PROJECT TWO

Database Export Feature

Abstract

This document presents an analysis of the database export feature, focusing on its design, development, testing, and deployment. The feature allows users to export the database efficiently and reliably. The export feature aims to enhance the utility and user satisfaction of the product by providing users with the capability to export database logs for further analysis and reference.

Alister David

alisterdavid03@gmail.com

Contents

The Task: 2
Feasibility Study: 3
  Scope Definition: 3
  Technical Feasibility: 4
Requirements Analysis: 7
User Story: 8
Design: 9
  UI Component Design: 9
  UML Diagram: 10
  Pseudo Code: 11
Development: 13
  Exporting the database: 13
  Initial front-end integration of a react button: 15
  ExportFunction File: 16
  Initial Testing: 19
  Docker Issues: 20
  Handling multiple running docker containers: 21
  Updating Database Export Command for File Path Location: 23
  Handling export success/failure using useState in React: 23
  Successful Implementation of Export Functionality: 25
Testing: 25
  Unit Testing: 25
  Integration Testing: 30
  Performance Testing: 35
Deployment: 40
Maintenance: 43
Conclusion: 44

The Task:

The task for Project 2 involves enhancing the functionality of the dashboard interface at IQHQ by implementing an export feature. This feature aims to enable users to export database logs to their devices, thereby improving the product's utility and customer satisfaction.

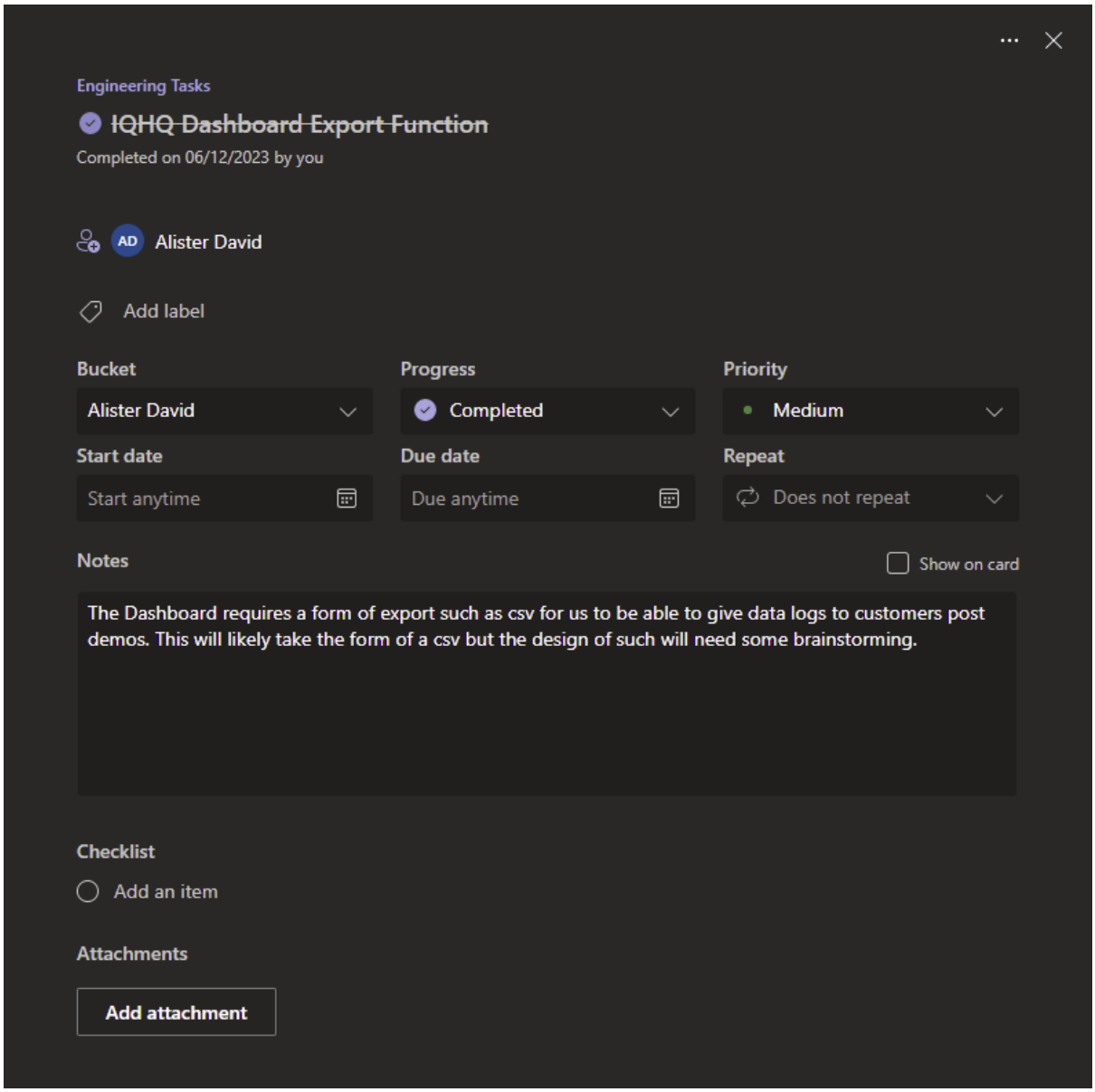


Figure 1 - Dashboard export function set on teams.

Feasibility Study:

Scope Definition:

Objective:

The primary objective of this project is to implement an export feature in the dashboard interface at IQHQ, enabling users to export database logs. The export feature aims to enhance the utility and user satisfaction of the product by providing users with the capability to export database logs for further analysis and reference.

Specific Requirements:

- **File Formats Supported**: The export feature should support multiple file formats for exporting database logs, such as CSV, Excel, or PDF, to cater to varying user preferences and use cases.
- **User Interface Integration**: Implement an export button or similar user interface element within the dashboard interface to initiate the database export process.
- **Backend Implementation**: Develop backend functionality to handle the database export process, including retrieving all database logs and generating the export file.
- **Export Entire Database**: The export feature should allow users to export the entire database logs without the need for selecting specific criteria or options.
- **Compatibility with Existing Functionalities**: The export feature should seamlessly integrate with existing functionalities of the dashboard interface, maintaining consistency in user experience and workflow.
- **Progress Feedback**: Provide feedback to users during the export process, indicating the progress and status of the export operation to ensure transparency and user confidence.
- **Error Handling**: The system should handle errors gracefully and provide informative error messages to users in case of export failures or data inconsistencies.
- **Scalability**: Ensure that the export feature is scalable to accommodate future growth in data volume and user demand without compromising performance or reliability.

Technical Feasibility:

Database Structure and Management:

- Evaluate the compatibility of the existing database structure (PostgreSQL) with the export feature requirements.
- Ensure that the database schema is designed to efficiently store and retrieve the required data for export.

Relational & Non-relational Databases:

In assessing the technical feasibility of implementing the export feature in the dashboard interface at IQHQ, it is essential to consider the principles and uses of relational and non-relational databases.

Relational Database (PostgreSQL):

- PostgreSQL, as the chosen relational database management system at IQHQ, follows principles of relational databases.
- Relational databases organise data into structured tables with predefined schemas, facilitating data integrity and consistency.
- PostgreSQL offers robustness, flexibility, and comprehensive support for SQL-based data manipulation, making it suitable for managing structured data in IQHQ's environment.

Non-relational Database Considerations:

- While PostgreSQL meets the structured data requirements of IQHQ, consideration was given to non-relational databases.
- Non-relational databases, such as MongoDB or Cassandra, offer advantages in handling unstructured or semi-structured data and scalability.
- However, for the export feature project, the structured nature of database logs and the existing familiarity with PostgreSQL favoured its selection over non-relational alternatives.

There are several reasons as to why a relational database is the most suitable type of database for storing data at IQHQ. Relational databases enforce strict data integrity constraints, such as primary keys, foreign keys, and referential integrity, ensuring that data remains consistent and accurate. This is crucial in military applications where precision and reliability are critical, as any inaccuracies or inconsistencies in data could have serious consequences.

Relational databases organise data into structured tables with predefined schemas, making it easier to model and manage complex relationships between different data entities. This structured

approach is well-suited for storing device metrics and data from various military devices, allowing for efficient querying and analysis.

Moreover, relational databases offer powerful querying capabilities using SQL, allowing users to perform complex queries to retrieve, filter, and analyse data. This flexibility is essential for generating actionable insights from device metrics and operational data, enabling informed decision-making in military operations.

Additionally, relational databases provide robust security features, including user authentication, authorisation, and encryption, to protect sensitive military data from unauthorised access or tampering. This is crucial for maintaining confidentiality and ensuring that only authorised personnel have access to classified information.

Relational databases often have built-in features and mechanisms to support regulatory compliance requirements, such as data privacy regulations or industry standards for handling sensitive information. This helps the company ensure compliance with government regulations and contractual obligations when handling military data.

Lastly, relational databases facilitate integration with other systems and applications used in military operations. This allows for seamless data exchange and communication between different defence systems, enhancing collaboration and coordination among various stakeholders.

Backend Technology Stack:

- Assess the suitability of the backend technologies (e.g., Node.js, TypeScript) for implementing the export functionality.
- Determine if the chosen technologies support database connectivity and file manipulation required for exporting database logs.

The backend technologies of Node.js and TypeScript are well-suited for implementing the export functionality in the project. With strong support for database connectivity, evidenced by dependencies like pg for PostgreSQL interaction, these technologies enable efficient retrieval of database logs. Additionally, Node.js provides powerful file manipulation capabilities, complemented by TypeScript's type safety and code maintainability advantages. The project's development environment, featuring scripts for automatic server reloading and TypeScript execution, fosters a conducive atmosphere for feature implementation and testing. Supported by extensive community backing and a rich ecosystem, Node.js and TypeScript offer ample resources for backend development tasks, ensuring the successful integration of the export feature with robust database log retrieval and export file generation functionalities.

Export Mechanism:

- Research and identify appropriate methods for exporting database logs, considering factors such as database size, performance, and file format requirements.
- Implement backend logic to retrieve database logs efficiently and generate the export file in the desired format.

In addressing the task of exporting database logs, I conducted extensive research and identified methods, considering factors like database size, performance, and file format requirements. Given the substantial size of the database, spanning several gigabytes with extensive metric data, it was crucial to choose a method that could efficiently handle this volume of information without compromising performance.

While the initial proposal suggested using a CSV file format, upon thorough examination of available options, I opted to export the logs in an SQL file format instead. My decision was based on the SQL file's ability to maintain the database's structural integrity, preserving complex relationships and data hierarchies inherent in the system. Additionally, the SQL file format provides better support for large datasets and offers more robust handling of various data types and structures present in the database.

In implementing the backend logic for efficient log retrieval and export file generation, I carefully optimised database queries and file writing processes to minimise resource consumption and maximise performance. This involved leveraging appropriate database querying techniques, such as batching, to efficiently retrieve logs without overwhelming system resources. Furthermore, implementing mechanisms for asynchronous processing or parallel execution helped expedite the export process, ensuring the timely generation of the export file.

Overall, my research-driven approach to identifying suitable export methods and the decision to utilise an SQL file format reflects a strategic consideration of performance, scalability, and data integrity concerns inherent in exporting large volumes of database logs.

Integration with Frontend:

- Ensure seamless integration of the export feature with the frontend dashboard interface built using React components.
- Implement frontend UI elements (e.g., export button) to initiate the export process and provide feedback to users.

In considering the technical feasibility of integrating the export feature with the frontend dashboard interface, the decision to utilise React components stemmed from several key factors. Firstly, the entire dashboard frontend already constructed using React components, therefore the task is to seamlessly incorporate the export functionality within this framework.

Reacts component-based architecture aligns well with the modular nature of frontend development, allowing for the creation of reusable and composable UI elements. This modular approach not only promotes code reusability and maintainability but also facilitates scalability as the dashboard interface evolves and expands with additional features.

Moreover, React's virtual DOM (Document Object Model) and efficient rendering mechanisms contribute to optimal performance, ensuring smooth and responsive user interactions even with complex UI components and large datasets, which is crucial for maintaining a seamless user experience in the dashboard interface.

Additionally, React's ecosystem offers a rich selection of libraries, tools, and community support, providing developers with resources to streamline development processes and address challenges effectively. Through leveraging React components for the frontend dashboard interface, I aimed to capitalise on these benefits to implement the export feature efficiently while ensuring compatibility, performance, and maintainability across the dashboard application.

Backend-frontend Communication:

- Establish communication protocols and APIs for frontend-backend interaction to facilitate the export functionality.
- Implement mechanisms for passing export requests from the frontend to the backend and receiving export status updates.

In terms of backend-frontend communication, my chosen method of API relies on HTTP-based routes implemented using the Express framework in Node.js. I opted for this approach due to its simplicity, versatility, and widespread adoption in web development. HTTP provides a standardised protocol for client-server communication, making it suitable for exchanging export requests and status updates between the frontend and backend components of our application. By defining specific routes for export functionality, I establish clear endpoints for frontend interaction, enabling seamless integration of export features into the dashboard interface.

Additionally, Express simplifies the implementation of these routes with its concise and intuitive syntax, allowing me to focus on core functionality rather than low-level networking concerns. Overall, my choice of HTTP-based APIs and Express for backend-frontend communication ensures robust, efficient, and scalable interaction between the frontend and backend components, facilitating the seamless implementation of export functionality in our application.

Error Handling and Logging:

- Develop robust error handling mechanisms to handle exceptions and errors encountered during the export process.
- Implement logging functionalities to track export activities and capture relevant information for troubleshooting and auditing purposes.

In ensuring the reliability and stability of the export process, I developed robust error handling mechanisms to gracefully manage exceptions and errors encountered. By implementing thorough error handling procedures, I can effectively identify and address any issues that arise during the export process, minimising disruptions and ensuring smooth operation. Additionally, logging functionalities were incorporated to track export activities comprehensively. This logging system captures relevant information for troubleshooting and auditing purposes, providing valuable insights into the export process, and facilitating timely resolution of any encountered issues.

Requirements Analysis:

Background:

Our dashboard application serves as a central hub for monitoring and managing various aspects of a system. One common requirement expressed by several companies is the ability to export the data stored in the PostgreSQL database used by the dashboard. This feature is essential for generating backups and facilitating data analysis by external tools.

Functional Requirements:

1. **Export Functionality**: Users should be able to export the entire PostgreSQL database used by the dashboard.
2. **User Feedback**: Upon initiating the export process, users should receive clear feedback indicating whether the export was successful or not.
3. **Location of Exported Data**: The exported database file should be stored in a location easily accessible to users, such as the desktop.

Non-Functional Requirements:

1. **Reliability**: The export process should be reliable, ensuring that no data is lost during the export.
2. **User Interface**: The export functionality should be integrated seamlessly into the dashboard's user interface, making it intuitive for users to access and use.
3. **Performance**: The export process should be efficient, minimising the time required to complete the export operation.
4. **Error Handling**: The system should handle any errors encountered during the export process gracefully, providing meaningful error messages to users.

User Story:

Title: Export PostgreSQL Database As a dashboard user, I want to export the PostgreSQL database used by the dashboard So that I can create backups and analyse the data using external tools.

Acceptance Criteria:

1. When I click on the "Export" button in the dashboard toolbar,
2. Then I should receive feedback indicating whether the export was successful or not.
3. If the export is successful, the database file should be accessible on my desktop.
4. If multiple running containers are detected, the system should inform me and prompt for further action.
5. Any errors encountered during the export process should be handled gracefully, with meaningful error messages displayed to me.

This user story outlines the need for the export functionality and specifies the expected behaviour and outcomes when using the feature.

Design:

UI Component Design:

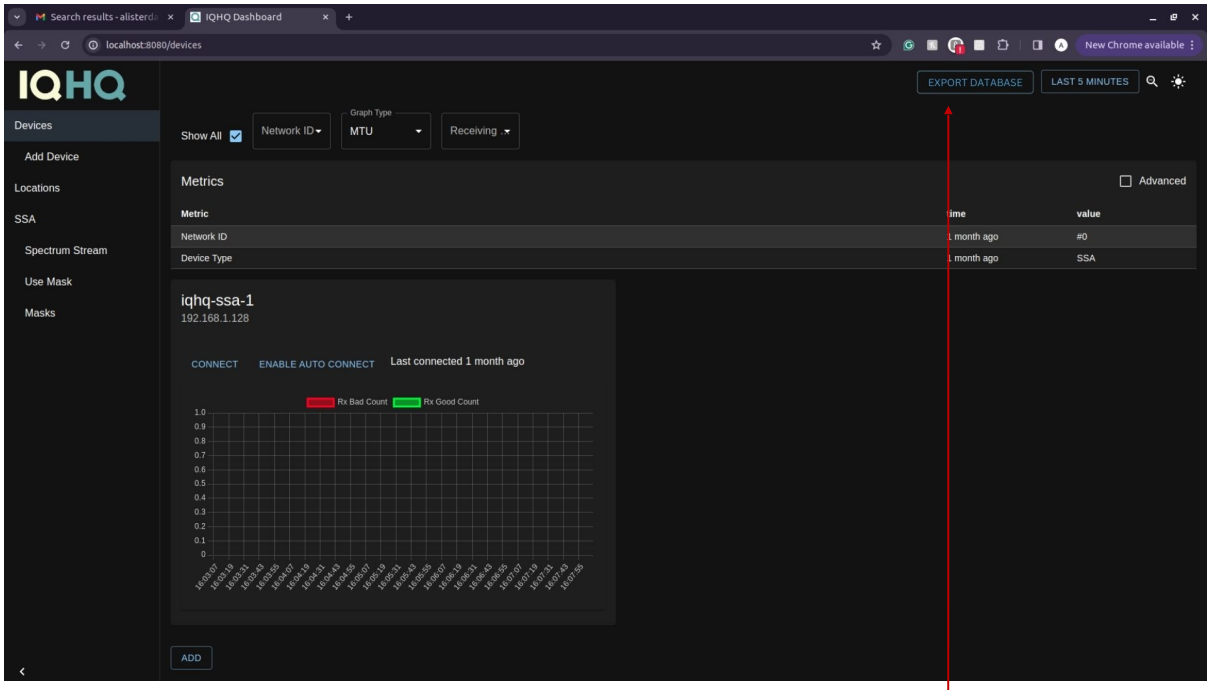


Figure 2 - Proposed database export button design.

Proposed Database Export Button

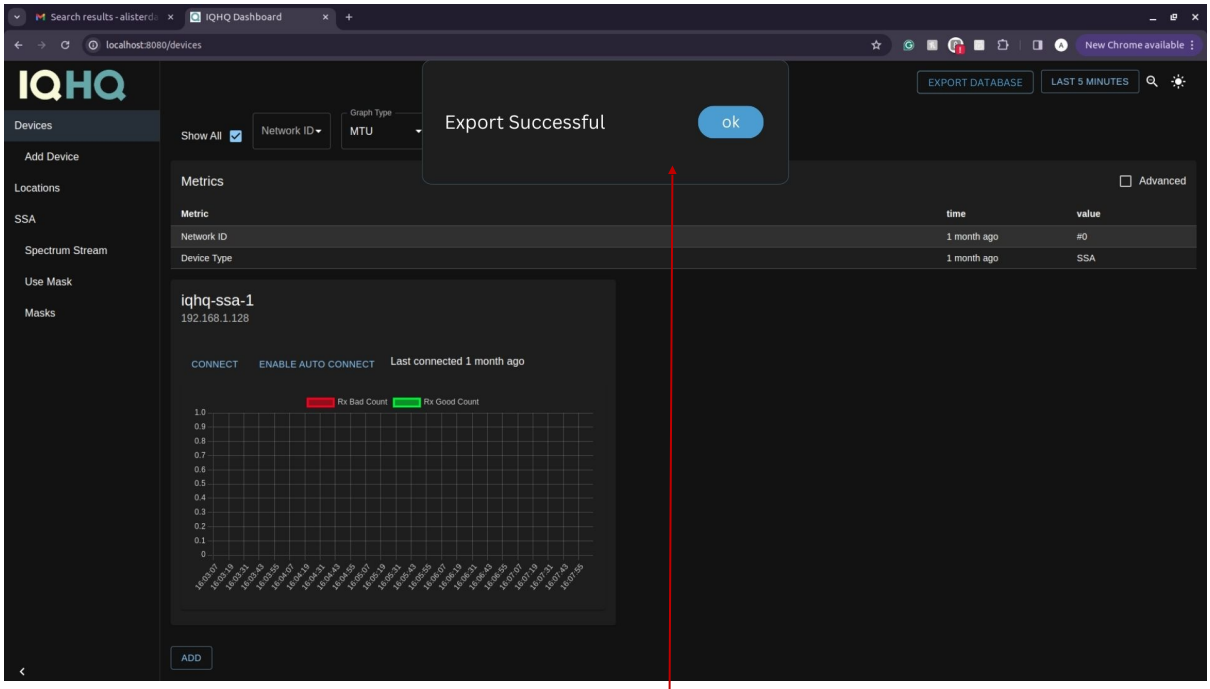


Figure 3 - Proposed successful export message design.

Proposed Successful Export Message

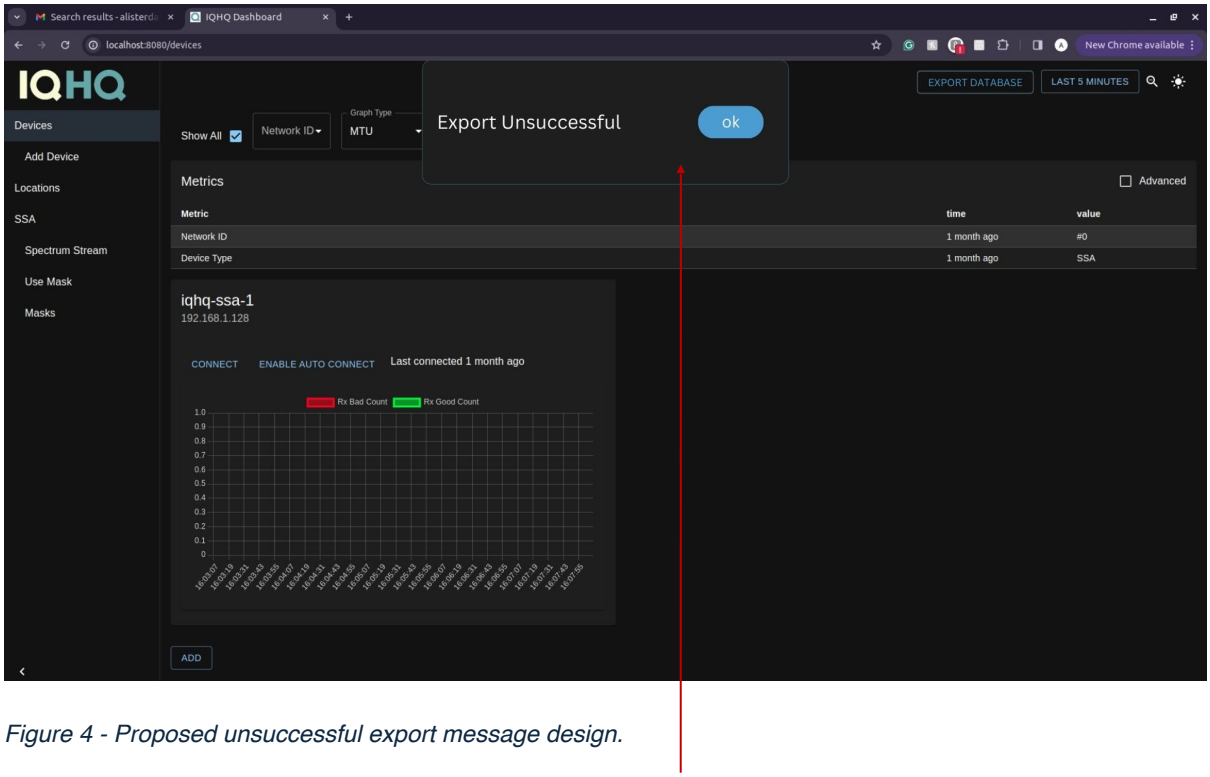


Figure 4 - Proposed unsuccessful export message design.

Proposed Unsuccessful Export Message

UML Diagram:

The UML diagram presented here offers a high-level visualisation of the database export feature within the application. It outlines the sequence of events triggered when a user interacts with the 'Export Database' button, illustrating the flow of control through various components. This diagram serves as a visual aid to understand the interactions and dependencies involved in the database export process, providing stakeholders with a clear overview of the feature's functionality.

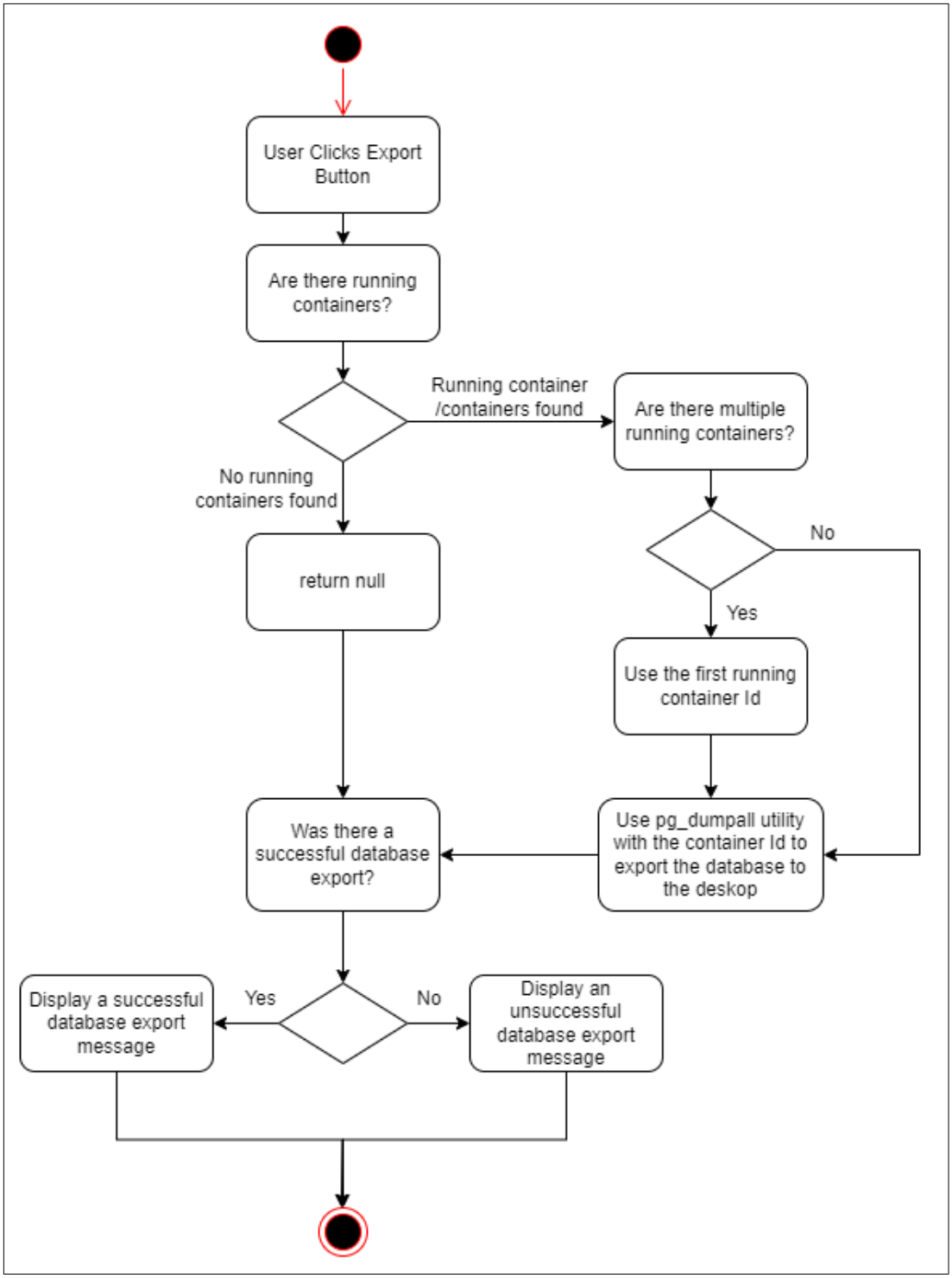


Figure 5 - UML design for the export process.

Pseudo Code:

I included pseudocode in the design documentation to provide a clear roadmap of the steps involved in implementing the database export feature. It serves as a concise outline of the logical flow of the feature, independent of any specific programming language or framework. By detailing the key actions and decision points, pseudocode helps ensure that all stakeholders have a shared understanding of how the feature will function. It also aids in collaboration among developers,

allowing for easier discussion and refinement of the feature's design before actual implementation begins.

Pseudo Code For The Export Function:

```
Define a function getContainerIds(imageName: string):
    Try:
        Execute docker command to get running container IDs based on image name
        Return the container IDs or null if error occurs
    Catch any errors

Define the imageName constant

Get all container IDs for the given image name using getContainerIds function

Split the container IDs into a list

Get the first container ID from the list

If no containers are running:
    Log 'No running containers'
Else if more than one container is running:
    Log 'More than one running container'
    // Define a route.post here
Else:
    Log the running container ID
    Build the command to export PostgreSQL database
    Define a route.post handler to export database:
        Try:
            Execute the export command synchronously
            Send HTTP status OK response
        Catch any errors and send internal server error response
```

Figure 6 - Pseudo code for the export function.

Pseudo Code For The UI:

```
Define a state variable exportSuccess initialized to false

Define a function handleExportSuccess:
    Set exportSuccess to true
    Show an alert with the message "Export Successful"

Define a function handleExportFailure:
    Set exportSuccess to false
    Show an alert with the message "Export Unsuccessful"


Render an HTTPTriggerButton component:
    Set path to "/export"
    Define onComplete event handler:
        If export was successful:
            Call handleExportSuccess function
        Else:
            Call handleExportFailure function
    Set button label as "EXPORT DATABASE"
```

Figure 7 - Pseudo code for the export UI.

```
execSync("docker exec -t <container_id> pg_dumpall -c -U <username> > dump_`date
+%d-%m-%Y' '%H_%M_%S`.sql");
```

Figure 8 - command to pg_dumpall a database.

Development:

Exporting the database:

My first step was to figure out how I could export a PostgreSQL database using Postgres commands. After doing some research, I found that PostgreSQL has a utility called pg_dumpall. pg_dumpall is a utility for writing out ("dumping") all PostgreSQL databases of a cluster into one script file. The script file contains SQL commands that can be used as input to psql to restore the databases. It does this by calling pg_dump for each database in the cluster. pg_dumpall also dumps global objects that are common to all databases, namely database roles, tablespaces, and privilege grants for configuration parameters.

- **execSync:** This is a function from a Node.js library (likely a framework like NestJS) that allows you to execute system commands and capture the output.
- **docker exec:** This command is used to run a process inside a Docker container.
- **-t:** This flag tells docker exec to allocate a pseudo-tty (terminal) for the process being executed. This might be required by some database utilities.
- **<container_id>:** This should be the ID of the Docker container running the PostgreSQL database.
- **pg_dumpall:** This is a command-line utility included with PostgreSQL that is used to create a backup of a database cluster.
- **-c:** tells pg_dumpall to include table data in the backup.
- **-U <username>:** specifies the username to connect to the PostgreSQL server.
- **dump_date +%d-%m-%Y''%H%M_%S.sql:** This part redirects the output of pg_dumpall to a file.

I added a file called ExportFunction.ts, this file will contain the pg_dumpall command needed to export the database. To initially test the command, I needed a couple of variables. I need the relevant docker container ID, as well as the username used to connect to the PostgreSQL server. To find the docker ID I ran the 'docker ps' command in the dashboard repository, which displayed all the container IDs, initially I was not sure which ID to use, therefore since this was a test I used the most recent ID. I opened up pgAdmin to find the username which was set to "postgres" as default. I now had a complete command I could use, to test it out I ran the command in the terminal. The output of the command was a successful database export, I was able to open the SQL file and view the database content in it.

```
alister@AID-039:~$ docker ps
CONTAINER ID        IMAGE                                COMMAND                  CREATED             STATUS
45d560aeb747        timescale/timescaledb:latest-pg12    "docker-entrypoint.s…"   3 months ago        Restarting
eff849e3b786        timescale/timescaledb:latest-pg12    "docker-entrypoint.s…"   3 months ago        Restarting
195dba25e5a7        timescale/timescaledb:latest-pg12    "docker-entrypoint.s…"   3 months ago        Up 5 days
5ff155e1ee71        timescale/timescaledb:latest-pg12    "docker-entrypoint.s…"   3 months ago        Restarting
f15250bdd2b8        timescale/timescaledb:latest-pg12    "docker-entrypoint.s…"   4 months ago        Restarting
alister@AID-039:~$ docker exec -t 195dba25e5a7 pg_dumpall -c -U postgres > dump_`date +%d-%m-%Y'_'%H_%M_%S`.sql
alister@AID-039:~$
```

Figure 9 - Terminal output of running docker ps.

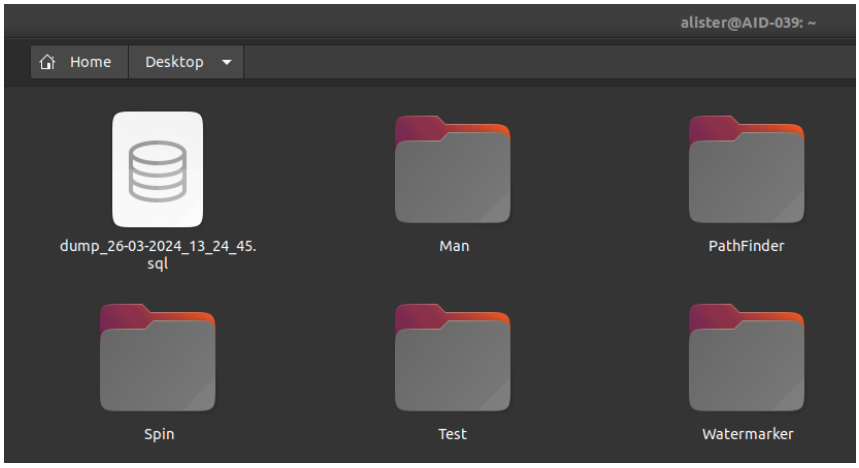


Figure 10 - Database SQL file.

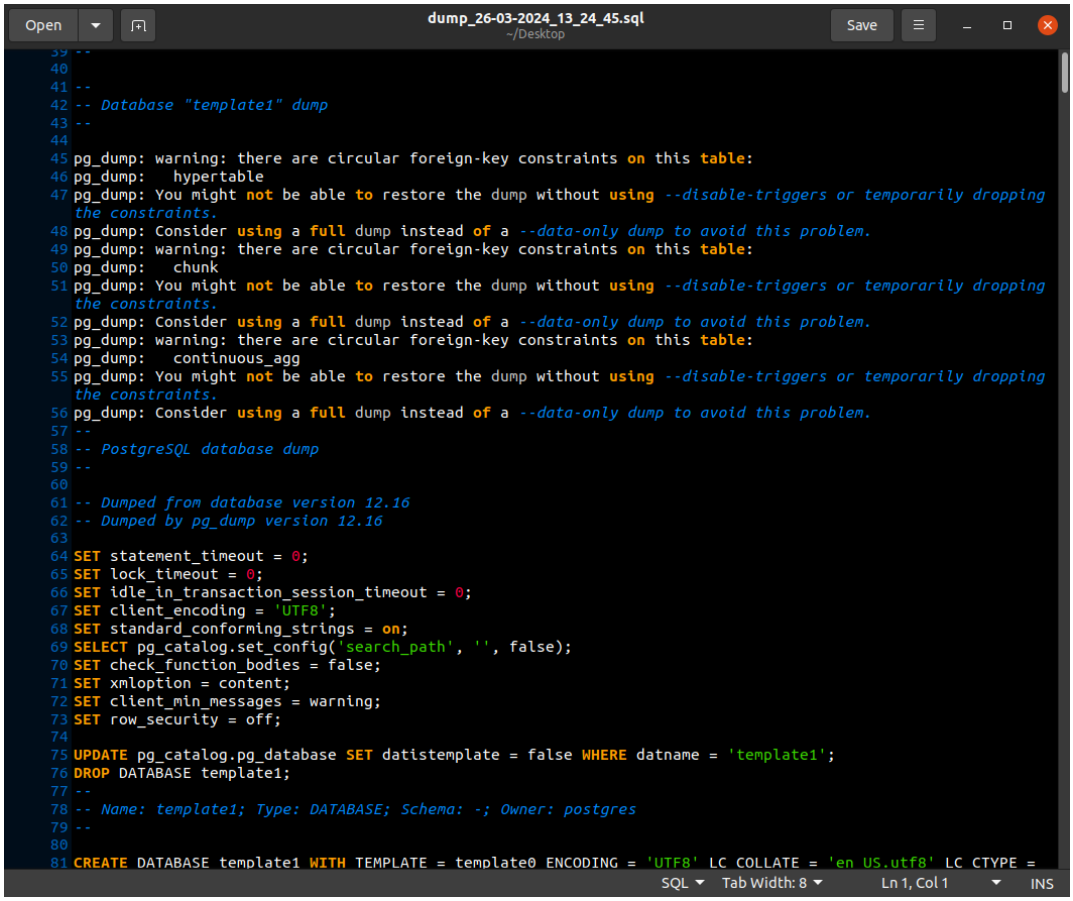


Figure 11 - Contents of database SQL file.

Initial front-end integration of a react button:

To incorporate the export functionality into the dashboard interface, I identified the 'AppToolbar.tsx' file responsible for rendering the toolbar at the top of the dashboard. Recognising its broad applicability for various dashboard features, including the export button, I decided to integrate the export button within this component.

Within the 'AppToolbar.tsx' file, I introduced the export button using the 'HTTPTriggerButton' component, a reusable component designed for handling HTTP requests. Utilising this component provided consistency in user interactions across the dashboard. I configured the export button to trigger a POST request to the '/export' endpoint upon user interaction.

```
63          {!props.isSidebarOpen
64              && <Grid item><IconButton onClick={props.onSidebarOpen}>
65                  <Menu />
66              </IconButton></Grid>
67          }
68          <div style={{
69              flexGrow: 1,
70          }}/>
71
72          <HTTPTriggerButton variant="outlined"
73          style={{marginRight:10, alignSelf: "center"}}
74          key={"Export"}
75          method="post"
76          path={"/export"}>
77          EXPORT</HTTPTriggerButton>
78
79          <Grid item
80              style={{
81                  alignSelf: "center",
82              }}
83          >
84              <ComplexDatePicker />
85          </Grid>
86          {undoLastChange && <Grid item><IconButton
87              onClick={()=>{
88                  undoLastChange()
89              }}
90          >
91              <Undo/>
92          </IconButton></Grid>}
93          <Grid item><IconButton
94              onClick={()=>{
```

Figure 12 - React Button.

The 'HTTPTriggerButton' component encapsulates the necessary logic for sending HTTP requests, including handling loading states and response data. By specifying the appropriate HTTP method (POST) and endpoint ('/export'), I ensured seamless interaction between the frontend and backend components.

To enhance user experience and maintain visual consistency, I positioned the export button alongside other toolbar elements, ensuring intuitive navigation and accessibility for dashboard users.

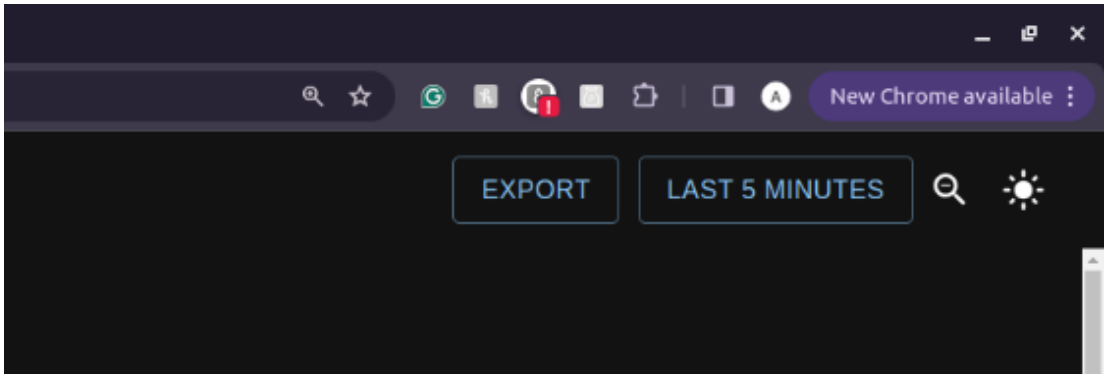


Figure 13 - IQHQ Dashboard.

When the export button is clicked, it triggers the ‘onClick’ event handler defined in the ‘HTTPTriggerButton’ component. At this stage, the button's appearance changes to indicate that an action is being performed. The conditional styling logic within the component checks the response status (‘response.ok === false’) to determine if the button should be displayed in the 'error' colour. Since we have not implemented any HTTP functionality yet, the response status is undefined, resulting in the button turning red to visually indicate an error state.

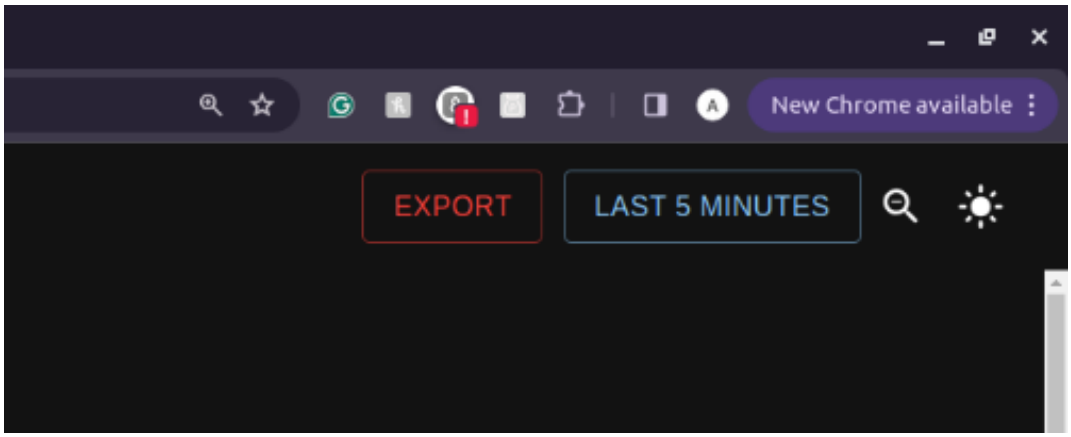


Figure 14 - IQHQ Dashboard.

ExportFunction File:

I added a file called ‘ExportFunction.ts’ to encapsulate the logic responsible for exporting the PostgreSQL database. This file serves as a dedicated module for handling export-related functionality within our application. To establish a route for handling export requests, I began by setting up a ‘route.post()’ method within the file. This method defines a route that listens for HTTP POST requests sent to a specific endpoint, allowing us to trigger the export process when a POST request is received.

Within the POST route handler, I incorporated the necessary Docker command to export the database using the execSync() function from the Node.js child_process module. This command

utilises the docker exec utility to execute a command within the Docker container running the PostgreSQL database.

Overall, the ‘route.post()’ method, combined with the Docker command, enables us to handle export requests and execute the necessary operations for exporting the database within our Express application. This approach provides a structured and maintainable solution for integrating export functionality into our application architecture.

src > routes > TS ExportFunction.ts > default

```ts
import { Router } from "express";
import networkManager from "../NetworkManagerInstance";
import { Interval } from "../NetworkManager/ConnectionRepo";
import ExpressBodyTypeCheck from "./ExpressBodyTypeCheck";
import * as yup from "yup";
import HTTPStatus from "http-status-codes";
import { execSync } from "child_process";

const route = Router();

route.post("/", async (req, res)=>{
    try{
        console.log('Starting export process...');
        execSync("docker exec -t f15250bdd2b8 pg_dumpall -c -U postgres > dump_`date +%d-%m-%Y'_'%H_%M_%S`.sql");
        console.log('Export function finished.');
        res.status(HTTPStatus.OK).send();
    }catch(e) {
        res.status(HTTPStatus.INTERNAL_SERVER_ERROR)
            .send(e)
    }
})

export default route;
```

Figure 15 - ExportFunction.ts file containing pg_dumpall command.

I added an import statement to include the ‘ExportFunction’ module in the ‘index.ts’ file. This import allows us to use the export functionality defined in ExportFunction.ts within the main route file.

Next, I included a new route for the export functionality using ‘route.use("/export", ExportFunction)’; This line establishes a route endpoint /export that is handled by the ‘ExportFunction’ module. Now, when a request is made to the /export endpoint, the corresponding logic in the ExportFunction.ts file will be executed to handle the request.

By adding the export route to the ‘index.ts’ file, we ensure that the export functionality is integrated into the overall API routes of our application.

src > routes > TS index.ts > ...

```ts
import { Router, RequestHandler } from "express";
import * as express from "express";

import cors from 'cors'

import ActiveDeviceRoute from "./ActiveDevice"
import DeviceRoute from "./Devices"
import DeviceGroupRoute from "./DeviceGroup"
import MetricsRoute from "./Metrics"
import DisplayInfo from "../DisplayInfo";
import ExportFunction from "./ExportFunction";

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 16 - index.ts file containing routes.

Initial Testing:

For the Initial Testing phase, I extended the functionality of the existing export button in the AppToolbar.tsx file by adding an ‘onComplete’ event handler to the HTTPTriggerButton component. This event handler was configured to log 'Button Clicked' to the console upon successful completion of the export operation. After starting up the dashboard, I proceeded to test the export functionality. Upon clicking the export button, the expected behaviour occurred: the message 'Button Clicked' was logged to the Google Console, confirming that the button click event and export process was successfully executed.

```
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 17 - Updated react button.

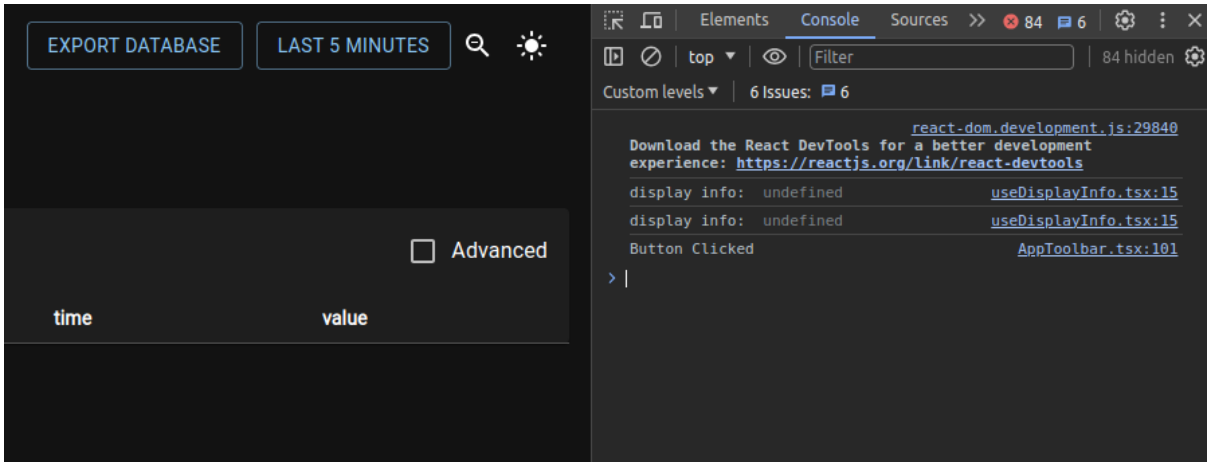


Figure 18 - IQHQ Dashboard showing console output of clicking the export database button.

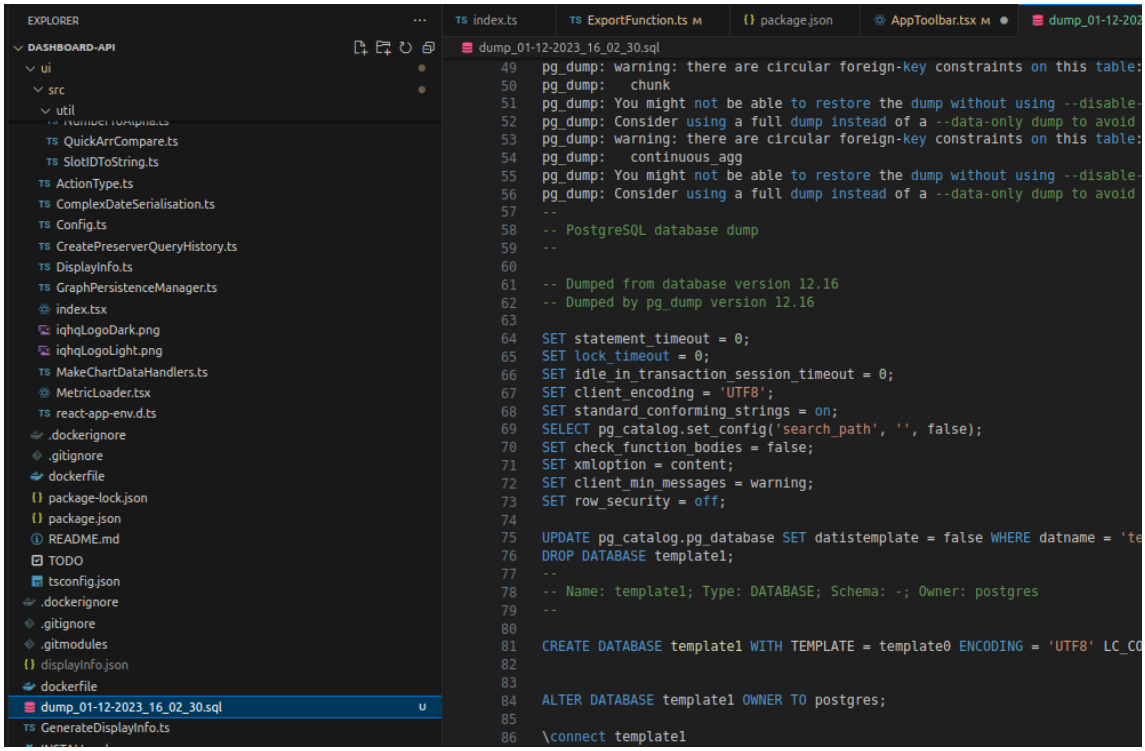


Figure 19 - Exported database SQL file.

While the initial testing yielded positive results in terms of capturing the button click event and initiating the export process, it also highlighted areas for refinement. One notable improvement involves exporting the database to a more user-friendly location, such as the desktop, to enhance accessibility and convenience for users. This adjustment would involve a straightforward implementation to redirect the exported database file to the desired location.

Additionally, it became apparent that there is a need for user interface feedback to indicate the start and end of the export process. Currently, there is no visual indication that the export operation is underway once the button is clicked. Incorporating UI messages or notifications would enhance the user experience by providing clear feedback about the status of the export process, ensuring users are informed and aware of ongoing operations.

Docker Issues:

I ran into issues exporting the database, since all of a sudden it seemed to not work. After speaking to software team, 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.ts' file that will only use the running container Id. I added a function called 'getContainerIDs' to the 'ExportFunction.ts' file that would return the running IDs using the relevant image name.

```
// 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;
    }
}
```

Figure 20 - getContainerIDs function.

The getContainerIDs function is designed to retrieve the IDs of running Docker containers associated with a specified image name. Within a try block, it constructs a Docker command tailored to filter containers by the provided image name and their running status. The execSync function executes this command synchronously, capturing the output containing container IDs. The output is then trimmed to remove any whitespace, and the resulting container ID or null is returned. If an error occurs during execution, such as failure to execute the Docker command, the catch block handles it by logging an error message and returning null.

```
18  const imageName = 'timescale/timescaledb:latest-pg12';
19  const containerIDs = getContainerIDs(imageName);
20
21  console.log("Running Containers:", containerIDs);
```

Figure 21 - console logging containerIDs.

```
alister@AID-039: /o...   alister@AID-039: /o...
> ts-node ./src/index.ts

PGHOST=localhost
PGUSER=postgres
PGDATABASE=postgres
PGPASSWORD=[redacted]
PGPORT=5432
AUTOCONNECT_INTERVAL=30
SERVER_PORT=3001
SERVER_PUBLIC_ROOT=./public
TILE_SERVER_ADDR=a.tile.openstreetmap.org
DISPLAY_INFO_PATH=./displayInfo.json
Running Containers: 195dba25e5a7
```

Figure 22 - output of console logging containerIDs.

Testing the function proved successful, now I can use the function to return the running container Id and use the Id in the pg_dumpall command to dump the right database.

```
const execCommand = `docker exec -t ${containerIDs} pg_dumpall -c -U postgres > ~/Desktop/dump_\`date +%d-%m-%Y'_'%H_%M_%S\`.sql`;
```

Figure 23 - using containerIDs in pg_dumpall command.

Handling multiple running docker containers:

Anticipating the possibility of multiple running Docker containers, I devised a strategy to handle this scenario effectively. Upon retrieving the container IDs using the 'getContainerIDs' function, I recognised that if more than one container is running, the function returns a string containing all the IDs. To address this, I implemented a solution that splits the 'containerIDs' string into a list, allowing for individual processing of each ID. By examining the length of this list, I could ascertain whether there were no running containers or if multiple instances were detected. This crucial step ensured that appropriate actions could be taken based on the specific circumstances. Additionally, logging mechanisms were implemented to alert users about the presence of multiple containers, enabling clear communication and proactive resolution of potential issues. This approach enhances the reliability and robustness of the Docker container management process, safeguarding against unexpected complications during database export operations.

```
const imageName = 'timescale/timescaledb:latest-pg12';
const containerIDs = getContainerIDs(imageName);

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

//Check to see if either more than one running container is found or no container is found
if (checkIDs.length === 0) {
    console.log('No running containers');
} else if (checkIDs.length > 1) {
    console.log('More than one running container');
    //some sort of route.post is needed here
} else {
    console.log('Looks good')
    const execCommand = `docker exec -t ${containerIDs} pg_dumpall -c -U postgres > ~/Desktop/dump_\`date +%d-%m

    // 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.
    // The export process is logged, and the response indicates success or failure.
    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);
        }
    });
}

```

Figure 24 - Code to check for either multiple running containers or no running containers.

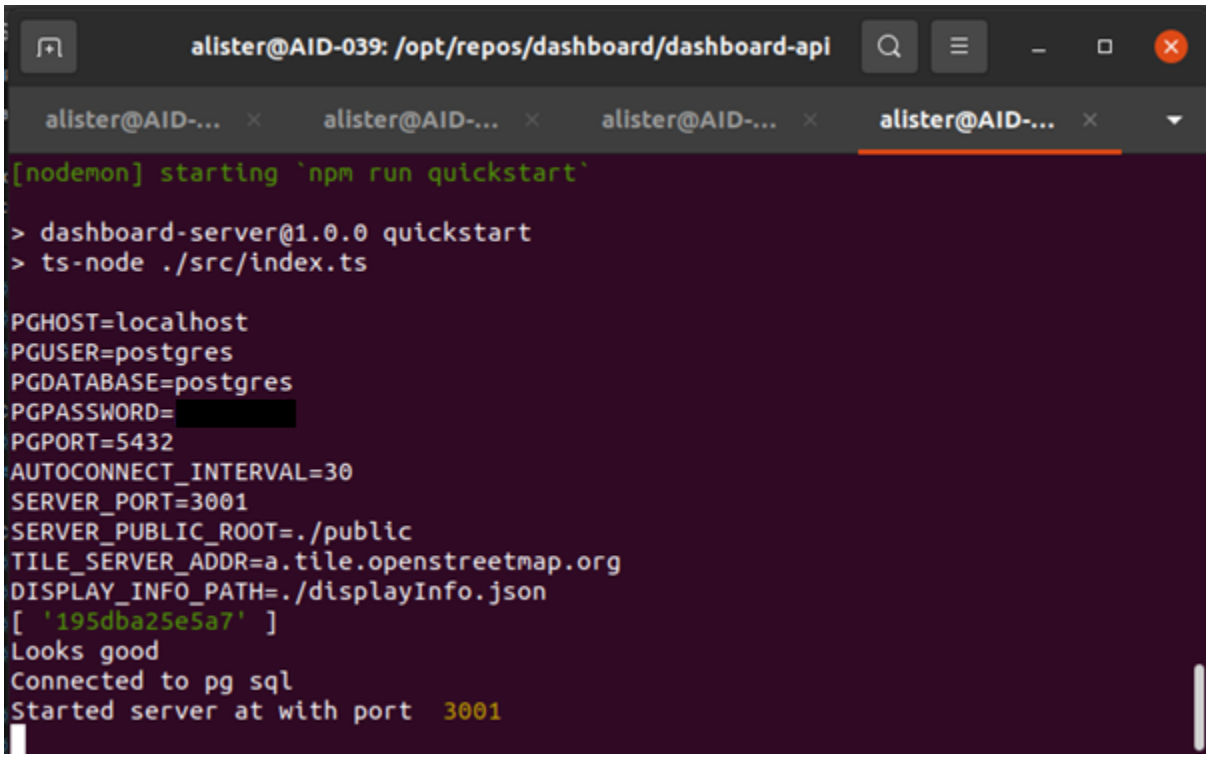


Figure 25 - Output of checking for multiple running containers.

Updating Database Export Command for File Path Location:

To ensure that the exported database file is stored in an appropriate and easily accessible location, I modified the export command to specify the Desktop directory as the destination. By appending the > ~/Desktop/ prefix to the command, I directed the output of the ‘pg_dumpall’ operation to the user's Desktop folder. This adjustment streamlines the file management process, allowing users to quickly locate and access the exported database file directly from their desktop environment. Additionally, it enhances the user experience by providing a clear and intuitive destination for the exported data.

Handling export success/failure using useState in React:

To provide feedback to the user regarding the success or failure of the database export operation, I integrated the ‘alert’ function into the application. This involved setting up a ‘useState’ hook named ‘exportSuccess’, initialised to ‘false’, to track the export status. Subsequently, I created two functions: ‘handleExportSuccess’, responsible for updating ‘exportSuccess’ to ‘true’ and displaying the "Export Successful" alert, and ‘handleExportFailure’, which sets ‘exportSuccess’ to `false` and triggers the "Export Unsuccessful" alert. Within the button's ‘onComplete’ callback, I implemented a check to determine whether the export operation was successful (‘ok’ parameter). If successful, ‘handleExportSuccess’ is invoked; otherwise, ‘handleExportFailure’ is triggered. This approach ensures that the user receives immediate feedback on the outcome of the export process.

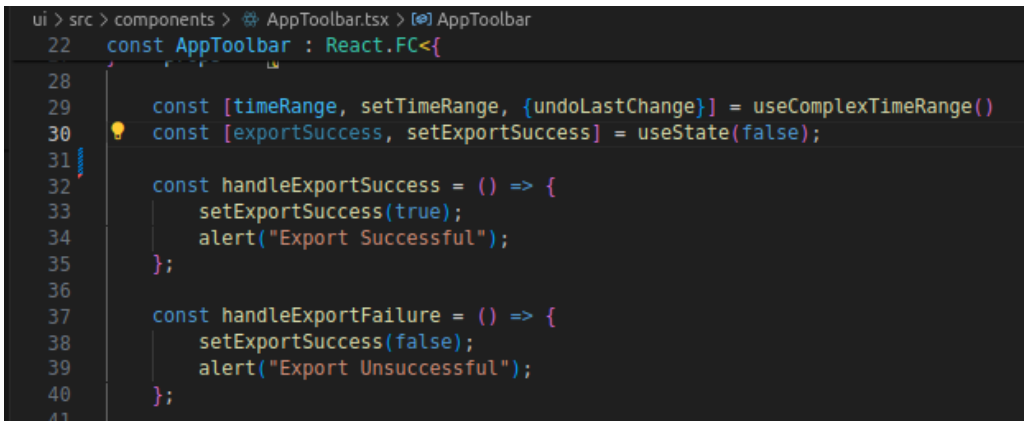


Figure 26 - React useState for handling a successful/unsuccessful database export.

Figure 27 - Updated react button using useState.

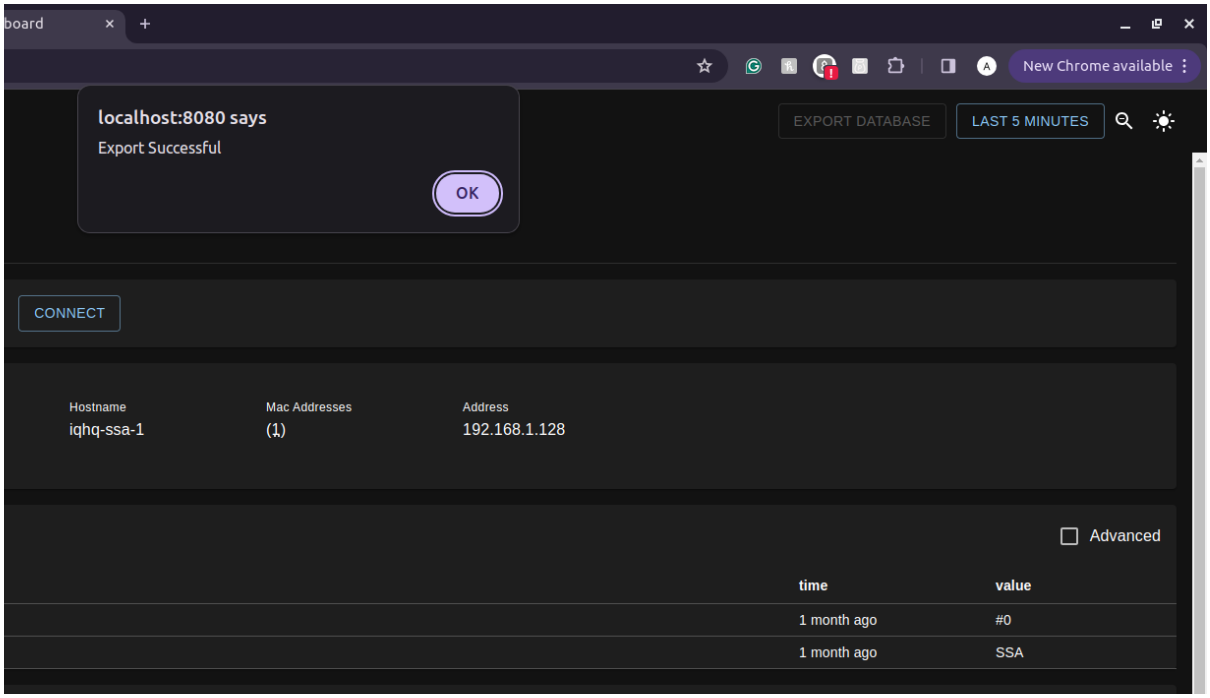


Figure 28 - IQHQ Dashboard showing the result of a successful database export.

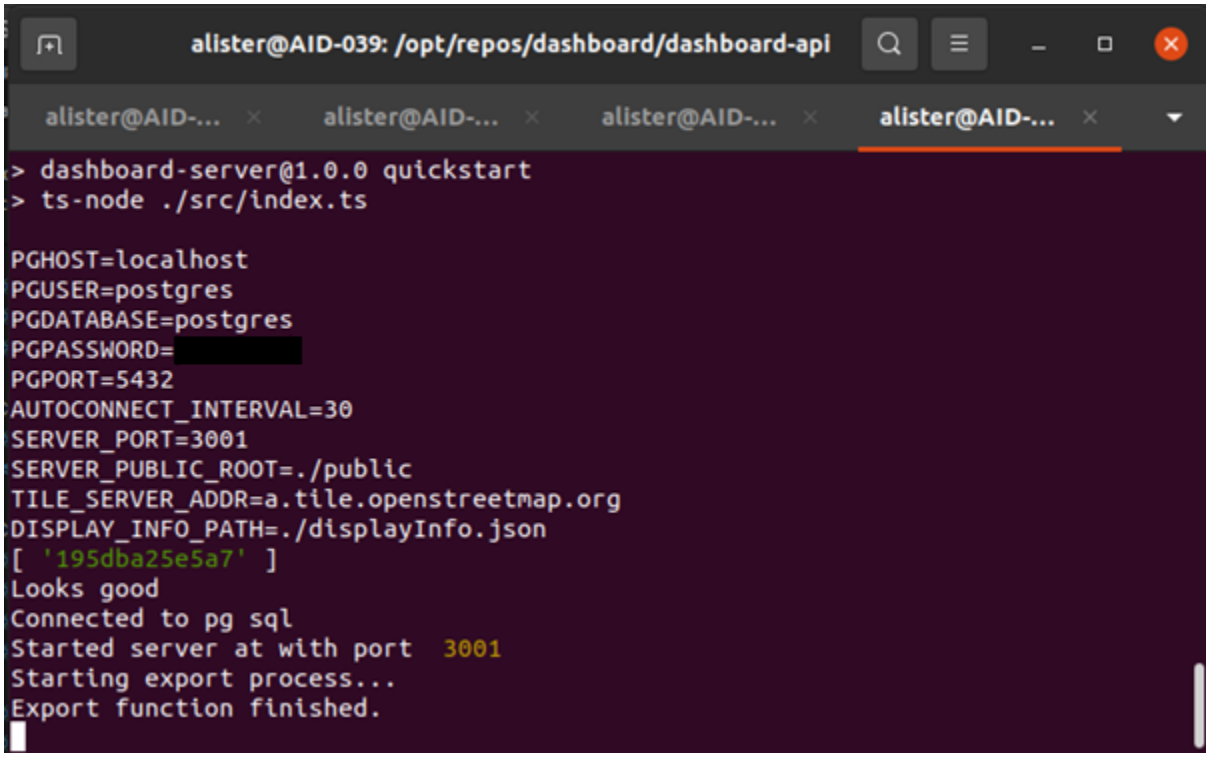


Figure 29 - Output as a result of a successful database export.

Successful Implementation of Export Functionality:

With the completion of the code development stage, the program now functions as intended according to the task description. Users can seamlessly export the PostgreSQL database by clicking the designated button on the dashboard's toolbar. The integration of alert notifications ensures that users receive immediate feedback on the success or failure of the export operation, enhancing the overall user experience. Additionally, the implementation of error handling and status checks ensures the robustness and reliability of the export functionality.

Testing:

Unit Testing:

Unit testing involves testing individual units or components of code in isolation to ensure they function correctly. It is a critical part of software development, helping identify and address issues early. By testing units independently, I can maintain code quality, promote good design practices, and ensure the reliability of the final product. My approach to unit testing involves systematically reviewing each file that I have modified or added and conducting thorough testing on the relevant components.

ExportFunction.ts:

getContainerIDs(imageName: string): string | null

```
// 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;
    }
}
```

Figure 30 - getContainerIDs function.

Test Case 1: valid imageName

Input: imageName = ‘timescale/timescaledb:latest-pg12’

Expected Output: 195dba25e5a7

Description: Test the unit with a valid image name to make sure it returns a valid running container Id.

Actual Output: 195dba25e5a7

Evaluation: This test case has passed.

Test Case 2: Invalid imageName

Input: imageName = ‘test’

Expected Output: null

Description: Test the unit with an invalid image name to make sure it returns null.

Actual Output: null

Evaluation: This test case has passed.

Test Case 3: Non string imageName

Input: imageName = 123

Expected Output: Typescript compilation error

Description: Test the unit with a non string value.

Actual Output: TS:Error: X Unable to compile TypeScript, Argument of type ‘number’ is not assignable to parameter of type ‘string’.

Evaluation: This was the expected output; however, it is very unlikely this error would occur.

A check to see if there is either no running container or multiple running containers.

```
const imageName = 'timescale/timescaledb:latest-pg12';
const containerIDs = getContainerIDs(imageName);

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

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

Figure 31 - Code that checks if there are multiple or no running containers.

Test Case 1: No running container

Input: imageName is set to 'test', indicating an invalid image name.

Expected Output: An empty array ([]) and the message 'No running containers'.

Description: Verify that when the provided image name does not correspond to any running containers, the application correctly identifies the absence of running containers and displays the message 'No running containers'.

Actual Output: An empty array ([]) and the message 'No running containers'.

Evaluation: This unit test has passed as the application correctly handles the scenario of no running containers.

Test Case 2: Single running container

Input: imageName is set to 'timescale/timescaledb:latest-pg12', representing a valid image name with a single running container.

Expected Output: An array containing the single running container ID (['195dba25e5a7']) and the message 'Looks good'.

Description: Validate that when there is only one running container corresponding to the provided image name, the application identifies the single container and confirms its presence with the message 'Looks good'.

Actual Output: An array containing the single running container ID (['195dba25e5a7']) and the message 'Looks good'.

Evaluation: This unit test has passed as the application successfully detects and acknowledges the presence of a single running container.

Test Case 3: Multiple running containers

Input: checkIDs is set to an array of multiple running container IDs (['45d560aeb747', 'eff849e3b786', '5ff155e1ee71']).

Expected Output: The same array of multiple running container IDs (['45d560aeb747', 'eff849e3b786', '5ff155e1ee71']) and the message 'More than one running container'.

Description: Ensure that when multiple running containers are found for the provided image name, the application identifies the situation and notifies the user about the presence of multiple containers with the message 'More than one running container'.

Actual Output: The same array of multiple running container IDs (['45d560aeb747', 'eff849e3b786', '5ff155e1ee71']) and the message 'More than one running container'.

Evaluation: This unit test has passed; however, an issue has been identified in the code. The execCommand variable references containerIDs, which does not handle cases of multiple running containers correctly. To resolve this, the code should be updated to use a single container ID from the checkIDs list instead of containerIDs. Additionally, I have improved clarity by renaming variables for better readability.

Updated check to see if there is either no running container or multiple running containers.

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

let containerList = allContainerIds ? allContainerIds.split("\n") : [];
const containerId = 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(`Running Container Found: ${containerList[0]}`)
    //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`;
```

Figure 32 - Updated code that checks if there are multiple or no running containers.

AppToolBar.tsx:

```
30      const [exportSuccess, setExportSuccess] = useState(false);
31
32      /*
33      const [multipleContainer, setMultipleContainer] = useState(false);
34
35      const handleMultipleContainer = () => {
36          setMultipleContainer(true);
37          alert("Multiple Running Containers");
38      }
39      */
40
41      const handleExportSuccess = () => {
42          setExportSuccess(true);
43          alert("Export Successful");
44      };
45
46      const handleExportFailure = () => {
47          setExportSuccess(false);
48          alert("Export Unsuccessful");
49      };
```

Figure 33 - Code for handling a successful/unsuccessful export.

```
<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 34 - Code that implements export handler.

Handling Export Success/Failure:

Test Case 1: Successful Export

Input: ok is set to true, simulating a successful export operation.

Expected Output: The application should display a React alert with the message "Export Successful".

Description: Verify that when the export operation completes successfully, the application notifies the user with an alert indicating a successful export.

Actual Output: React alert displays the message "Export Successful".

Evaluation: This unit test has passed as the application successfully alerts the user about the successful export operation.

Test Case 2: Unsuccessful Export

Input: ok is set to false, simulating an unsuccessful export operation.

Expected Output: The application should display a React alert with the message "Export Unsuccessful".

Description: Validate that when the export operation fails, the application informs the user with an alert indicating an unsuccessful export.

Actual Output: React alert displays the message "Export Unsuccessful".

Evaluation: This unit test has passed as the application appropriately notifies the user about the failed export operation.

Integration Testing:

Integration testing ensures that the various components of a system function together as expected. In this section, I will validate the interaction and collaboration between different modules, services, and layers of the application to ensure seamless functionality across the entire system. This includes testing API endpoints, frontend-backend interaction, database integration, external service integration, error handling, cross-browser compatibility, security, and performance.

UI Component Interaction:

Ensure that the UI components (such as buttons) behave as expected and trigger the appropriate actions when clicked or interacted with.

Test Case 1: Button Click Interaction

Description: Verify that clicking the "Export Database" button triggers the database export process.

Steps:

1. Open the application UI.
2. Locate the "Export Database" button.
3. Click the button.

Expected Result: The database export process initiates, and the user is provided with feedback indicating that the export is in progress.

Actual Result: The database export process initiated, and the user was provided with feedback indicating that the export is in progress.

Evaluation: This test was a success.

Test Case 2: Success Alert Display

Description: Ensure that upon successful database export, a success alert message is displayed to the user.

Steps:

1. Trigger a successful database export process.

Expected Result: The user receives a success alert message confirming that the export has been completed successfully.

Actual Result: The user received a success alert message confirming that the export has been completed successfully.

Evaluation: This test was a success.

Test Case 3: Failure Alert Display

Description: Validate that in case of a failed database export, an error alert message is presented to the user.

Steps:

1. Simulate a failed database export process.

Expected Result: The user receives an error alert message indicating that the export process encountered an issue.

Actual Result: The user received an error alert message indicating that the export process encountered an issue.

Evaluation: This test was a success.

Database Export:

Validate that the database export functionality works correctly, including exporting the data to the designated location on the desktop.

Test Case 1: Successful Database Export

Description: Verify that the database export process executes successfully, and the data is exported to the designated location on the desktop.

Steps:

1. Trigger the database export process.
2. Check the designated location on the desktop for the exported database file.

Expected Result: The database export process completes successfully, and the exported file is found at the specified location on the desktop.

Actual Result: The database export process completed successfully, and the exported file was found at the specified location on the desktop.

Evaluation: This test was a success.

Test Case 2: Export File Existence

Description: Ensure that the exported database file exists after the export process.

Steps:

1. Trigger the database export process.
2. Check if the exported file exists at the designated location on the desktop.

Expected Result: The exported file exists at the specified location.

Actual Result: The exported file exists at the specified location.

Evaluation: This test was a success.

Test Case 3: Export Process Feedback

Description: Validate that appropriate feedback is provided to the user during the database export process.

Steps:

1. Trigger the database export process.
2. Observe the application interface or any provided feedback mechanism.

Expected Result: The user receives feedback indicating the progress of the export process, such as a loading indicator or success message.

Actual Result: The user received feedback indicating the progress of the export process, such as a loading indicator or success message.

Evaluation: This test was a success.

Error Handling:

Verify that the application handles errors gracefully, displaying meaningful error messages to the user in case of failures or exceptions.

Test Case 1: Docker Execution Error

Description: Verify that appropriate error handling is in place when there is an issue with executing Docker commands.

Steps:

1. Simulate an error in executing a Docker command, such as a syntax error or a command failure.

Expected Result: The application displays an error message indicating the failure to execute the Docker command.

Actual Result: The application displays an error message indicating the failure to execute the Docker command.

Evaluation: This test was a success.

Test Case 2: Database Export Failure

Description: Validate that the application responds correctly when there is a failure in exporting the database.

Steps:

1. Trigger a database export process that intentionally fails, such as by providing incorrect database credentials.

Expected Result: The application provides an error message informing the user of the database export failure.

Actual Result: The application provides an error message informing the user of the database export failure.

Evaluation: This test was a success.

Test Case 3: General Exception Handling

Description: Ensure that the application handles unexpected exceptions gracefully.

Steps:

1. Introduce an unexpected exception in the application code, such as a runtime error.

Expected Result: The application presents an error message explaining the issue and provides guidance on how to proceed.

Actual Result: The application presents an error message explaining the issue and provides guidance on how to proceed.

Evaluation: This test was a success.

API Endpoints:

Test the API endpoints used for database export to ensure they handle requests properly and return the expected responses.

Test Case 1: Export Endpoint Availability

Description: Verify that the export endpoint is accessible and responds correctly to HTTP requests.

Steps:

1. Send a GET or POST request to the export endpoint ("/export").

Expected Result: The endpoint responds with a status code indicating success (e.g., 200 OK) or an appropriate error status code (e.g., 404 Not Found).

Actual Result: The endpoint responds with the expected status code.

Evaluation: This test was a success.

Test Case 2: Export Request Handling

Description: Ensure that the export endpoint properly handles export requests and initiates the database export process.

Steps:

1. Send a POST request to the export endpoint ("/export").
2. Verify that the database export process is triggered.

Expected Result: The export endpoint initiates the database export process as expected.

Actual Result: The export endpoint successfully initiates the database export process.

Evaluation: This test was a success.

Test Case 3: Response Validation

Description: Validate the format and content of the response returned by the export endpoint.

Steps:

1. Send a POST request to the export endpoint ("/export").
2. Receive the response from the endpoint.

Expected Result: The response contains the appropriate data, such as success or error messages, in the expected format (e.g., JSON).

Actual Result: The response contains the expected data in the correct format.

Evaluation: This test was a success.

Frontend-Backend Interaction:

Validate the communication between the frontend and backend systems, ensuring that data is transmitted correctly, and actions are executed as intended.

Test Case 1: Data Transmission

Description: Confirm that data is transmitted accurately from the frontend to the backend during the database export process.

Steps:

1. Trigger the database export action from the frontend.

2. Monitor the data sent to the backend.

Expected Result: The relevant data, such as export parameters or user inputs, is successfully transmitted to the backend.

Actual Result: The frontend correctly transmits the required data to the backend.

Evaluation: This test was a success.

Test Case 2: Action Execution

Description: Ensure that actions initiated by the frontend are executed correctly by the backend.

Steps:

1. Trigger a database export action from the frontend.
2. Monitor the backend to verify the execution of the export process.

Expected Result: The backend correctly executes the requested action, such as exporting the database, in response to frontend input.

Actual Result: The backend successfully executes the requested action triggered by the frontend.

Evaluation: This test was a success.

Test Case 3: Response Handling

Description: Validate the frontend's handling of responses received from the backend.

Steps:

1. Trigger a database export action from the frontend.
2. Receive and process the response returned by the backend.

Expected Result: The frontend appropriately interprets and displays the response received from the backend, such as success or error messages.

Actual Result: The frontend correctly handles and displays the response received from the backend.

Evaluation: This test was a success.

Performance Testing:

Performance testing aims to evaluate how well the system performs under different scenarios, such as varying load levels or database sizes.

Objective:
The objective of performance testing is to assess the efficiency and responsiveness of the database export feature under various conditions. This includes measuring the time taken to

complete the export process, evaluating system resource utilisation (such as CPU and memory usage), and analysing network bandwidth usage. The goal is to ensure that the feature meets performance requirements, maintains acceptable response times, and can handle expected workloads without degradation in performance.

Test Environment:

Hardware Specifications:

- **Device Name:** AID-085
- **Processor:** 12th Gen Intel(R) Core(TM) i5-1235U @ 1.30 GHz
- **Installed RAM:** 8.00 GB (7.64 GB usable)
- **System Type:** 64-bit operating system, x64-based processor
- **Operating System:** Windows 10 Pro, Version 22H2 (Build 19045.4170)
- **Network:** Wi-Fi 5 (802.11ac), Network band: 5 GHz

Database Management System:

- **Database:** PostgreSQL
- **Management Tool:** pgAdmin4

Network Configuration:

- **SSID:** N.E.R.D.
- **Security Type:** WPA2-Enterprise
- **Network band:** 5 GHz
- **Network channel:** 36
- **Link speed (Receive/Transmit):** 400/400 (Mbps)
- **Manufacturer:** Realtek Semiconductor Corp.
- **Description:** Realtek RTL8852BE WiFi 6 802.11ax PCIe Adapter

Performance Testing Details

Software Dependencies:

- Node.js
- npm packages
- Docker

Browser Compatibility:

- Chrome
- Firefox
- Edge

Database Size and Schema:

- Current database size: 51 MB
- Schema:
  - **device_manager_group**: Stores information about device manager groups.
  - **devices**: Contains details of devices registered in the system.
  - **file_event**: Tracks events related to file operations.
  - **fusion_reports**: Stores fusion reports generated by the system.
  - **local_device_info**: Stores information specific to local devices.
  - **metrics**: Stores general metrics data.
  - **number_metrics**: Stores numeric metrics data.
  - **ssa_masks**: Contains data related to SSA masks.
  - **ssa_metrics**: Stores SSA metrics data.

- Note: The database size used for testing may vary to simulate different scenarios. During testing, various database sizes, such as 100 MB and 200 MB, will be used to assess system performance under different data volumes.

Test Scenarios:

- Simulate various database export scenarios, including small and large database sizes.
- Evaluate performance under different load levels and concurrent user interactions.

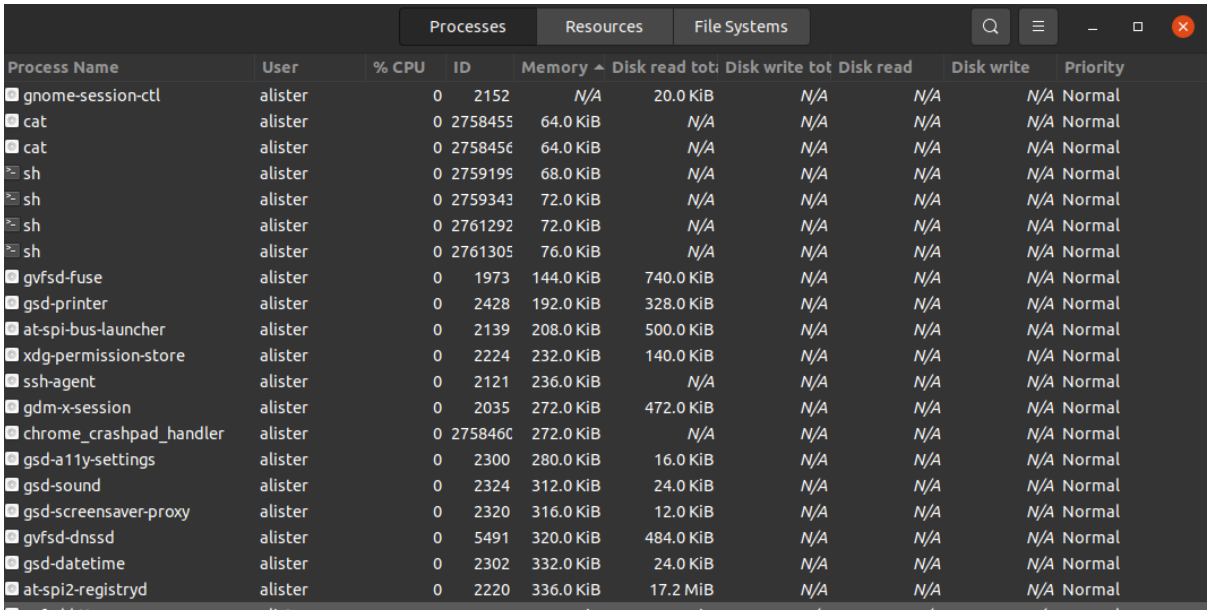
Performance Testing Tools:

- Custom script for measuring speed:

```
    // 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...');
            const startTime = performance.now();
            execSync(execCommand);
            const endTime = performance.now()
            console.log('Export function finished.');
            console.log('Time Taken:',(endTime-startTime)/1000,'s')
            res.status(HTTPStatus.OK).send();
        } catch (e) {
            res.status(HTTPStatus.INTERNAL_SERVER_ERROR).send(e);
        }
    });
}
```

Figure 35 - Custom code to measure the time taken for the export process.

- Ubuntu System Monitor for monitoring memory usage.



Process Name	User	% CPU	ID	Memory	Disk read tot	Disk write tot	Disk read	Disk write	Priority
gnome-session-ctl	alister	0	2152	N/A	20.0 KiB	N/A	N/A	N/A	Normal
cat	alister	0	2758455	64.0 KiB	N/A	N/A	N/A	N/A	Normal
cat	alister	0	2758456	64.0 KiB	N/A	N/A	N/A	N/A	Normal
sh	alister	0	2759195	68.0 KiB	N/A	N/A	N/A	N/A	Normal
sh	alister	0	2759343	72.0 KiB	N/A	N/A	N/A	N/A	Normal
sh	alister	0	2761292	72.0 KiB	N/A	N/A	N/A	N/A	Normal
sh	alister	0	2761305	76.0 KiB	N/A	N/A	N/A	N/A	Normal
gvfsd-fuse	alister	0	1973	144.0 KiB	740.0 KiB	N/A	N/A	N/A	Normal
gsd-printer	alister	0	2428	192.0 KiB	328.0 KiB	N/A	N/A	N/A	Normal
at-spi-bus-launcher	alister	0	2139	208.0 KiB	500.0 KiB	N/A	N/A	N/A	Normal
xdg-permission-store	alister	0	2224	232.0 KiB	140.0 KiB	N/A	N/A	N/A	Normal
ssh-agent	alister	0	2121	236.0 KiB	N/A	N/A	N/A	N/A	Normal
gdm-x-session	alister	0	2035	272.0 KiB	472.0 KiB	N/A	N/A	N/A	Normal
chrome_crashpad_handler	alister	0	275846(	272.0 KiB	N/A	N/A	N/A	N/A	Normal
gsd-a11y-settings	alister	0	2300	280.0 KiB	16.0 KiB	N/A	N/A	N/A	Normal
gsd-sound	alister	0	2324	312.0 KiB	24.0 KiB	N/A	N/A	N/A	Normal
gsd-screensaver-proxy	alister	0	2320	316.0 KiB	12.0 KiB	N/A	N/A	N/A	Normal
gvfsd-dnssd	alister	0	5491	320.0 KiB	484.0 KiB	N/A	N/A	N/A	Normal
gsd-datetime	alister	0	2302	332.0 KiB	24.0 KiB	N/A	N/A	N/A	Normal
at-spi2-registryd	alister	0	2220	336.0 KiB	17.2 MiB	N/A	N/A	N/A	Normal

Figure 36 - Ubuntu system monitor to measure the memory usage.

Performance Metrics:

- Time taken for the export process to finish.
- Memory usage of the export process.

Testing Environment Setup:

1. Ensure that the hardware and software dependencies are properly installed and configured.
2. Set up the test database with realistic data and schema.
3. Configure performance testing tools to simulate load and monitor system metrics.
4. Execute performance tests according to predefined scenarios.
5. Collect and analyse performance metrics to identify bottlenecks and optimise system performance.

		Time (s)				Memory Usage (KiB)			
Test	**Database Size (MB)**	**Time 1**	**Time 2**	**Time 3**	**Average Time**	**Memory Usage 1**	**Memory Usage 2**	**Memory Usage**	**Average Memory Usage**
1	50	1.87	1.92	1.79	1.86	72	72	73	72.33
2	100	4.41	4.41	4.66	4.49	76	72	72	73.33
3	150	7.32	7.50	7.62	7.48	75	76	72	74.33
4	200	10.52	10.60	10.13	10.42	76	72	74	74.00
5	250	12.96	13.37	12.88	13.07	72	72	74	72.67
6	300	15.89	16.10	15.60	15.86	72	72	75	73.00

7	350	18.54	18.66	18.64	18.61	72	72	76	73.33
8	400	21.43	22.06	21.26	21.58	72	74	68	71.33
9	450	23.77	23.52	24.02	23.77	72	76	72	73.33
10	500	28.47	27.25	26.68	27.47	76	72	76	74.67

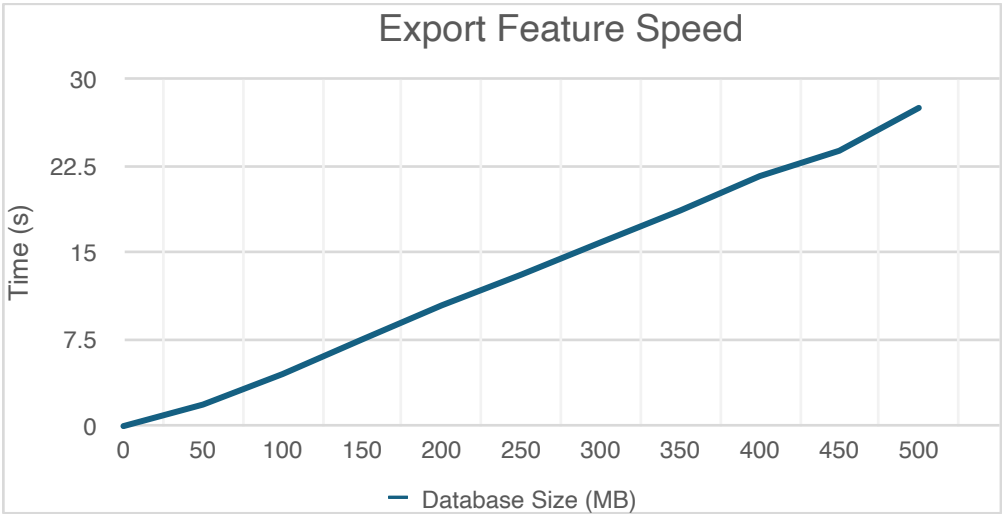


Figure 37 - Chart that shows the relationship between database size and export time.

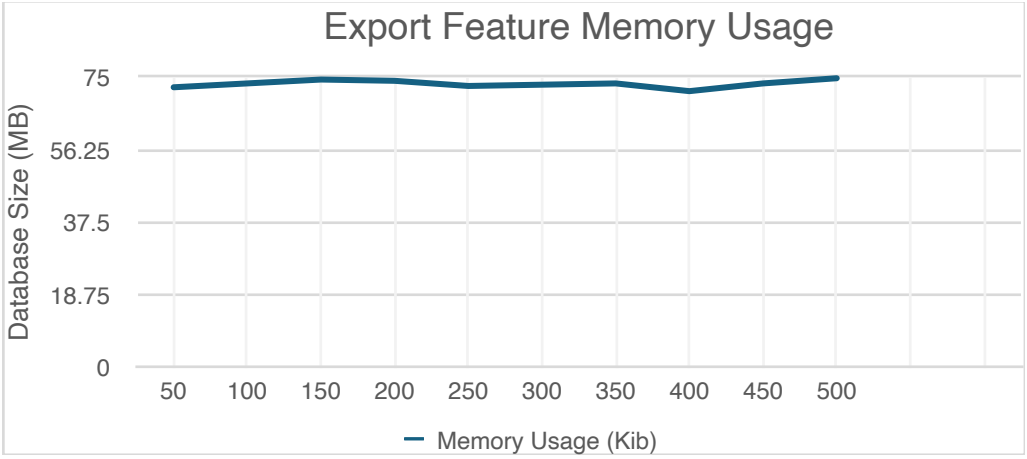


Figure 38 - Chart that shows the relationship between database size and memory usage.

Performance Test Evaluation:

In evaluating the performance test results, it is evident that the time taken for the database export process exhibited a consistent trend of increase as the database size increased. This observation aligns with expectations, as larger databases naturally require more time for export due to the greater volume of data involved. Despite this increase, the execution time remained within acceptable bounds for the given database sizes, with no significant deviations or anomalies observed.

Regarding memory usage, it is noteworthy that the measurements remained relatively stable across different database sizes. This consistency suggests that the memory requirements for the export process did not significantly fluctuate with changes in the database size. Such stability in memory usage is indicative of efficient resource management and suggests that the application is able to handle varying database sizes without experiencing excessive memory overhead.

Overall, while the execution time increased proportionally with the database size, the performance remained within satisfactory levels, indicating that the database export feature is capable of handling databases of different sizes effectively. These findings provide valuable insights into the system's performance characteristics and serve as a basis for further optimisation efforts to enhance overall efficiency and scalability.

Deployment:

For deployment to production, we utilise npm as our package manager and git for version control. The deployment process involves several steps to ensure the smooth transition of changes to the live environment.

Version Control with Git:

Each major change or feature addition undergoes version control using Git. This involves staging and committing changes before pushing them to the server repository.

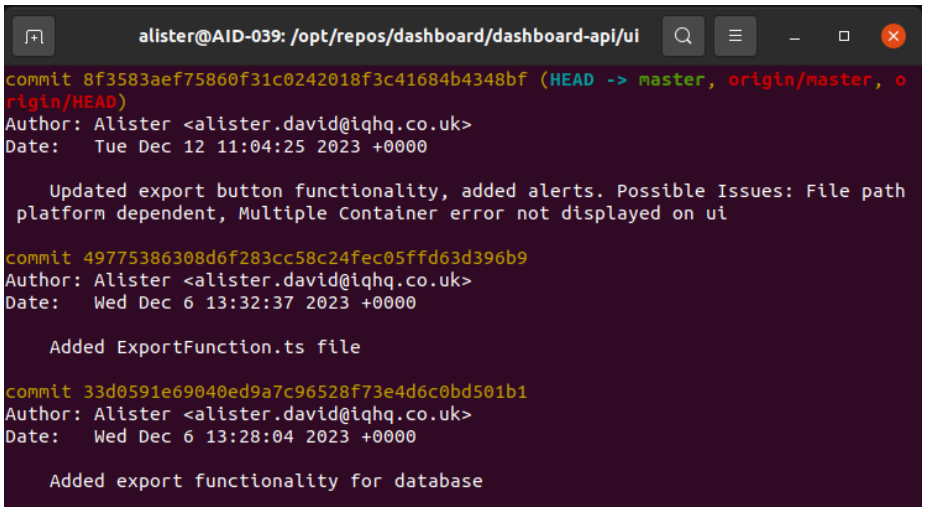


Figure 39 - Dashboard git log.

NPM Commands:

Navigate to Repository:

Initially, I navigate to the repository directory using the terminal on Ubuntu.

Terminal Setup:

Two additional terminals are opened, both already positioned in the repository directory.

Run Commands:

- In one terminal, I execute npm start. This command initiates the development server, automatically opening a web browser with the dashboard application.
- In the second terminal, I run npm run build. This command generates the production-ready build of the application.
- In the third terminal, npm run dev is executed. This command runs the application in development mode, providing access to logs and other debugging information.
-

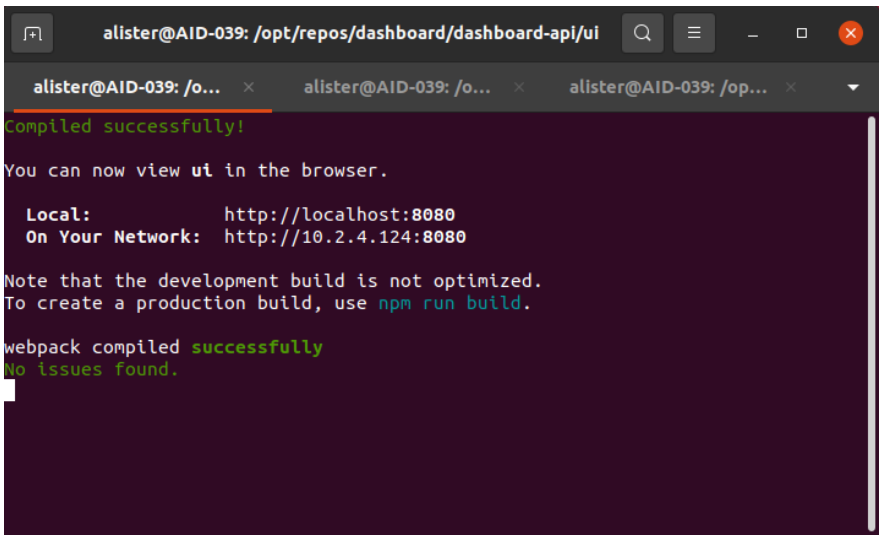


Figure 40 - Terminal running 'npm start' command.

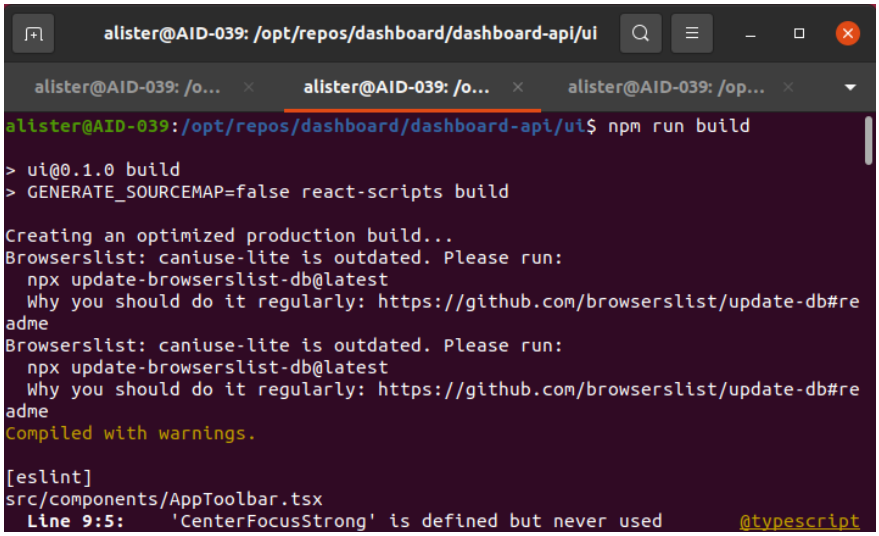


Figure 41 - Terminal running 'npm run build' command.

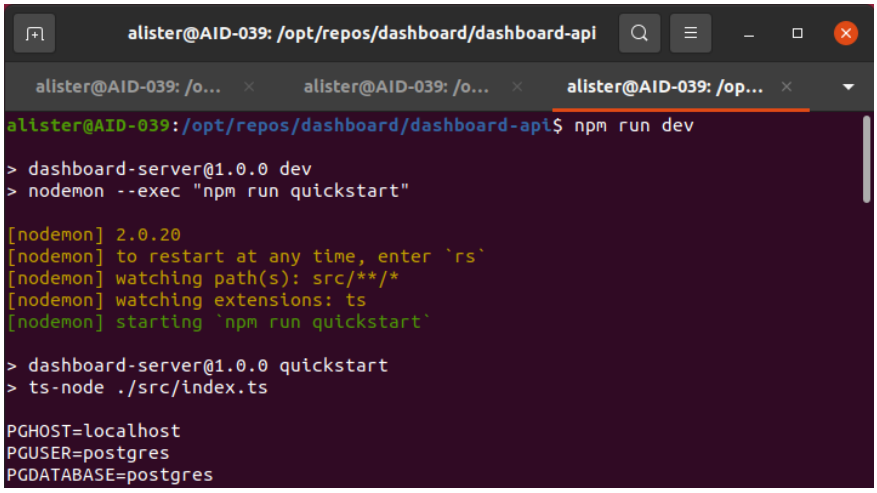


Figure 42- Terminal running 'npm run dev' command.

Monitoring and Debugging:

The terminal running npm run dev facilitates monitoring of logs and debugging during the development process.

User Interaction:

Once the application is running, interaction with the dashboard is possible through the web browser opened by npm start.

Please note that my involvement in the deployment process is limited, and I do not directly interact with the end-stage deployment to the customer environment.

Maintenance:

After the deployment of the dashboard application, ongoing maintenance and support are crucial to ensure its optimal performance and reliability. The maintenance phase involves several key activities aimed at addressing issues, implementing updates, and providing continuous support to users.

Bug Fixes and Issue Resolution:

- Regular monitoring of the application to identify and address any bugs or issues encountered by users.
- Timely resolution of reported issues to maintain the functionality and usability of the dashboard.

Software Updates and Enhancements:

- Implementation of software updates and patches to address security vulnerabilities and improve performance.
- Incorporation of new features and enhancements based on user feedback and evolving requirements.

Performance Monitoring and Optimisation:

- Continuous monitoring of application performance to identify and address any performance bottlenecks or optimisation opportunities.
- Optimisation of code, database queries, and server configurations to improve overall performance and responsiveness.

User Support and Training:

- Provision of user support to address queries, provide assistance, and troubleshoot issues encountered by users.
- Conducting training sessions or providing documentation to ensure users are familiar with the features and functionalities of the dashboard.

Backup and Disaster Recovery:

- Implementation and regular testing of backup procedures to ensure data integrity and resilience against data loss.
- Development and testing of disaster recovery plans to minimise downtime and ensure business continuity in the event of system failures or disasters.

Performance Reporting and Analysis:

- Generation of performance reports and analysis to track key metrics, identify trends, and make data-driven decisions to improve the application's performance and user experience.

Feedback Collection and Iterative Improvement:

- Regular collection of feedback from users to gather insights, identify areas for improvement, and prioritise future development efforts.

- Iterative improvement of the dashboard based on user feedback and evolving business needs to ensure its continued relevance and effectiveness.

Conclusion:

In conclusion, the development and implementation of the database export feature have been a significant milestone for our team. Overall, I am pleased with the outcome of the project, as it successfully addressed the need for a streamlined and efficient database export process.

Project Outcome

The database export feature has received positive feedback from both my manager and the end users. It has significantly improved the efficiency of exporting databases, saving valuable time and effort for our team members. The feature has also enhanced the overall user experience by providing a more intuitive and user-friendly interface.

Learning Experience

Throughout the project, I gained valuable insights into database management, software development, and project management practices. I learned how to effectively plan and execute a feature development cycle, including requirements gathering, design, implementation, testing, and deployment. Additionally, I improved my collaboration and communication skills through regular interaction with team members and stakeholders.

Areas for Improvement

While the database export feature meets the current requirements and expectations, there are areas where it could be further improved in the future. This includes enhancing error handling and logging mechanisms to provide more detailed feedback to users in case of issues. Additionally, implementing automated testing and continuous integration practices could help ensure the robustness and reliability of the feature across different environments.

Future Directions

Looking ahead, there are opportunities to expand the functionality of the database report feature by integrating additional data export formats, such as CSV or Excel. Furthermore, incorporating advanced analytics and reporting capabilities could provide users with deeper insights into their database performance and usage patterns.

In conclusion, the database report feature has been a valuable addition to our system, enhancing productivity and user satisfaction. I am grateful for the opportunity to work on this project and look forward to contributing to future enhancements and developments.