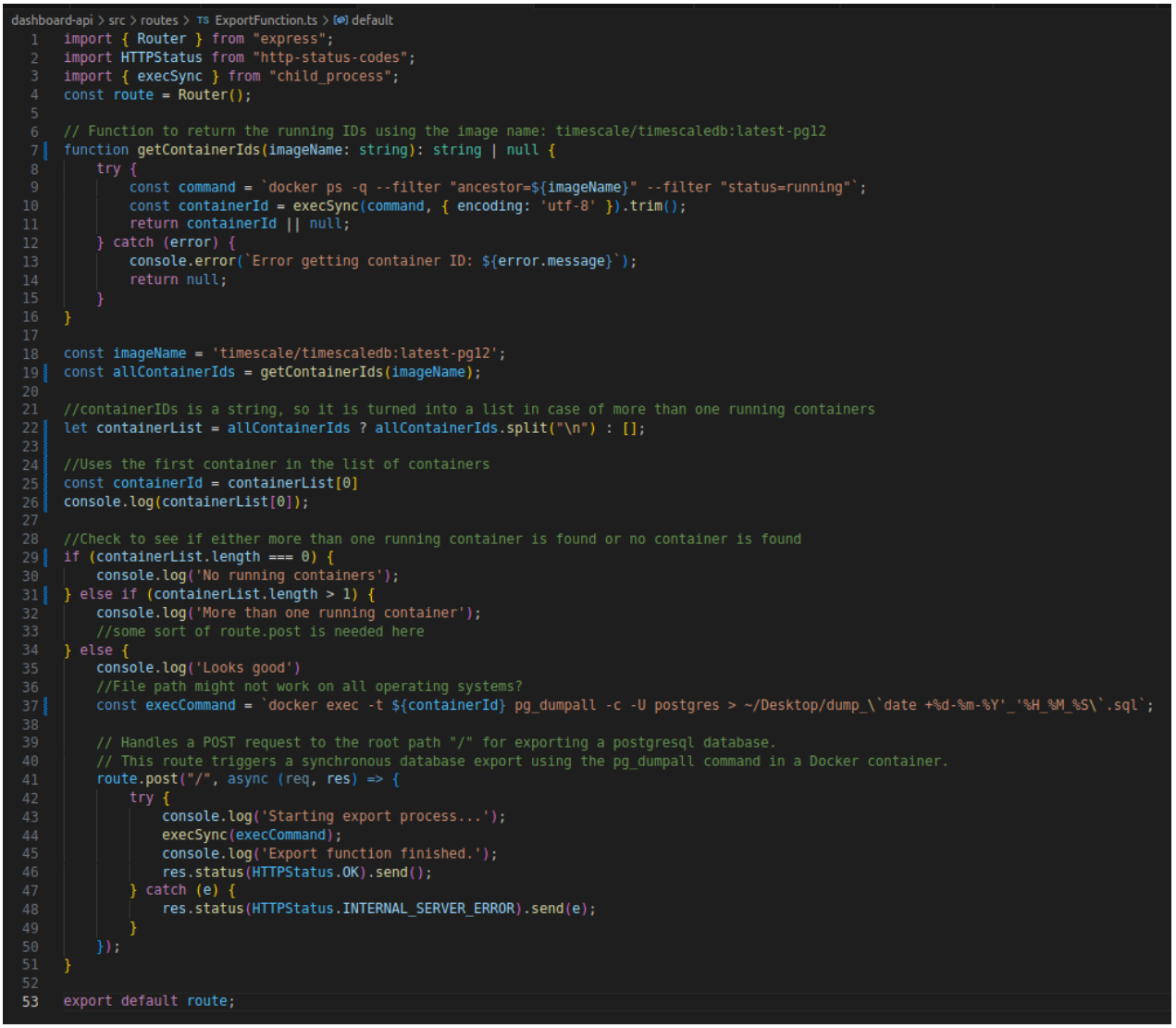
Figure 10 - Function that gets the running IDs.

I was able to take parts of existing code from another file that handled a POST request for exporting the database. The route triggered an asynchronous database export, ensuring that the database dump was running in the background, simultaneously with other processes.

```

    // Handles a POST request to the root path "/" for exporting a postgresql database.
    // This route triggers a synchronous database export using the pg_dumpall command in a Docker container.
    route.post("/", async (req, res) => {
        try {
            console.log('Starting export process...');
            execSync(execCommand);
            console.log('Export function finished.');
            res.status(HTTPStatus.OK).send();
        } catch (e) {
            res.status(HTTPStatus.INTERNAL_SERVER_ERROR).send(e);
        }
    });
}

export default route;
```

Figure 11 - Route that triggers the database export.

After speaking with the software team, I realised a potential issue that the user could run into was when there were multiple running containers using the same imageName. Since the getContainerIds function returned a string of running containers, I split the string into a list and changed the exec command so it would only use the first item in the list. If more than one running container is found, then this issue is logged to let the user know that the wrong running container may be being used. I used the alert method to display a warning on the UI if multiple running containers were found.



```
dashboard-api > src > routes > TS ExportFunction.ts > default
import { Router } from "express";
import HTTPStatus from "http-status-codes";
import { execSync } from "child_process";
const route = Router();

// Function to return the running IDs using the image name: timescale/timescaledb:latest-pg12
function getContainerIds(imageName: string): string | null {
    try {
        const command = `docker ps -q --filter "ancestor=${imageName}" --filter "status=running"`;
        const containerId = execSync(command, { encoding: 'utf-8' }).trim();
        return containerId || null;
    } catch (error) {
        console.error(`Error getting container ID: ${error.message}`);
        return null;
    }
}

const imageName = 'timescale/timescaledb:latest-pg12';
const allContainerIds = getContainerIds(imageName);

//containerIDs is a string, so it is turned into a list in case of more than one running containers
let containerList = allContainerIds ? allContainerIds.split("\n") : [];

//Uses the first container in the list of containers
const containerId = containerList[0]
console.log(containerList[0]);

//Check to see if either more than one running container is found or no container is found
if (containerList.length === 0) {
    console.log('No running containers');
} else if (containerList.length > 1) {
    console.log('More than one running container');
    //some sort of route.post is needed here
} else {
    console.log('Looks good')
    //File path might not work on all operating systems?
    const execCommand = `docker exec -t ${containerId} pg_dumpall -c -U postgres > ~/Desktop/dump_\`date +%d-%m-%Y'_'%H_%M_%S\`.sql`;

    // Handles a POST request to the root path "/" for exporting a postgresql database.
    // This route triggers a synchronous database export using the pg_dumpall command in a Docker container.
    route.post("/", async (req, res) => {
        try {
            console.log('Starting export process...');
            execSync(execCommand);
            console.log('Export function finished.');
            res.status(HTTPStatus.OK).send();
        } catch (e) {
            res.status(HTTPStatus.INTERNAL_SERVER_ERROR).send(e);
        }
    });
}

export default route;
```

Figure 12 - Complete code for database export.

In the front end, I began by adding a useState from React hook, allowing me to change the state of the functional component. I added two functions to handle a successful export and an unsuccessful export. If the export was successful, an alert was presented on the UI, and vice versa. Within the React button component, if it received an HTTPStatus.ok from the ExportFunction, then the program would present the successful export message. If it received any errors, then it would present an unsuccessful error message. I was also able to move the button out from the device view section to the main toolbar.

```
const [timeRange, setTimeRange, {undoLastChange}] = useComplexTimeRange()
const [exportSuccess, setExportSuccess] = useState(false);

/*
const [multipleContainer, setMultipleContainer] = useState(false);

const handleMultipleContainer = () => {
    setMultipleContainer(true);
    alert("Multiple Running Containers");
}
*/

const handleExportSuccess = () => {
    setExportSuccess(true);
    alert("Export Successful");
};

const handleExportFailure = () => {
    setExportSuccess(false);
    alert("Export Unsuccessful");
};
```

Figure 13 - Handling a successful and unsuccessful database export.

```
return (
    <Toolbar>
        <Grid container>
            {!props.isSidebarOpen
                && <Grid item><IconButton onClick={props.onSidebarOpen}>
                    <Menu />
                </IconButton></Grid>
            }
            <div style={{
                flexGrow: 1,
            }}/>

            <Grid style={{alignSelf: "center"}}>
                <HTTPTriggerButton
                variant="outlined"
                style={{marginRight: 10}}
                key={"Export"}
                method="post"
                path={"/export"}
                onComplete={(ok) => {
                    if (ok) {
                        handleExportSuccess();
                    } else {
                        handleExportFailure();
                    }
                }}
                >
                EXPORT DATABASE
                </HTTPTriggerButton>
            </Grid>
```

Figure 14 - React button for exporting the database.

Initially, the ExportFunction file did not work as expected. After some investigation, I was able to deduce that the ExportFunction route was not added to the main API routes in the index.ts file. This oversight meant that the ExportFunction was not recognised or accessible by the application, leading to the failure of the export

process. Therefore, I added the ExportFunction route to the list of API routes in the index.ts file.

```
function makeAPIRoute(extraRoutes: {
    key: string,
    route: Router,
}[]) {
    const route = Router();

    route.use(cors({
        origin: "*",
    }));

    route.use(express.json({}) as RequestHandler);
    route.use((req, res, next) => {
        res.set('Cache-Control', 'no-store')
        next()
    })

    route.use("/activeDevices", ActiveDeviceRoute);
    route.use("/devices", DeviceRoute);
    route.use("/deviceGroups", DeviceGroupRoute);
    route.use("/metrics", MetricsRoute);
    route.use("/export", ExportFunction);

    route.get("/displayInfo", (req, res)=>{
        res.send(DisplayInfo)
    });

    extraRoutes.forEach(e=>{
        route.use(e.key, e.route);
    })

    route.use((req, res)=>{
        res.status(400).send("Unknown API call");
    })
    return route;
}

export default makeAPIRoute;
```

Figure 15 - Adding the export function route to the list of API routes.

While working on this task, I ended up having many containers, the majority of which were restarting. This was not ideal. I was able to stop and remove all the restarting Docker containers manually, and I used pgAdmin to use the PostgreSQL terminal for database queries.

Result:

After rigorous debugging and refinement, I successfully implemented the export button on the dashboard interface. Despite encountering initial challenges with database export errors and frontend-backend communication, I managed to overcome them through effective problem-solving and collaboration with my peers. As a result, the export function works as intended, allowing users to now export the database for further analysis.

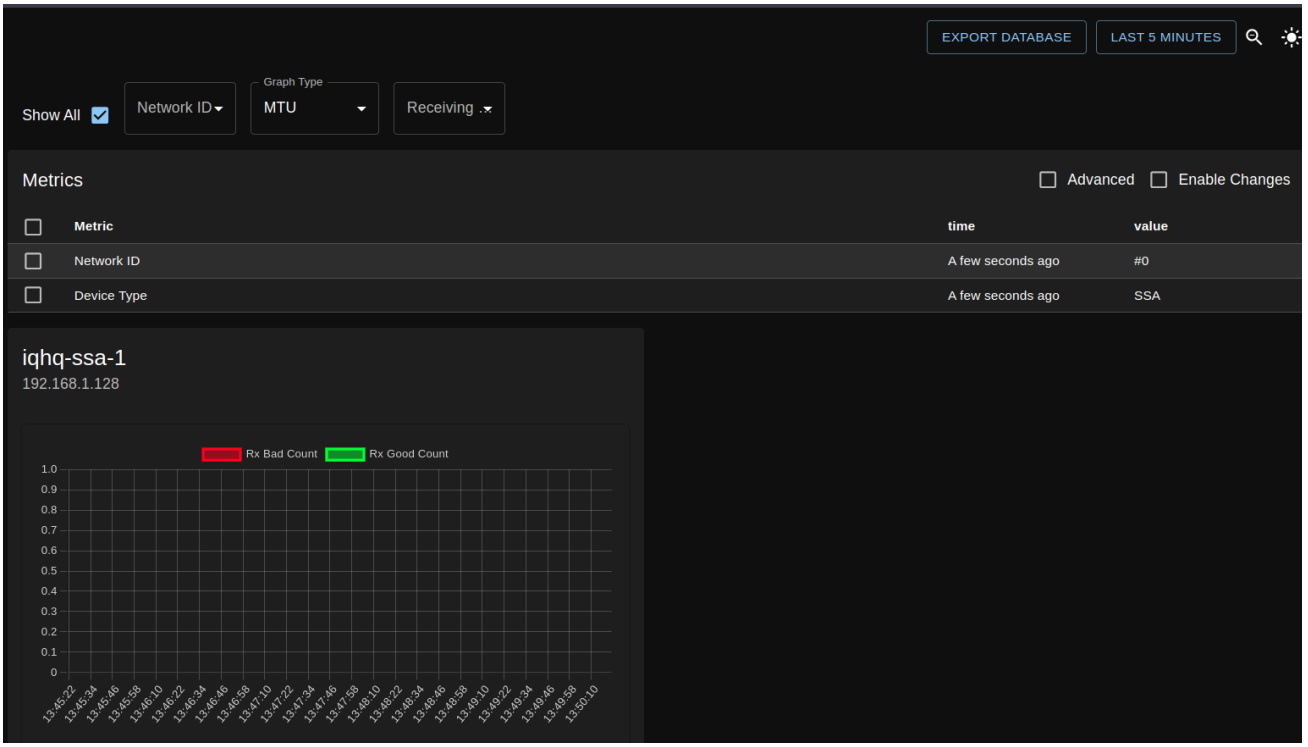


Figure 16 - UI dashboard with the export database button added.

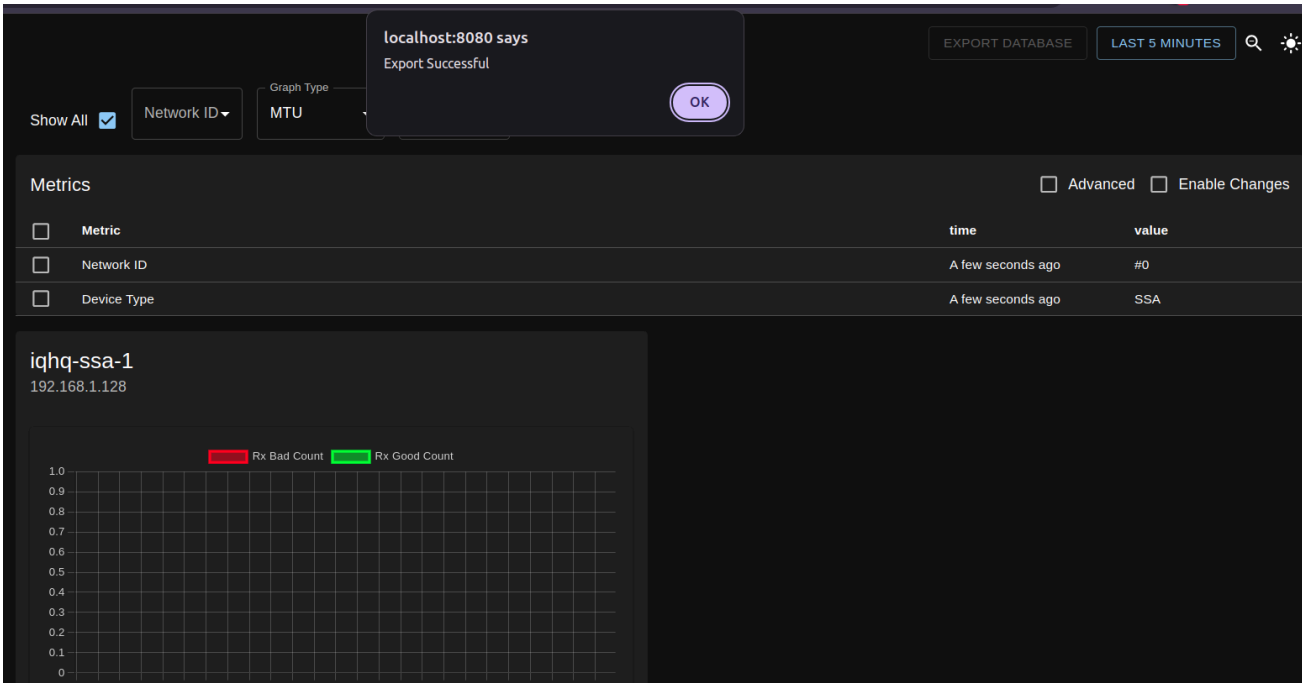


Figure 17 - The result of a successful database export.

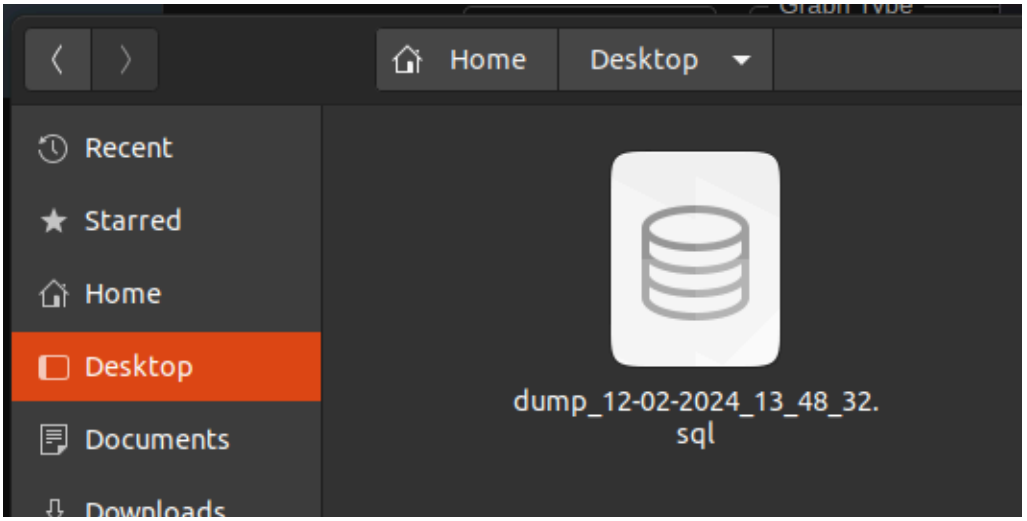


Figure 18 - The exported database file in the desktop directory.

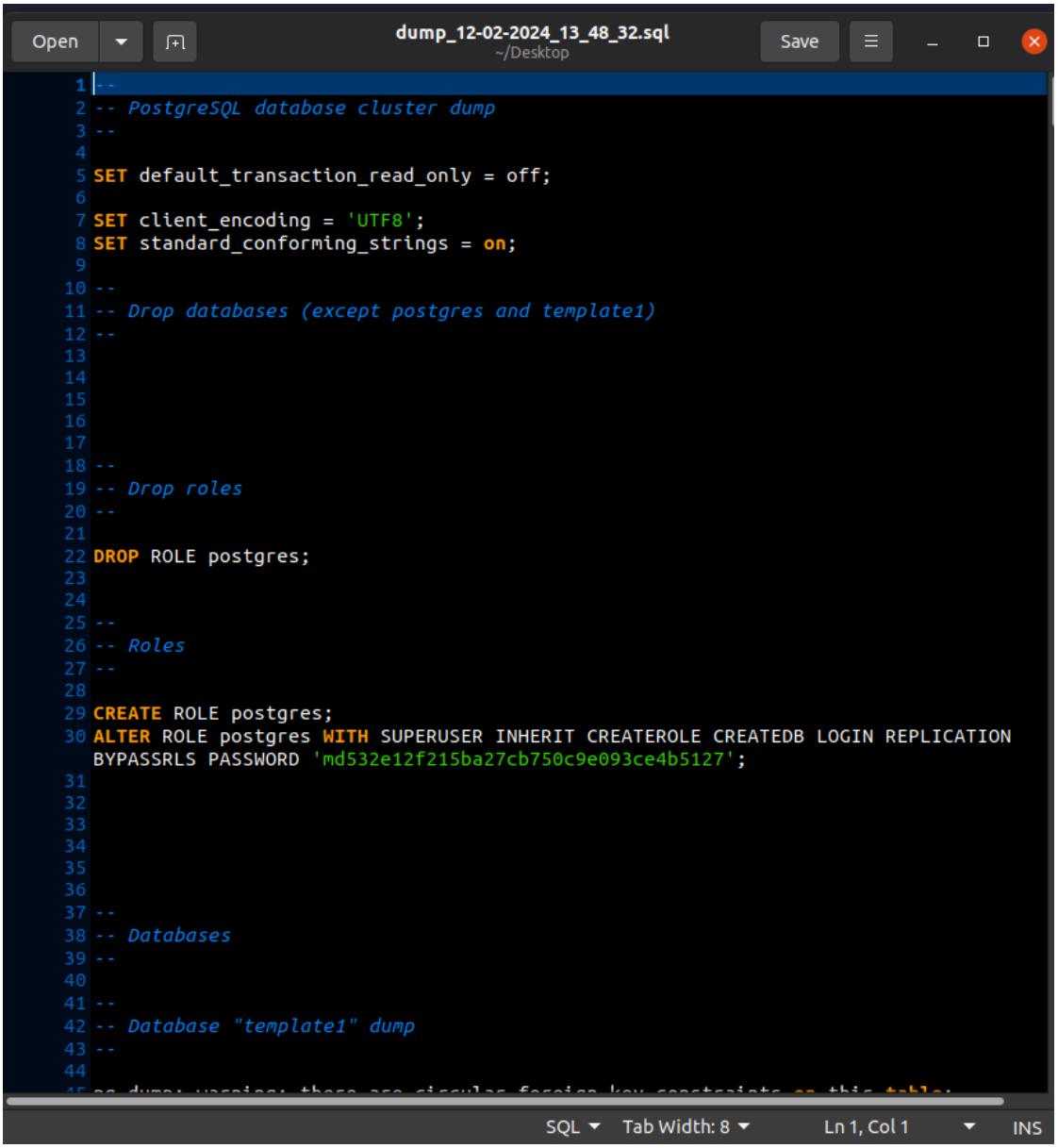


Figure 19 - The contents of the exported database file.

Reflection:

Throughout this task, I acquired valuable insights into database management, Docker utilisation, and front-end-backend integration. Working with technologies like JavaScript, TypeScript, and React deepened my understanding of frontend development, particularly React components and state management. Additionally, collaborating with the software team facilitated knowledge exchange and skill enhancement, enriching my overall learning experience.

Although the team was satisfied with my addition of the export feature, there are a few improvements I can think of to improve the usability of this feature. Currently the exported database is saved to the user's desktop directory on their device; in the future it would be more useful for the user to choose where to store the file. There

should also be the option to choose a filename, since saving the export by the date is not truly relevant or specific.

Task 2: Waterfall Data Capture

Situation:

Over the course of three days, from Wednesday, my primary objective was to capture waterfall data from one of our radio devices, the SSA and store it in a database, aligning with project requirements and enhancing data management capabilities. I also had to provide various stats on the data capture process.

Task:

My objective was to store SSA data into its own table within the database efficiently. Provide stats on the data capture process, including the capture rate, an ETA and disk usage analysis. I also needed to add a button that would start and stop the data capture process.

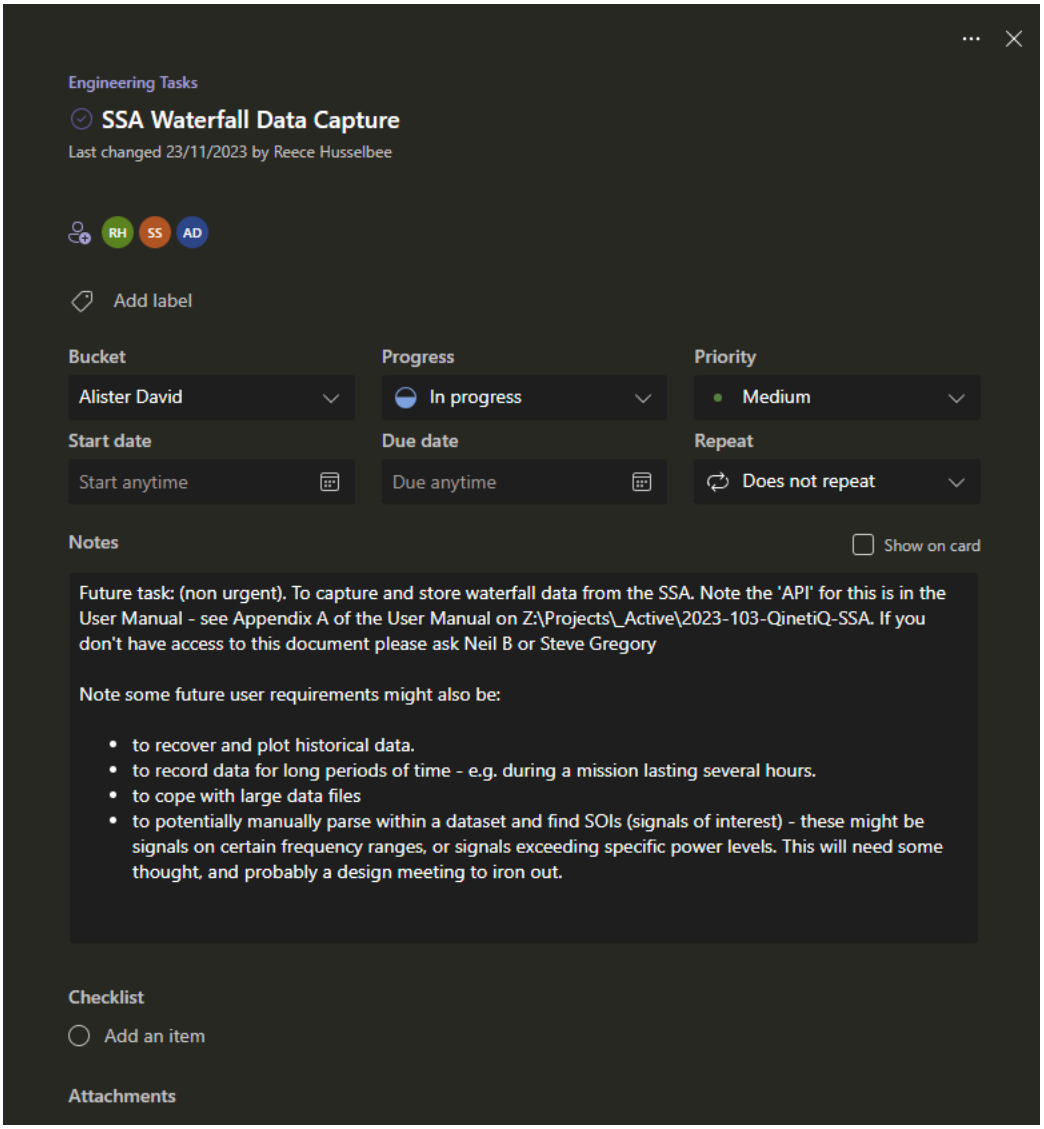
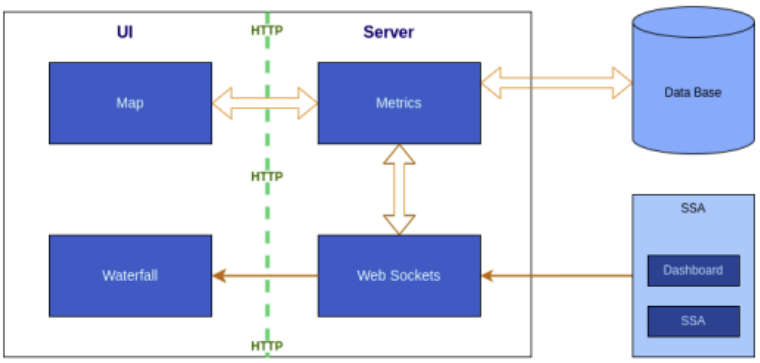


Figure 20 - SSA waterfall data capture task on teams.

SSA (Spectrum Situational Awareness):

Use:

Radio spectrum monitoring technology, designed for detecting radio transmissions of interest.



Requirements:

- Export SSA metrics to the database.
- Move SSA plugin into the dashboard, so that it can be sent with the other metrics.
- Find a method to manage data base size.
- Provide warnings to users if chosen SSA settings could lead to excessive data volume.

Future Features:

- Allow pre-filtering of SSA data based on power level ranges.
- Utilize data retention with periodic deletion to manage database size effectively.
- Give users the option to save all SSA metrics permanently, using an archiving or alternative storage approach.

Steps:

1. Add ssametrics table to the database.
2. Write method to store SSA metrics to the database.
3. Export SSA metrics to the database.
4. Incorporate SSA plugin to the main dashboard.

Figure 21 - Initial planning & requirements for the SSA waterfall data capture.

Action:

After breaking down the task into manageable steps, I began working on my first step, which was to add a table for the SSA metrics into the database. There are four main values that need to be recorded in the database table: device ID, time, power & frequency. There are eight tables in the dashboard; therefore, I was able to use pre-existing code from the other tables to create a table for SSA metrics. After several adjustments, I successfully added an SSA metrics table using PostgreSQL queries.

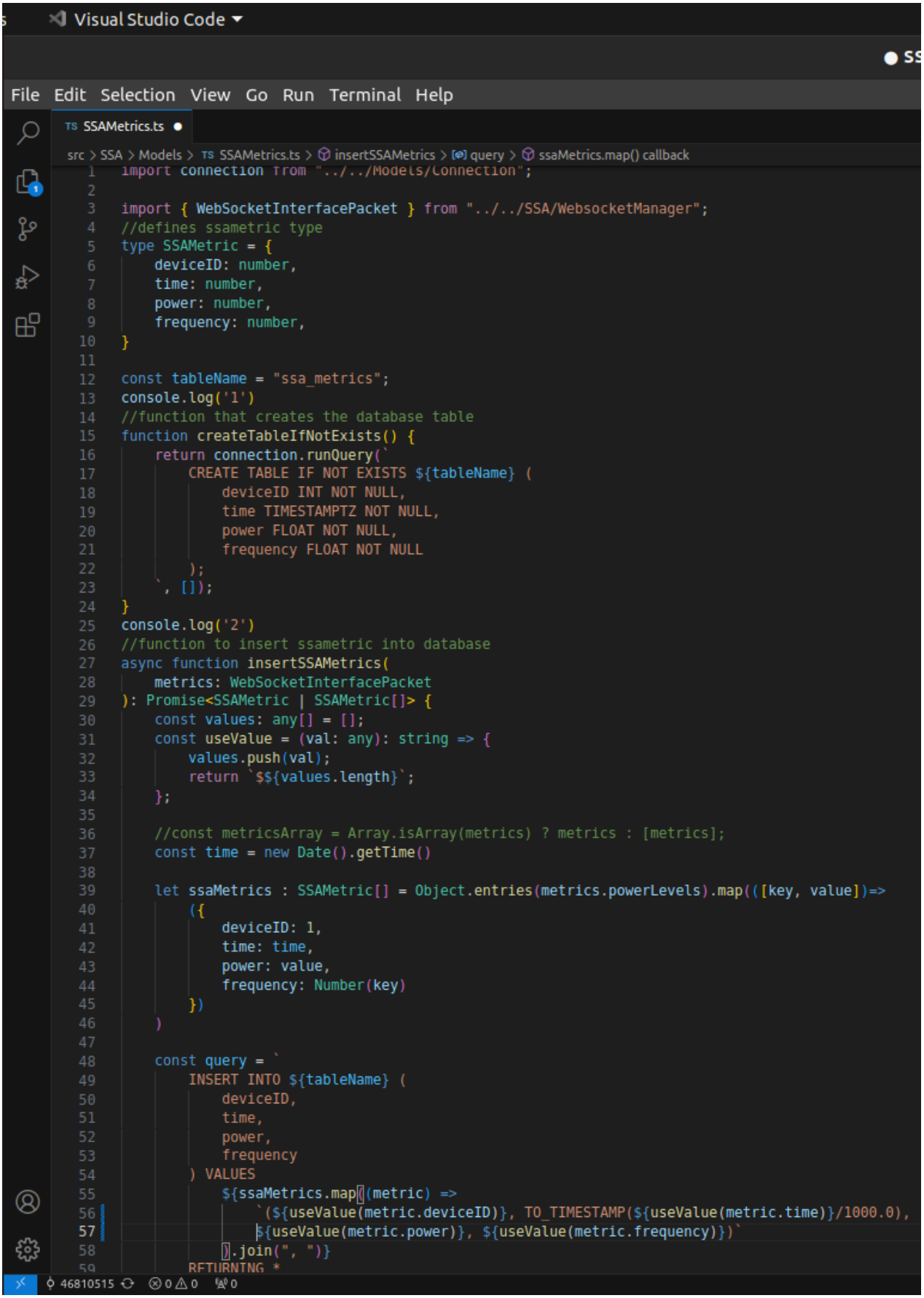


Figure 22 - File containing the code to set up the PostgreSQL table for ssa_metrics.

Figure 23 - File containing the code to set up the PostgreSQL table for ssa_metrics continued

I used pgAdmin to test if the table was created using psql commands, the table was successfully added. The next step was to figure out how to add data to the table. I added a file called testSSAMetrics, in which I created a function which would add sample data to the table.

dashboard-api > src > SSA > Models > TS testSSAMetrics.ts > ...

```ts
import { createTableIfNotExists, insertSSAMetrics, SSAMetric } from './SSAMetrics'

//fucntion to insert data into database table
async function testSSAMetrics() {
    try {

        //creates a table if it doesn't exist
        console.log('Table creation started...');
        await createTableIfNotExists();
        console.log('Table creation completed.');

        //test data
        const testMetrics: SSAMetric[] = [
            {
                deviceID: 1,
                time: new Date().getTime(),
                power: 10.5,
                frequency: 60,
            },
        ];

        //Inserts data into database table
        console.log('Inserting test data...');
        await insert(testMetrics);
        console.log('Test data inserted.');


    } catch (error) {
        console.error('Error:', error);
    }
}

testSSAMetrics();

```

Figure 24 - Function to test adding sample data to the ssa_metrics table.

I initially had an issue with the testSSAMetrics function since it was not working as intended. After adding logs, it seemed that the function was not being executed in the right order. After speaking with my team, I found out the Issue was caused by not using an async function. Using a regular function meant that my code was preventing other tasks from executing while waiting for asynchronous operations to complete. To address this issue, I modified the function responsible for adding test data to the PostgreSQL table to be an asynchronous function. This change allows the program to continue its execution while waiting for asynchronous operations, preventing unnecessary delays.

My primary goal for Thursday was to find a way to upload the SSA metrics coming from the SSA straight into the PostgreSQL table. I was able to locate a file called index within the SSA directory, which was initialising the SSA server. I added a function called addDataBaseListener to this file, which establishes a WebSocket

connection and adds the SSA metrics to the database using the insertSSAMetrics function from the SSAMetrics file.

```
const host = "localhost:3001";
const WebSocket = require('ws')

//function that establishes WebSocket connection and adds ssa data to database.
function addDataBaseListener() {

    let sock = new WebSocket("ws:" + host + "/api");
    sock.addEventListener("message", (async e=>{
        try {
            console.log('Inserting test data...');
            const data = JSON.parse(e.data) as WebSocketInterfacePacket;
            await insertSSAMetrics(data);
            // await insertSSAMetrics(data);
            // console.log('Test data inserted.');
            // console.log(data.powerLevels)

        } catch (error) {
            console.error('Error:', error);
        }
    }))
}

async function startupSSAPlugin(opts: {
    httpServer: Server,
    registerApiRoute: (key: string, route: Router)=>void,
}) {
    addWebsocketListener(opts.httpServer);
    await ...createTableIfNotExists();
    await FusionReportModel.createTableIfNotExists();
    await SSAMetricsModel.createTableIfNotExists();
    opts.registerApiRoute("/ssa", SSARoute);
}

export {
    startupSSAPlugin,
    addDataBaseListener,
};
```

Figure 25 - Function that established a WebSocket connection and inserts the SSA data into the ssa_metrics table.

I then added this function to the index file within the source directory so that the function was executed after all the SSA plugins had been set up. I also had to ensure the webSocketInterfacePacket, which serves as a data structure for the SSA metrics, was being exported from the WebsocketManager file so it could be used to get the SSA data for the SSA index file.

```
import { addDataBaseListener } from "./SSA"

(async ()=>{
    await createTablesIfNotExists()

    let ds = new DataStore(networkManager);
    const deviceManager = new DeviceManager(ds, networkManager);

    DeviceGroup.select()
        .then(groups=>{
            if(!groups.length) {
                console.log("Inserting default device group: ", initialDeviceGroup);
                deviceManager.addGroup(initialDeviceGroup);
            }
        })

    deviceManager.loadGroups();

    const app = express();
    const httpServer = createServer(app);

    const PluginAPIRoutes : {key: string, route: Router}[] = [];

    //This will initialise all things SSA for the server side
    await startupSSAPlugin({
        httpServer,
        registerApiRoute: (key, route)=>{
            PluginAPIRoutes.push({key, route});
        }
    });

    app.use("/tiles", proxy(TileServerPath));

    app.use("/api", makeAPIRoute(PluginAPIRoutes))

    app.use("/", express.static(ExpressConfig.publicRoot, {
        index: ExpressConfig.publicIndex
    }));
    // gqlServer.applyMiddleware(app, httpServer);
    app.use("/", (req, res)=>{
        res.sendFile(Path.join(process.cwd(), ExpressConfig.publicRoot, ExpressConfig.publicIndex));
    });

    await httpServer.listen(ExpressConfig.port);
    addDataBaseListener();
    console.log("Started server at with port ", ExpressConfig.port);
})()
```

Figure 26 - Adding the addDataBaseListener function to the index file.

```
src > SSA > TS WebsocketManager.ts > ...
import { WebSocket } from "ws";
import * as HTTP from "http"
import { TypedEmitter } from "tiny-typed-emitter";
import spectrumStreamManager from "./SpecturmStreamManager";

let connections : Connection[] = [];

type WebSocketInterfacePacket = {
    type: "SpectrumStreamData",
    streamId: number,
    powerLevels: {[freq:number]: number},
    capturesPerSecond: number,
    bps: number,
}

export type {
    WebSocketInterfacePacket,
}
```

Figure 27 - Exporting WebSocketInterfacePacket from the WebSocketManager file.

Currently, the webSocketInterfacePacket does not hold the device ID; instead, it uses the stream ID, which is not what is needed according to the project requirements. Therefore, I would need to find a way to upload device ID externally or by adding it to the data structure; however, that would require a lot of changes throughout the whole program since it could affect other areas.

After some investigation, I found out that with the current rate, there was 28mb of data being uploaded per minute; therefore, some steps will need to be taken to reduce this amount and make the process more efficient. By changing the data types of the database fields. Initially, device ID had the BIGINT data type, which meant it was storing eight-byte integers. This was unnecessary since the device ID can be stored using INT, which stores four-byte integers. Changing the data type of the device ID field reduced the size of the data being uploaded per minute by a few MB.

On Friday, I started by reviewing my approach to ensure it aligned with the project requirements. I outlined my next steps, including displaying the upload rate in MB/Sec, providing an estimated time of arrival (ETA) to indicate the remaining duration for data capture at the current upload rate, and presenting the remaining disk space on the device. I also need to add device ID to the database and ensure the seamless functionality of the buttons for initiating and stopping the data capture process.

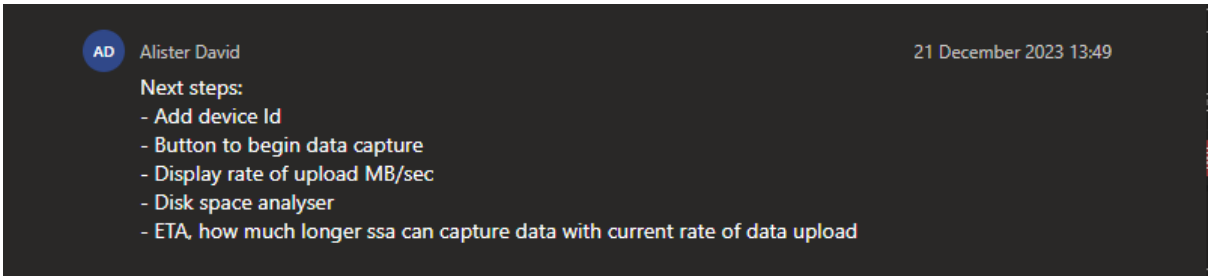


Figure 28 - Planning out next steps in teams.

I created a file called DataCaptureInfo.ts, which will hold functions to calculate the various data capture stats. I began by creating a function called getDiskSpace that would use the child_process module and the exec command to get the disk space in bytes. The function returns the available disk space in gigabytes. I added another function called getSSASize that runs a table size query on the SSA_metrics table. This function returns the SSA_metrics table size in megabytes. My idea behind calculating the capture rate was to constantly add the SSA_metrics table size to a list, and once the length of the list reaches a hundred, I would calculate the average of the difference between each consecutive item in the list. I would use this value to calculate the eta using the formula ETA = available space/capture rate. I created the function getDataCaptureRate that would do exactly that using the findAverageDifference function that returns the average of

the difference between each consecutive item in a list. The getDataCaptureRate function returns a list containing the eta followed by the capture rate.

12 Feb 14:36 ●

DataCaptureInfo.ts - dashboard-api - Visual Studio Code

TS SSAMetrics.ts ● | TS index.ts .../SSA | TS index.ts src | TS WebsocketManager.ts | TS DataCaptureInfo.ts ×

src > SSA > Models > TS DataCaptureInfo.ts > ...

```ts
import connection from "../../Models/Connection";
import { exec } from "child_process";
import { promisify } from "util";

const execAsync = promisify(exec);

// function that returns the available disk space on the device in GB
const getDiskSpace = async () => {
    try {
        const { stdout } = await execAsync('df -k /');
        const lines = stdout.trim().split('\n');
        const parts = lines[1].split(/\s+/);

        const availableSpaceKB = parseInt(parts[3], 10);
        const availableSpaceMB = availableSpaceKB / 1024;
        const availableSpaceGB = (availableSpaceMB / 1024).toFixed(2);
        return availableSpaceGB;
    } catch (error) {
        console.error(`Error getting disk space: ${error.message}`);
        throw error;
    }
};

// function that returns the size of the database in MB
const getDatabaseSize = async (): Promise<number> => {
    try {
        const db_result = await connection.runQuery("SELECT pg_size_pretty(pg_database_size('postgres')) as size");
        const sizeString = db_result.rows[0].size;
        const sizeInMB = parseInt(sizeString.replace(/\D/g, ''), 10);
        return sizeInMB;
    } catch (error) {
        console.error("Error getting database size:", error);
        throw error;
    }
};

// function that returns the size of the ssa_metrics table in MB
const getSSASize = async (): Promise<number> => {
    try {
        const ssa_result = await connection.runQuery("SELECT pg_size_pretty(pg_relation_size('ssa_metrics')) as size");
        const sizeString = ssa_result.rows[0].size;
        const sizeInMB = parseInt(sizeString.replace(/\D/g, ''), 10);
        return sizeInMB;
    } catch (error) {
        console.error("Error getting ssa_metrics size:", error);
        throw error;
    }
};
```

*Figure SEQ Figure * ARABIC 29 - File that contains functions that returns the various data capture stats.*

```
49
50  // function that returns the average of the difference between each consecutive items in an array
51  function findAverageDifference(arr) {
52      let sumOfDifferences = 0;
53
54      for (let i = 1; i < arr.length; i++) {
55        const difference = arr[i] - arr[i - 1];
56        //console.log(`${arr[i]}-${arr[i-1]} = ${difference}`)
57        sumOfDifferences += Math.abs(difference);
58      }
59      const averageDifference = sumOfDifferences / (arr.length - 1);
60      return averageDifference;
61    }
62
63  // function that returns the rate of data capture per sec
64  const createDataCaptureRateFunction = () => {
65      let captureList: number[] = [];
66      let returnList: number[] = [];
67      const getDataCaptureRate = async () => {
68          try {
69              const ssa_start = await connection.runQuery("SELECT pg_size_pretty(pg_relation_size('ssa_metrics')) as size");
70              const availableSpace = (parseFloat(await getDiskSpace())*1024);
71              const sizeString = ssa_start.rows[0].size;
72              const ssa_one = parseInt(sizeString.replace(/\D/g, ''), 10);
73
74              captureList.push(ssa_one);
75              console.log(captureList.length)
76              if (captureList.length === 100) {
77                  const currentCaptureRate = ((findAverageDifference(captureList))*10).toFixed(2)
78                  captureList = []
79                  const etaInSeconds = availableSpace / parseFloat(currentCaptureRate)
80                  const etaInMin = etaInSeconds / 60;
81                  const etaInHours = (etaInMin / 60).toFixed(2);
82                  returnList.push(parseFloat(etaInHours), parseFloat(currentCaptureRate))
83                  return returnList;
84              }
85              return null;
86          } catch (error) {
87              console.error("Error getting ssa_metrics size:", error);
88              throw error;
89          }
90      };
91
92      return getDataCaptureRate;
93  };
94
95  const getDataCaptureRate = createDataCaptureRateFunction();
```

Figure 30 - Function to find the data capture rate and ETA.

I had all the values logged for testing purposes to see if setting the list limit to one hundred was the right number or whether that number was too high. I made sure to place all the relevant functions in the index file so I could view the logs.

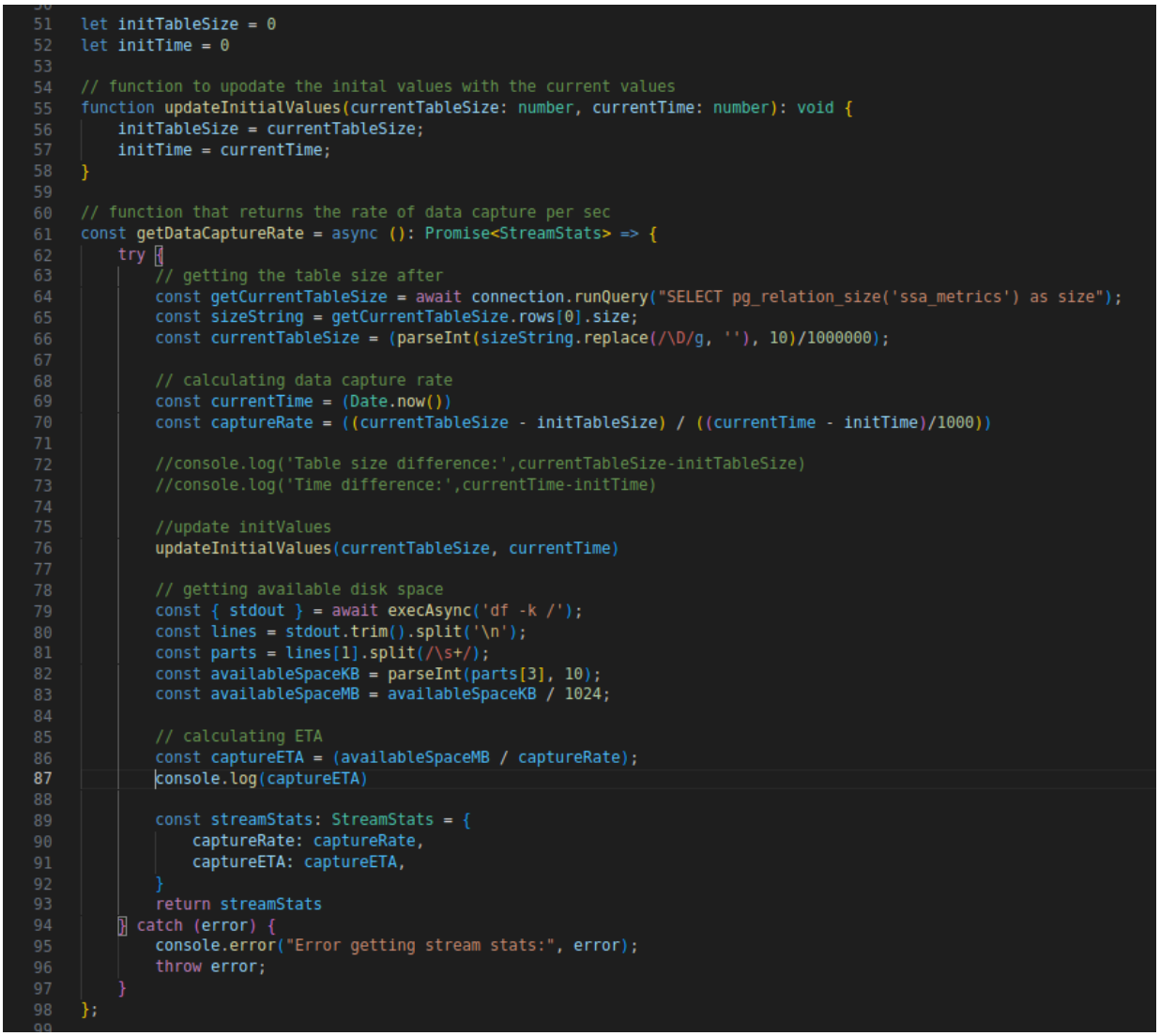
```
16  //function that establishes WebSocket connection and adds ssa data to database.
17  function addDataBaseListener() {
18
19      let sock = new WebSocket("ws:" + host + "/api");
20      sock.addEventListener("message", (async e=>{
21          try {
22              const dataBaseSize = await DataCaptureInfo.getDatabaseSize();
23              console.log(`\nData Base: ${dataBaseSize} MB`)
24
25              const ssaSize = await DataCaptureInfo.getSSASize();
26              console.log(`ssa_metrics: ${ssaSize} MB`)
27
28              const diskSpace = await DataCaptureInfo.getDiskSpace();
29              console.log(`Disk Space Available: ${diskSpace} GB\n`)
30
31              const captureRate = await DataCaptureInfo.getDataCaptureRate();
32              if (captureRate != null) {
33                  console.log(`Capture Rate: ${captureRate[1]} MB/s`)
34                  console.log(`Capture ETA: ${captureRate[0]} hrs`)
35              }
36
37              //console.log('Inserting test data...');
38              const data = JSON.parse(e.data) as WebSocketInterfacePacket;
39              await insertSSAMetrics(data);
40          } catch (error) {
41              console.error('Error:', error);
42          }
43      }))
44  }
45
```

Figure 31 - Logging the results of the data capture stats.

```
97

Data Base: 119 MB
ssa_metrics: 106 MB
Disk Space Available: 791.79 GB

98

Data Base: 119 MB
ssa_metrics: 106 MB
Disk Space Available: 791.79 GB

99

Data Base: 119 MB
ssa_metrics: 106 MB
Disk Space Available: 791.79 GB

100
Capture Rate: 0.61 MB/s
Capture ETA: 369.21 hrs

Data Base: 119 MB
ssa_metrics: 106 MB
Disk Space Available: 791.79 GB

1

Data Base: 119 MB
ssa_metrics: 106 MB
Disk Space Available: 791.79 GB

2

Data Base: 119 MB
ssa_metrics: 106 MB
```

Figure 32 - Initial results of the data capture stats implementation.

After manually checking the capture rate by recording the size of the SSA_metrics table before and after a period of time. The capture logged using the getCaptureRate function was not always accurate. This makes sense since you would not be viewing the live capture rate because, by the time the function returns the average from a list of a hundred, the actual capture rate would have changed. A better way to calculate the average would be to have two variables, one to hold the values of the previous SSA_metrics table size and one that holds the new size. You would also need other variables, one to calculate the time of finding the previous table size and a timestamp of the new table size. Then, to calculate the live capture rate, you would use the formula capture rate = (new table size - old table size) / (new time - old time). This would give you a more accurate representation of the capture rate and, therefore, a more accurate eta.



```
let initTableSize = 0
let initTime = 0

// function to upodate the inital values with the current values
function updateInitialValues(currentTableSize: number, currentTime: number): void {
    initTableSize = currentTableSize;
    initTime = currentTime;
}

// function that returns the rate of data capture per sec
const getDataCaptureRate = async (): Promise<StreamStats> => {
    try {
        // getting the table size after
        const getCurrentTableSize = await connection.runQuery("SELECT pg_relation_size('ssa_metrics') as size");
        const sizeString = getCurrentTableSize.rows[0].size;
        const currentTableSize = (parseInt(sizeString.replace(/\D/g, ''), 10)/1000000);

        // calculating data capture rate
        const currentTime = (Date.now())
        const captureRate = ((currentTableSize - initTableSize) / ((currentTime - initTime)/1000))

        //console.log('Table size difference:',currentTableSize-initTableSize)
        //console.log('Time difference:',currentTime-initTime)

        //update initValues
        updateInitialValues(currentTableSize, currentTime)

        // getting available disk space
        const { stdout } = await execAsync('df -k /');
        const lines = stdout.trim().split('\n');
        const parts = lines[1].split(/\s+/);
        const availableSpaceKB = parseInt(parts[3], 10);
        const availableSpaceMB = availableSpaceKB / 1024;

        // calculating ETA
        const captureETA = (availableSpaceMB / captureRate);
        console.log(captureETA)

        const streamStats: StreamStats = {
            captureRate: captureRate,
            captureETA: captureETA,
        }
        return streamStats
    } catch (error) {
        console.error("Error getting stream stats:", error);
        throw error;
    }
};
```

Figure 33 - Updated function to find the capture rate and ETA.

Result:

As a result of my efforts over the three days, I successfully achieved the objectives outlined in the task. I established a robust process for capturing waterfall data from the SSA device and storing it efficiently in the database. Additionally, I implemented functionalities to provide insightful statistics on the data capture process, including the capture rate, an ETA, and disk usage analysis. By addressing issues and refining the methods iteratively, such as optimising data types and ensuring asynchronous operations, I enhanced the overall efficiency and accuracy of the data capture system. Despite encountering challenges along the way, such as inaccuracies in the initial capture rate calculations, I implemented a more accurate method for calculating the live capture rate. Overall, the project was completed successfully, meeting the project requirements and laying a solid foundation for future enhancements and optimisations.

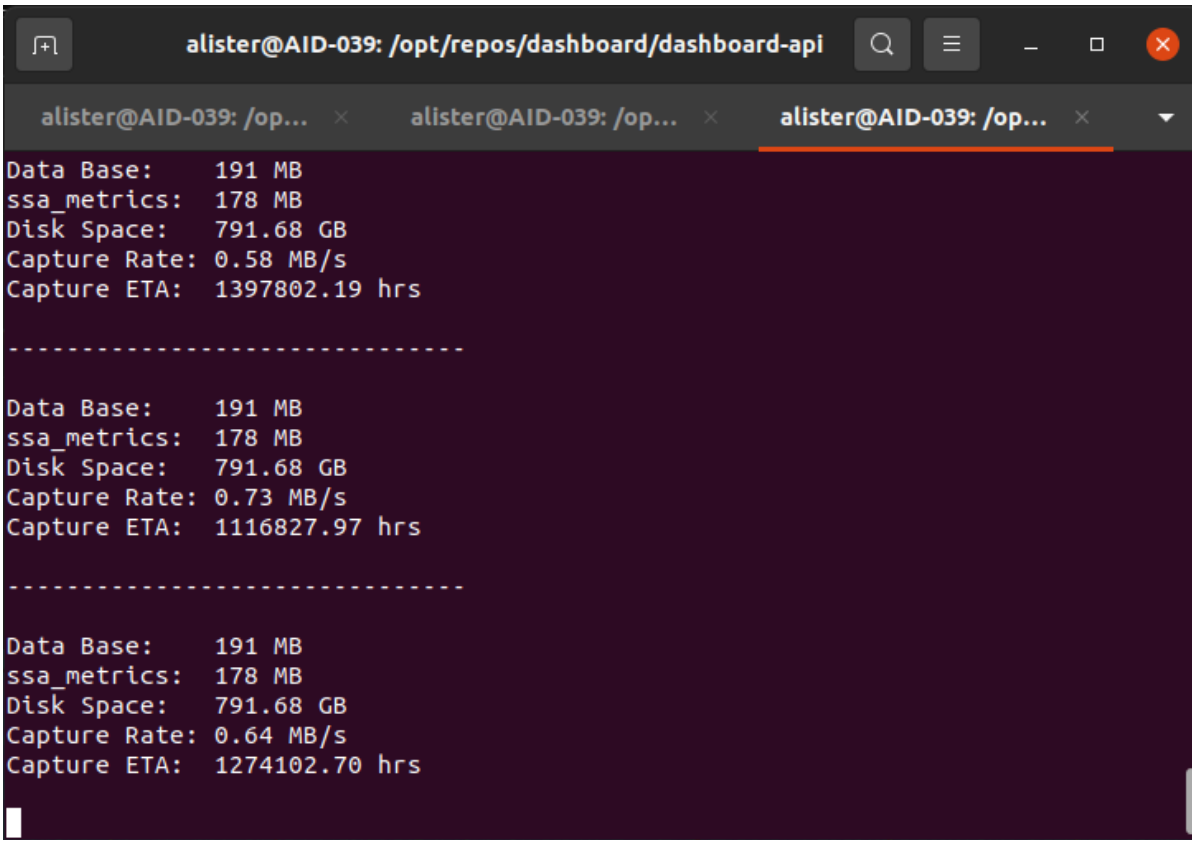


Figure 34 - Results of the data capture stats implementation.

Reflection:

In my recent learning journey, I have mastered creating database tables using PostgreSQL queries and testing them with tools like pgAdmin. Understanding how to structure tables for specific metrics was crucial. I realised the importance of using asynchronous functions for tasks like database operations to avoid delays caused by synchronous functions. Additionally, I learned how choosing the right data types directly impacts efficiency. These experiences gave me practical insights into system integration and WebSocket connections with databases. I was able to understand the importance of modularity when I began working on the dataCaptureInfo file. I tried fitting everything into one function, which led to confusion and decreased the readability of the code. By using separate functions, it was easier to read and understand the code, and I was able to reuse functions. I was able to explore alternative algorithms to find an accurate capture rate. I was also able to exercise my knowledge of the child_process module as well as async functions.

There are numerous ways to improve the functionality and efficiency of the current data capture process. The user is given the ability to choose the SSA settings manually on the dashboard, which directly affects the size of the data received from the SSA and being uploaded to the database. To prevent capturing excessive data volumes, a warning could be displayed to let the user know that the chosen settings could affect the database. A more effective way to manage the capture rate is to abstract the data, being received, therefore only storing values which could be relevant to the user. Building on this method of data storage, the user could select a specific range of data to store rather than storing all incoming data from the SSA.

Conclusion:

In conclusion, my recent tasks at IQHQ have provided invaluable learning experiences in database management, frontend-backend integration, and system optimisation. By implementing export functionality on the dashboard interface and capturing waterfall data from the SSA device, I expanded my skills in utilising technologies like PostgreSQL, Docker, and React, while also gaining insights into efficient data storage and retrieval techniques. Through collaboration with colleagues and iterative problem-solving, I successfully overcame challenges and delivered solutions that met project objectives. Reflecting on these experiences, I recognise the importance of modularity in code organisation and the significance of choosing the right tools and algorithms for optimal performance. Moving forward, there are several opportunities for further enhancements, such as refining the data capture processes, providing more user control over data management, and improving data visualisation capabilities. Overall, these tasks have not only expanded my technical skill set but also deepened my understanding of effective software development practices, setting a solid foundation for future projects.