PROJECT FOUR

SSA DATA RETRIEVAL

Abstract

This document focuses on the design, development, testing and deployment of the SSA data retrieval feature within the existing dashboard application. Building on the previously implemented SSA data capture functionality, this project aims to enhance the usability and analytical capabilities of the dashboard by enabling users to retrieve and visualise historical SSA data. The key objective is to create an intuitive and user-friendly interface that allows users to select specific time periods and apply various filters to view the SSA data in a detailed waterfall plot.

Alister David

alisterdavid03@gmail.com

Contents

The Task:

The task for project 4 involves enhancing the functionality of the dashboard interface at IQHQ by implementing features to retrieve, filter, and visualise historical SSA data. This project aims to build upon the previously developed SSA data capture functionality by allowing users to access and analyse stored SSA data effectively.

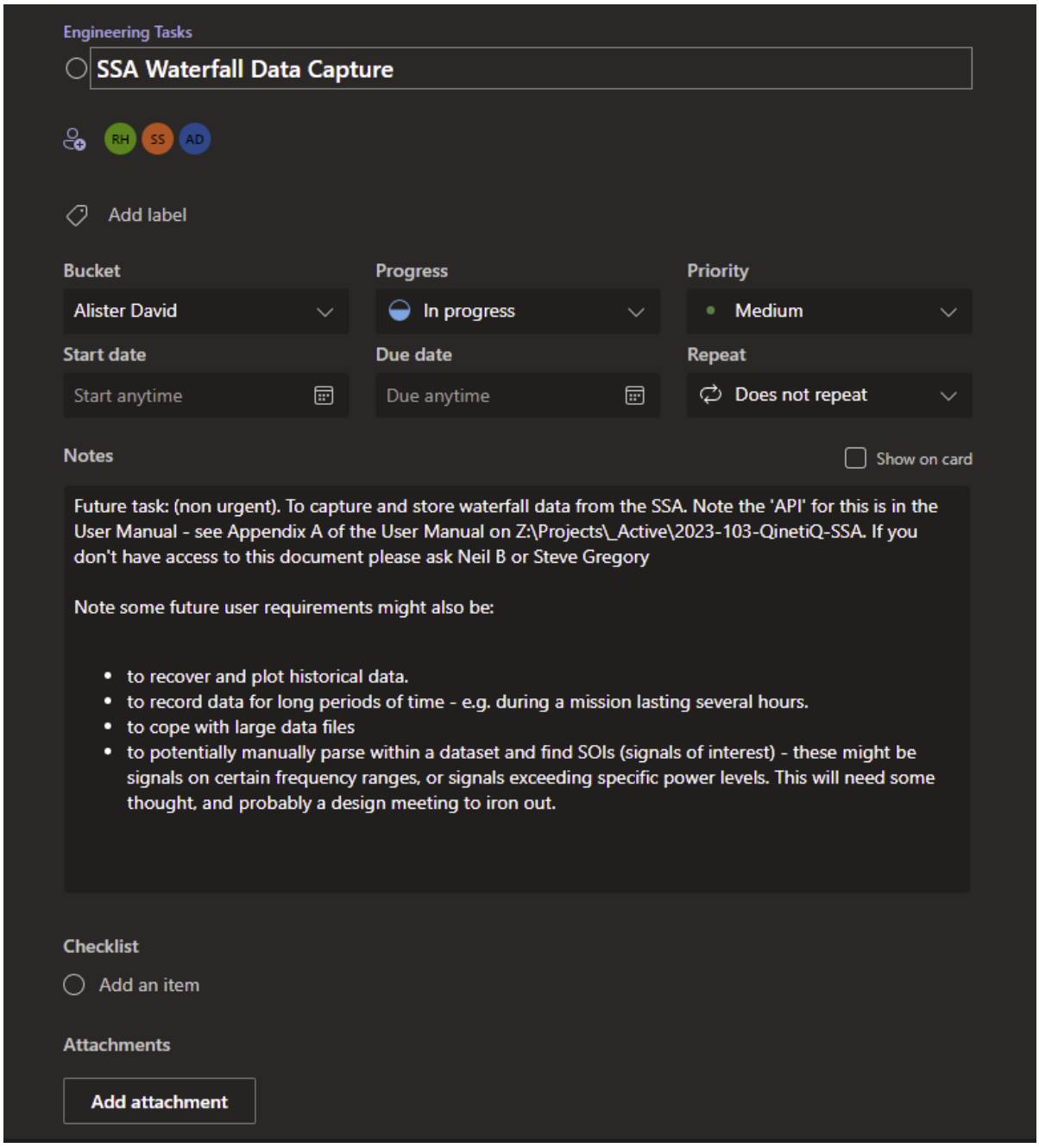


Figure 1 - SSA data retrieval task set on teams.

Feasibility Study:

Scope Definition:

Objective:

The primary objective is to create a new section on the dashboard where users can select specific time periods from the SSA database and view the corresponding data in a waterfall plot. Additionally, the task includes incorporating various filter options, such as frequency range, power levels, and down sampling rates, to refine the data selection and provide a more detailed and insightful analysis. This enhancement will significantly improve the dashboard's utility, offering users a powerful tool to review and interpret historical SSA data.

Specific Requirements:

- **UI Development for SSA Data Retrieval:**
  - Design and implement a user-friendly interface on the dashboard where users can interactively retrieve SSA data stored in the database.
- **Data Retrieval from Database:**
  - Develop backend functionality to retrieve SSA data stored in a dedicated database table based on user-defined time periods.
- **Data Filtering Options:**
  - **Down sampling:** Implement down sampling functionality to reduce the number of data points retrieved, allowing users to view summarised data over larger time periods.
  - **Threshold Filtering:** Enable filtering of SSA data based on power levels, allowing users to focus on data points above specified power thresholds to identify signals of interest.
- **Visualisation Features:**
  - **Waterfall Plot:** Display retrieved SSA data in a waterfall plot format, providing a comprehensive overview of frequency power levels over time.
  - **Graphical Representation:** Present SSA data graphically in other formats as needed, enhancing user understanding and analysis capabilities.
- **Compatibility and Integration:**
  - Ensure seamless integration of the SSA data retrieval and visualisation functionalities with existing features of the dashboard interface.
  - Maintain consistency in user experience and workflow across different dashboard modules.
- **Scalability and Performance:**

- Design the solution to handle potentially large datasets efficiently, ensuring scalable performance even with increased data volume over time.
- Implement efficient data handling and retrieval mechanisms to support long-term retrieval of SSA data.

Technical Feasibility:

Database Structure and Management:

- Assess compatibility of the existing PostgreSQL database with the requirements for retrieving SSA data.
- Design the schema for efficient retrieval of large volumes of power and frequency level data.

Relational & Non-relational Databases:

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

Relational Database (PostgreSQL):

- PostgreSQL, our chosen relational database management system, aligns with the principles of relational databases.
- It organises data into structured tables with predefined schemas, ensuring data integrity and consistency.

Non-relational Database Considerations:

- While non-relational databases offer advantages in handling unstructured data and scalability, we opted for PostgreSQL due to its structured data requirements and familiarity.
- Non-relational databases, such as MongoDB or Cassandra, offer advantages in handling unstructured or semi-structured data and scalability.
- However, for this project, the structured nature of database logs and the existing familiarity with PostgreSQL favoured its selection over non-relational alternatives.

I chose to use a relational database for this project due to its compatibility with the structured nature of the data captured from the SSA device. Relational databases, such as PostgreSQL, adhere to strict data integrity constraints, ensuring the consistency and accuracy of information, a critical aspect in military applications where precision is paramount. By organising data into structured tables with predefined schemas, PostgreSQL facilitates efficient management of complex data relationships, a feature particularly advantageous for handling diverse metrics from the SSA device.

Moreover, PostgreSQL's robust SQL querying capabilities empower us to extract actionable insights from the captured data. This querying power is essential for informed decision-making in military operations, allowing us to analyse trends, identify patterns, and extract valuable intelligence. Additionally, PostgreSQL offers comprehensive security features, encompassing authentication, authorisation, and encryption mechanisms. These security measures are indispensable for safeguarding sensitive military data from unauthorised access or tampering, ensuring the integrity and confidentiality of our data repository.

In summary, the combination of data integrity enforcement, structured schema organisation, powerful querying capabilities, and robust security features positions PostgreSQL as the ideal choice for managing the SSA data in our project. Its ability to seamlessly handle the structured data streams while ensuring data integrity and security aligns perfectly with the rigorous demands of military applications, thereby making it a reliable foundation for our data management endeavours.

Integration with Frontend:

- Design and implement a complete UI for the retrieval of SSA data, including filtering capabilities and time selection.

In ensuring the seamless integration of the SSA data retrieval feature with the frontend dashboard interface, I will prioritise compatibility with React components. Given that the entire dashboard frontend is constructed using React components, the goal is to seamlessly incorporate the retrieval functionality within this existing framework. React's component-based architecture aligns perfectly with the modular nature of frontend development, enabling the creation of reusable and composable UI elements. This modular approach not only enhances code reusability and maintainability but also facilitates scalability as the dashboard interface evolves with additional features.

React's virtual DOM and efficient rendering ensure smooth user interactions, crucial for maintaining a seamless dashboard experience. Leveraging React's rich ecosystem will streamline development and address challenges effectively. By using React components, I aim to efficiently implement the retrieval feature while ensuring compatibility, performance, and maintainability across the dashboard application.

Graph Display Using Chart.js and Waterfall Plot with HTML Canvas:

- Implement graphical representation of SSA data using Chart.js and HTML canvas elements.
- Integrate Chart.js for dynamic and interactive data visualisation and user-friendly graphs.

For displaying graphical representations of SSA data in the dashboard interface, I have utilised two primary technologies: Chart.js for versatile charting capabilities and HTML graphics canvas for creating waterfall plots.

Chart.js Integration:

Chart.js provides a powerful framework for rendering interactive charts within web applications. Leveraging its intuitive API and extensive chart types, I implemented dynamic graphs to visually represent SSA data trends. Chart.js supports various chart types such as line charts, bar charts, and pie charts, allowing flexibility in presenting various aspects of SSA data. Its responsiveness and built-in animations enhance user interaction by visualising real-time data updates seamlessly.

Waterfall Plot with HTML Canvas:

In addition to Chart.js, I employed HTML canvas elements to generate waterfall plots, a specialised graph type that displays data as a series of rising and falling columns. This approach is particularly useful for visualising spectrum usage over time, where the height of each column represents signal intensity or power levels. By leveraging the low-level drawing capabilities of HTML canvas, I achieved precise control over graphical elements, ensuring accurate representation of SSA data in a format that is both informative and visually appealing.

Backend-Frontend Communication:

- Implement mechanisms for retrieval of SSA data from backend to frontend based on the filter options set in the frontend.

Regarding the backend-frontend communication, the need for seamless interaction between the two components is vital for displaying the SSA data graphically and status updates in the frontend dashboard interface. To achieve this, I will implement HTTP-based routes using the Express framework in Node.js. This approach offers simplicity, versatility, and widespread adoption in web development, making it well-suited for exchanging data between the frontend and backend components of our application. By defining specific routes for sending filter options and getting the SSA data, I will establish clear endpoints for frontend interaction, enabling seamless integration of data retrieval into the dashboard interface. Through this backend-frontend communication mechanism, users will have access to historic SSA data and relevant insights, enhancing the overall usability and effectiveness of the dashboard application.

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 data retrieval functionality in our application.

Backend Technology Stack:

- Assess the suitability of the backend technologies (e.g., Node.js, TypeScript) for implementing the retrieval functionality.
- Determine if the chosen technologies support database connectivity and database query for the retrieval of SSA data.

The backend technologies of Node.js and TypeScript are well-suited for implementing the retrieval functionality in the project. With dedicated support for database connectivity, evidenced by dependencies like pg for PostgreSQL interaction, these technologies enable efficient retrieval of data. 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 retrieval feature.

Error Handling and Logging:

- Develop error handling mechanisms to manage exceptions and ensure continuous data retrieval.
- Implement logging functionalities to track data retrieval activities and support troubleshooting and auditing.

To ensure the reliability of SSA data retrieval, robust error handling mechanisms will be developed to effectively manage exceptions and errors that may arise during the data retrieval process. These mechanisms will include thorough error detection and recovery procedures, designed to handle scenarios such as database query failures, network interruptions, or data parsing errors. By implementing proactive error management strategies, the system will minimise downtime and maintain uninterrupted access to critical SSA data, essential for real-time analysis and operational decision-making.

Scalability and Future Enhancements:

- Ensure the system is scalable to handle future growth in data volume and user demand.

To ensure robust scalability in SSA data retrieval, the architecture will be meticulously designed to accommodate anticipated increases in data volume and user demand. This entails employing scalable technologies and adopting best practices for database management. Strategies will include optimising database performance, implementing efficient indexing strategies, and considering horizontal scaling options to handle larger datasets seamlessly. By proactively addressing scalability concerns, the system will maintain optimal performance levels, ensuring uninterrupted access to critical SSA data as operational demands evolve.

Requirement Analysis:

Background:

The SSA Data Retrieval project aims to enhance the functionality of the dashboard interface by enabling users to select specific time periods from the database containing SSA data. This data will be visualised in the form of a waterfall plot, with additional filter options to refine the displayed data. The project builds upon previous work on SSA data capture, leveraging technologies like Chart.js for graphical representation and ensuring seamless integration with the existing React-based dashboard interface.

Functional Requirements:

1. **UI Development for SSA Data Retrieval:** Develop a user interface that allows users to select specific time periods from the database containing SSA data and apply filters such as down sampling and power threshold filtering.
2. **Backend Development:** Implement backend logic to efficiently retrieve SSA data from the PostgreSQL database, ensuring optimal performance and data integrity.
3. **Graphical Representation:** Integrate Chart.js library to visually represent SSA data in real-time using several types of charts such as line charts and bar charts, providing users with clear graphical insights.
4. **Waterfall Plot Implementation:** Utilise the HTML graphics canvas element to create interactive waterfall plots, enabling users to visualise spectrum data over time with detailed frequency and power levels.
5. **Compatibility and Integration:** Ensure seamless integration of the SSA data retrieval feature with the existing React-based dashboard interface, maintaining consistency in user experience and interface design.
6. **Error Handling and Logging:** Develop robust error handling mechanisms to manage exceptions, ensure continuous data capture, and implement logging

functionalities to track data capture activities and support troubleshooting and auditing.

Non-functional Requirements:

1. **Performance:** Ensure that the SSA data retrieval process and graphical rendering are responsive and performant, even under high data load conditions, to provide users with real-time updates and smooth interaction.
2. **Scalability:** Design the system architecture to scale efficiently with increasing data volume and user demands, supporting future growth without compromising performance or usability.
3. **Reliability:** Develop comprehensive error handling mechanisms to manage exceptions and ensure continuous operation of the SSA data retrieval feature, backed by logging functionalities for troubleshooting and auditing purposes.
4. **Usability:** Design an intuitive and user-friendly interface for the SSA data retrieval feature, incorporating clear navigation, informative tooltips, and contextual help to guide users effectively.
5. **Maintainability:** Document the system architecture, codebase, and deployment procedures thoroughly to facilitate ease of maintenance and future enhancements, adhering to coding standards and best practices.

User Story:

Title: Retrieve and Display Historic SSA Data

As a dashboard user, I want to retrieve SSA data based on a specified time period, view it on both a waterfall plot and a graph, and have the ability to apply filters for detailed analysis.

Acceptance Criteria:

1. There should be a dedicated section in the dashboard interface specifically for viewing historic SSA data.
2. Provide a dropdown menu listing available devices to select from for retrieving SSA data.
3. Offer a dropdown menu to choose the specific stream from which SSA data will be retrieved.
4. Include filter options such as down sampling by a factor and threshold filtering by power level to refine the displayed SSA data.
5. Upon setting the desired options and clicking "Update," the relevant SSA data should be displayed as a waterfall plot in the designated section.

6. When switching to the capture section, the SSA data should be visualised in a graph format for easy comparison and analysis.
7. If no SSA data matches the selected options, the system should notify the user accordingly to indicate that no data is available.
8. Errors encountered during data retrieval should be handled gracefully, with clear and meaningful error messages displayed to guide the user and facilitate troubleshooting.

This user story outlines the need for retrieving and displaying SSA data and specifies the expected behaviour and outcome when using this feature

Design:

System Architecture Design:

Below is the system architecture design, which provides an overview of the underlying structure and components of the application. This diagram illustrates the arrangement of frontend and backend elements, showcasing how they interact to facilitate the seamless capture, storage, and display of live data from the SSA device.

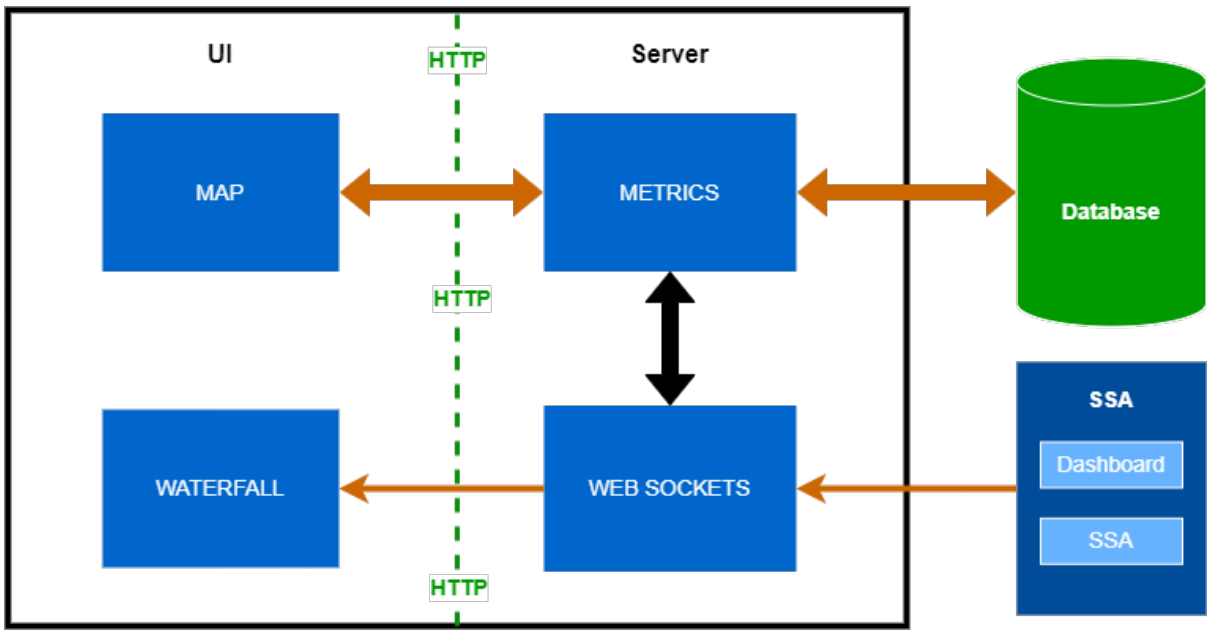


Figure 2 - System architecture design of the dashboard application.

Activity Diagram :

The activity diagram provided illustrates the process flow involved in retrieving and displaying data from the SSA database within the application. With three swim lanes representing the user, Dashboard Application, and Backend components, it highlights the sequential steps initiated when the user interacts with the update

button. This diagram offers stakeholders a clear depiction of the interactions and handoffs between different system elements, enhancing understanding of the data retrieval process and its integration within the application architecture.

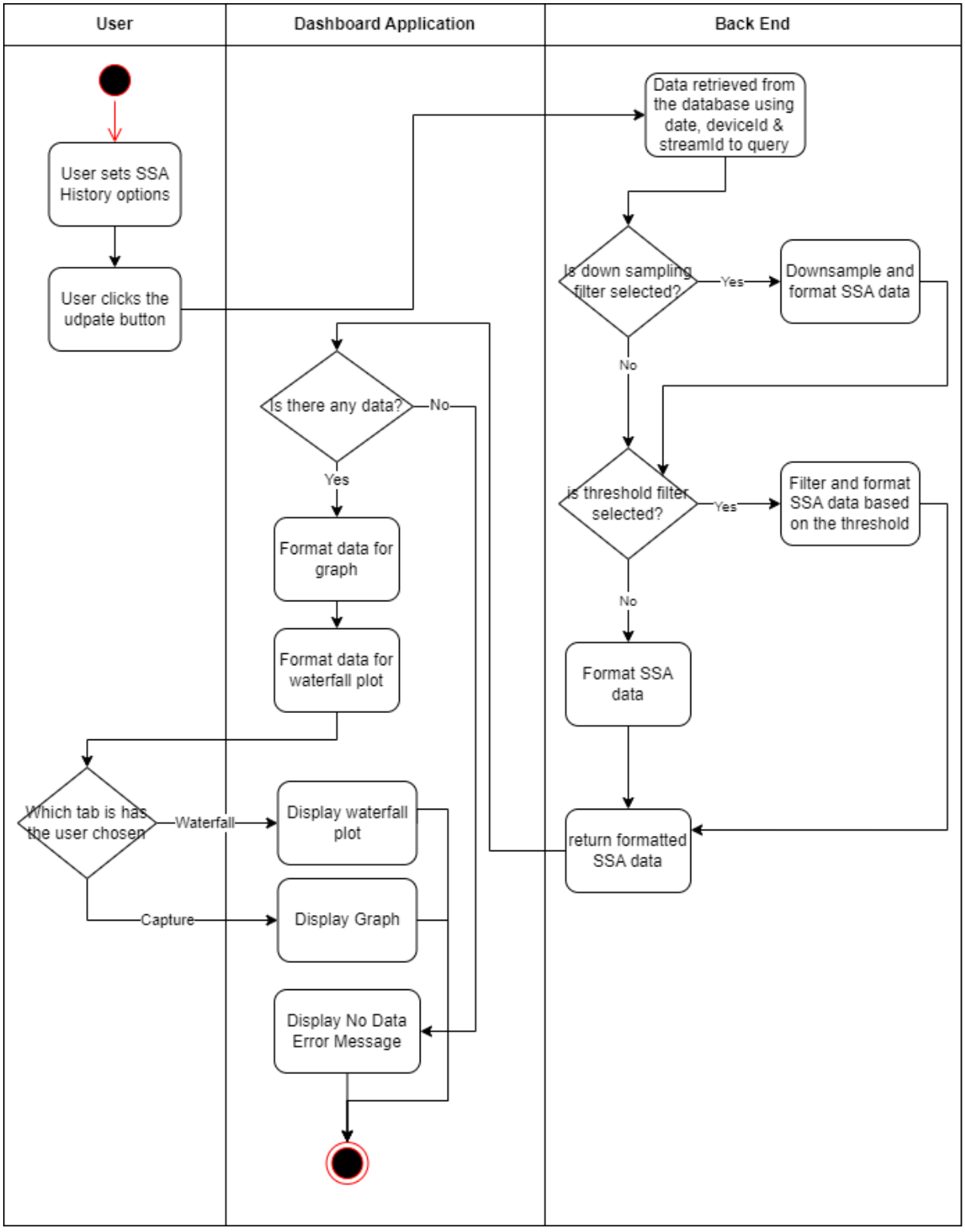


Figure 3 - Activity diagram of the data retrieval process.

ER Diagram:

The ER (Entity-Relationship) diagram presented here offers a concise visual representation of the database schema specifically tailored for this project. It showcases the structure of the database table responsible for storing SSA metrics. This focused diagram provides stakeholders with a clear understanding of the data model underpinning the SSA data retrieval mechanism, facilitating better comprehension of the system architecture and database design choices.

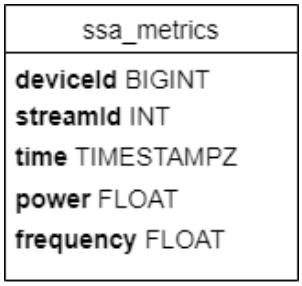


Figure 4 - Entity relation table for the ssa_metrics table.

Pseudo Code:

Included within the design documentation is pseudocode, serving as a clear roadmap for implementing various system functionalities. Pseudocode outlines the logical flow of features independently of specific programming languages or frameworks. By detailing key actions and decision points, it ensures stakeholders share a collective understanding of system behaviour. This aids collaboration among developers, facilitating discussion and design refinement before actual implementation. Pseudocode offers a concise outline of the steps involved, promoting clarity and efficiency in the development process.

Pseudo Code for retrieving the SSA data:

```
// Function to get SSA history data
function GetSSAHistory(deviceId, streamId, startDate, endDate,
powerFilter, downSampleFilter, power, factor):
    Try:
        Connect to database
        Query data for deviceId and date range
        Release database connection

        Apply filters to data
        Structure filtered data into HistoryStreamsData format

        Return HistoryStreamsData
    Catch error:
        Print error message
```

Figure 5 - Pseudo code to retrieve SSA data from the database.

Pseudo Code for applying the filters:

```
 // Function to downsample data by a given factor
function downSampling(data, factor):
    Create an empty list called filteredData
    For every factor-th element in data:
        Add element to filteredData
    Return filteredData

// Function to filter data by a minimum threshold value
function thresholdFiltering(data, threshold):
    Create an empty list called filteredData
    For each element in data:
        If element's power is greater than threshold:
            Add element to filteredData
    Return filteredData

// Function to apply filters and return the filtered data
function applyFilters(data, applyPowerFilter, applyDownSampleFilter, threshold, factor):
    Set filteredData to data
    If applyPowerFilter:
        filteredData = thresholdFiltering(filteredData, threshold)
    If applyDownSampleFilter:
        filteredData = downSampling(filteredData, factor)
    Return filteredData
```

Figure 6 - Pseudo code for the filters and applying them.

UI Component Design :

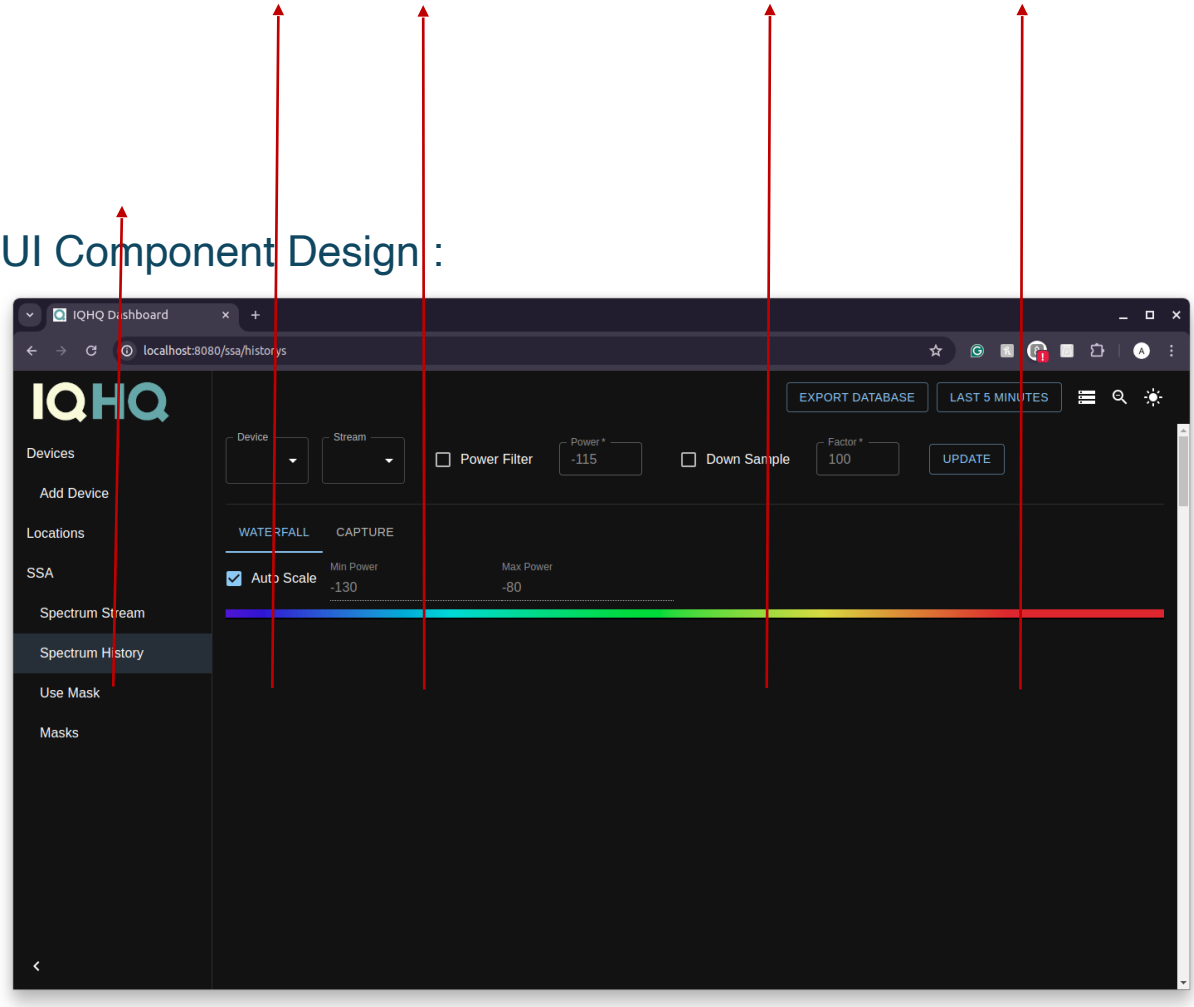


Figure 7 - Proposed SSA data retrieval UI.

New spectrum history section	Devices drop down	Streams drop down	Filter options	Update button

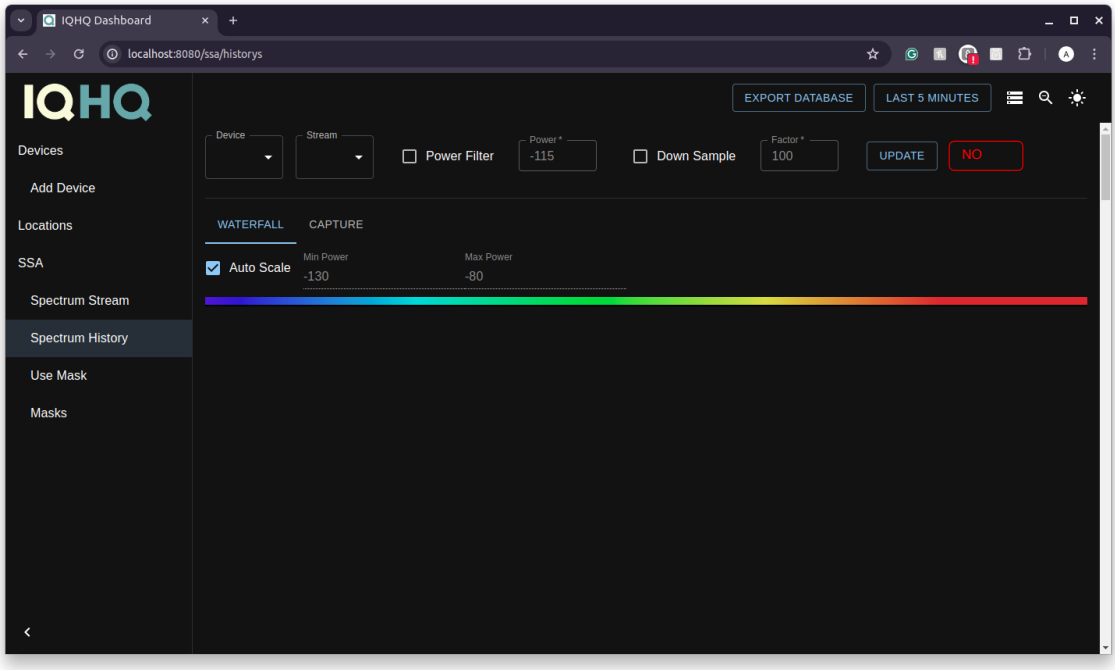


Figure 8 - Proposed data retrieval UI when no data is found.

No data found error

Development:

Creating a SSA history Section:

I decided to first develop a separate section for the data retrieval feature. Since it was directly related to the SSA component of the dashboard, I determined it would be best to place the SSA history section within the SSA content, below the Spectrum Stream feature. The Spectrum Stream section is used to stream live SSA data and includes options to set the stream parameters, view the waterfall plot, and display the frequency vs. power graph.

Recognising that the SSA history section would have a similar layout to the Spectrum Stream, I duplicated the ‘SSAStreamView.tsx’ file and renamed the duplicate to ‘SSAStreamHistory.tsx’. This approach leverages code reusability, ensuring efficiency and consistency across the application. By reusing existing code, I minimised development time and reduced the potential for introducing new bugs, as the duplicated code had already been tested and validated.

Next, I needed to connect the SSA history section to the dashboard. I located the ‘App.tsx’ file, which is responsible for the overall layout and routing of the dashboard. Within this file, I found the section defining the SSA routes in the sidebar. I added a route for SSAStreamHistory to this section, ensuring that users can navigate to the SSA history view from the dashboard sidebar.

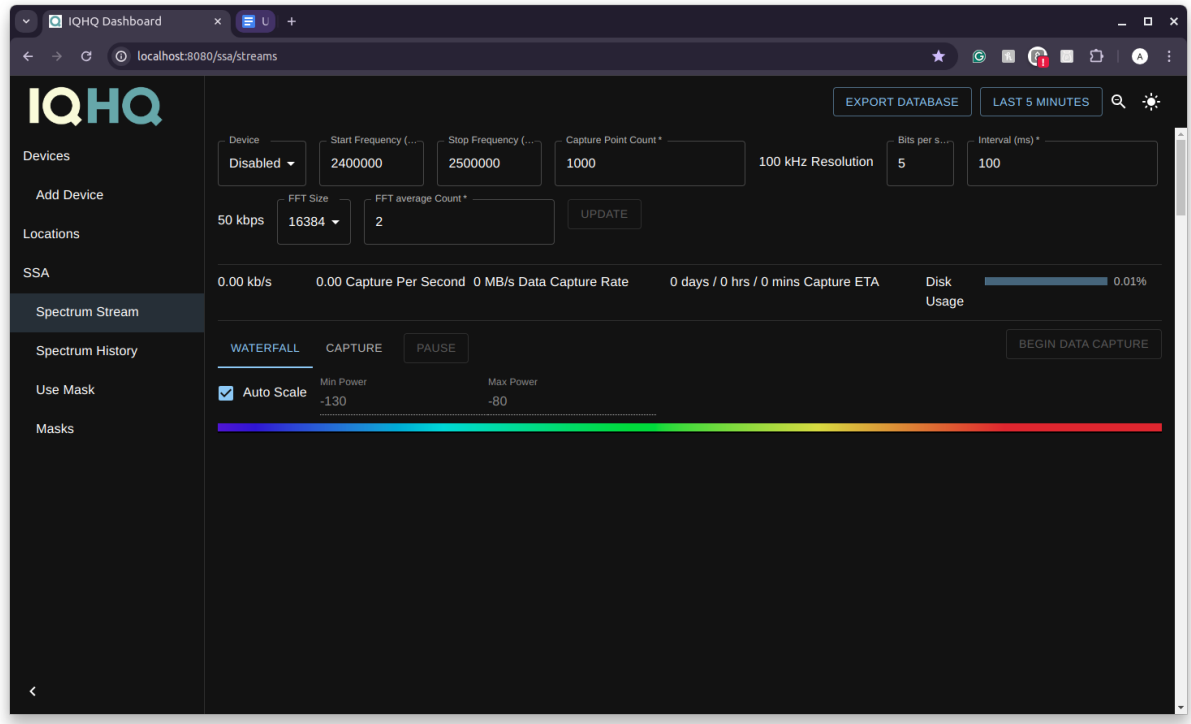


Figure 9 - Spectrum stream section on the dashboard.

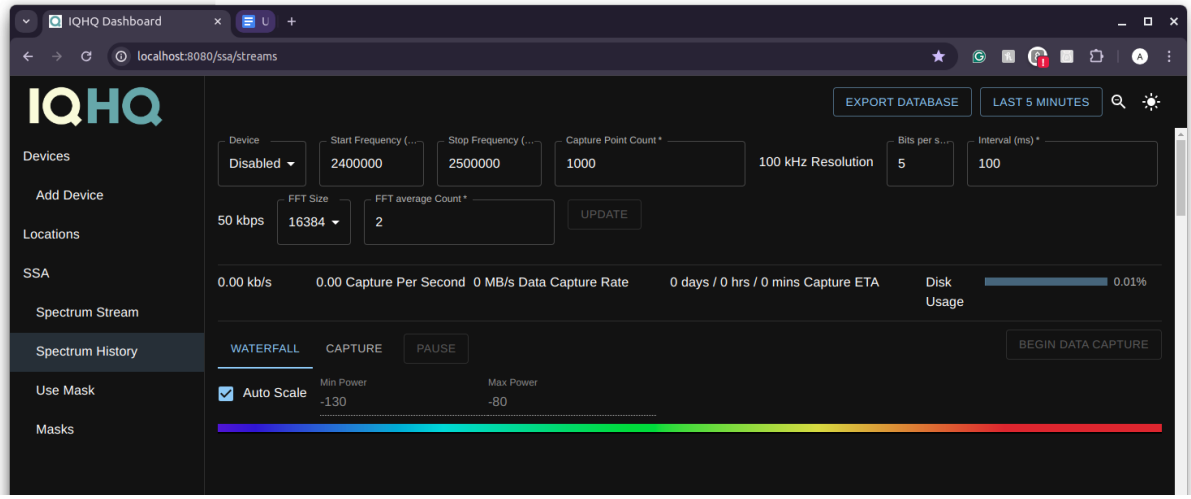


Figure 10 - Spectrum History section on the dashboard.

App.tsx | SSAStreamHistory.tsx | SSAStreamView.tsx

ui > src > components > App.tsx > App

```
42  function App(props: {
93      const ssaRoutes : typeof routes = [];
94      if(displayInfo?.rootViews.includes("SSAStreaming")) {
95          ssaRoutes.push({
96              path: "/ssa/streams",
97              el: <SSAStreamView />,
98              displayName: "Spectrum Stream",
99              indent: true,
100         }, {
101             path: "/ssa/history",
102             el: <SSAStreamHistory />,
103             displayName: "Spectrum History",
104             indent: true,
105         })
106     }
107     if(displayInfo?.rootViews.includes("SSAMasks")) {
108         ssaRoutes.push({
109             path: "/ssa/useMask",
110             el: <SendMaskView />,
111             displayName: "Use Mask",
112             indent: true,
113         }, {
114             path: "/ssa/masks",
115             el: <SSAMaskView />,
116             displayName: "Masks",
117             indent: true,
118         })
119     }
120     if(ssaRoutes.length) {
121         routes.push({
122             path: "/ssa",
123             displayOnly: true,
124             displayName: "SSA",
125         }, ...ssaRoutes);
126     }
127
128     const theme = useTheme();
129
130     const [isSidebarOpen, setIsSidebarOpen] = React.useState(true);
```

Figure 11 - App.tsx file.

Creating an SSA history type:

To ensure clarity and consistency in the parameters needed for the SSA history section, I decided to create a dedicated type for the SSA history options. By defining a clear type, I established a guide for the parameters required to develop the UI for the spectrum history section and provided a clear specification for the expected data to be sent and received.

In the shared directory, where other types are stored for access by both the frontend and backend, I created a file named ‘SSAHistoryOpts.ts’. In this file, I defined the SSAHistoryOpts type with the following parameters:

- **deviceId**: A number that filters SSA data from the database by the selected device.
- **startDate**: A string representing the start date to retrieve SSA data from a specific time period.

- **endDate**: A string representing the end date for the data retrieval period.
- **filters**: An object used to identify the chosen filter.
- **power**: A number used if the threshold filter is chosen, to filter out SSA data below the specified power level.
- **factor**: A number used if the down sampling filter is chosen, to reduce the SSA data size by the specified factor.

By setting these parameters, the SSAHistoryOpts type provides a structured and clear approach to managing the options required for SSA history retrieval, ensuring that all necessary information is included and correctly formatted.

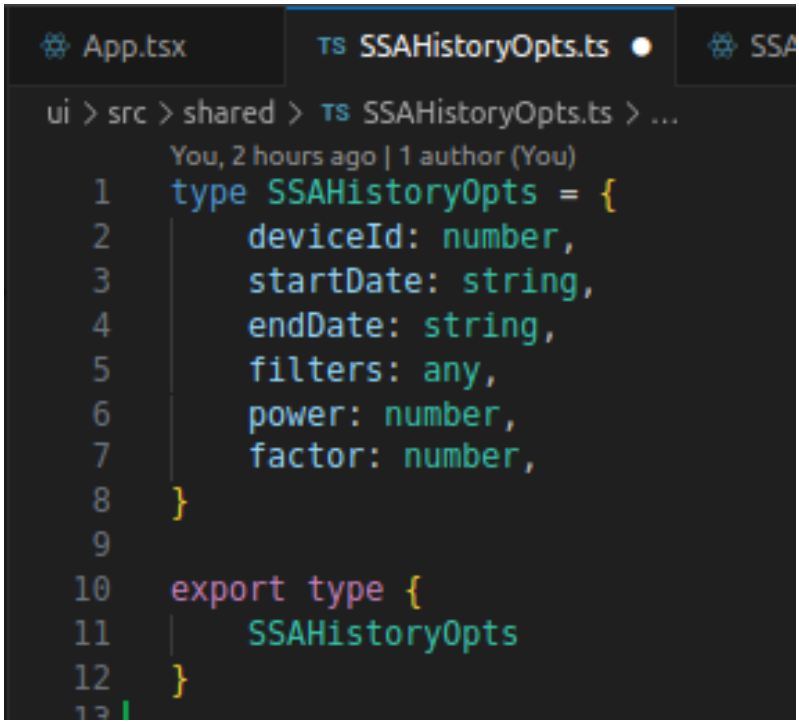


Figure 12 - SSAHistoryOpts type.

Updating SSAStreamHistory.tsx:

I decided to update the UI of the spectrum history section to make it specific to data retrieval and to utilise the SSAHistoryOpts type. In the StreamOptsEditor function, I removed all the old React text fields and drop-downs except for the device ID drop-down. I then added a React select component to choose filter options, incorporating checkboxes within the drop-down to allow the user to select multiple filters. Additionally, I introduced two text fields, one for the power value and one for the down sampling factor. The device ID drop-down and update button from the spectrum stream were retained for reuse in this section.

SSAStreamView.tsx | SSAStreamHistory.tsx

ui > src > components > SSA > SSAStreamHistory.tsx > ...

```tsx
function StreamOptsEditor(props: {

    const options = [
    'None',
    'DownSampling',
    'Power Filter'
    ];

    const [filters, setfilters] = React.useState<string[]>([]);

    const PowerFilterChange = (event: SelectChangeEvent<typeof filters>) => {
        const {
        target: { value },
        } = event;
        setfilters(
        // On autofill we get a stringified value.
        typeof value === 'string' ? value.split(',') : value,
        );
        value.includes("Power Filter") ? setPowerFilterChosen(true) : setPowerFilterChosen(false)
        value.includes("DownSampling") ? setDownSampleChosen(true) : setDownSampleChosen(false)
    };

    const disableOptions = filters.includes("None");

    return <form
        onSubmit={handleSubmit(values=>{
            setError(undefined);

            console.log('Power Value:', powerValue);
            console.log('Factor Value:', factorValue);
            console.log(filters)
            console.log("Start Time:",formatTimestampToString(startTime))
            console.log("End Time:",formatTimestampToString(endTime))

            let deviceId = isFinite(values.deviceId as number)
                ? values.deviceId as number
                : undefined
            dotrigger({
                ...values,
            })
                .then(()=>{
                    if(isMounted()) {
                        props.onComplet()
                        props.setStreamEnabled(deviceId !== undefined)
                        console.log("Device ID:",deviceId)
                        console.log('----------------------')
                    }
                })
        })}
```

Figure 13 - StreamOptsEditor function.

SSAStreamView.tsx | SSAStreamHistory.tsx

ui > src > components > SSA > SSAStreamHistory.tsx > ...

```
546  function StreamOptsEditor(props: {
655      >
656          <Grid container spacing={2}>
657              <Grid item>
658                  <DeviceSelect
659                      sx={{ width: 125 }}
660                      devices={devices || []}
661                      deviceTypes={[SSADeviceType]}
662                      label="Device"
663                      required
664                      defaultValue={props.defaultValue.deviceId ?? "None"}
665                      {...register("deviceId", {valueAsNumber: true})}
666                      onChange={(event) => event.target.value == 'None' ? setValidDevice(true) : setValidDevice(false)}
667                  />
668              </Grid>
669              <Grid item>
670                  <FormControl sx={{ width: 200 }}>
671                      <InputLabel id="filter-checkbox">Filters</InputLabel>
672                      <Select
673                      labelId="filter-checkbox"
674                      id="filter-checkbox"
675                      multiple
676                      value={filters}
677                      onChange={PowerFilterChange}
678                      input={<OutlinedInput label="Tag" />}
679                      renderValue={(selected) => selected.join(', ')}
680                      MenuProps={MenuProps}
681                      >
682                      {options.map((name) => (
683                          <MenuItem key={name} value={name} disabled={disableOptions && name !== 'None'}>
684                              <Checkbox checked={filters.indexOf(name) > -1} />
685                              <ListItemText primary={name} />
686                          </MenuItem>
687                      ))}
688                      </Select>
689                  </FormControl>
690
691              </Grid>
692              <Grid item>
693                  <TextField
694                  disabled={!powerFilterChosen}
695                  style={{width:100}}
696                  defaultValue="-118"
697                  id="power-value"
698                  label="Power"
699                  type="number"
700                  InputLabelProps={{
701                      shrink: true,
702                  }}
703                  onChange={(event) => setPowerValue(Number(event.target.value))}
704                  />
705              </Grid>
```

Figure 14 - React components to create the SSA history options.

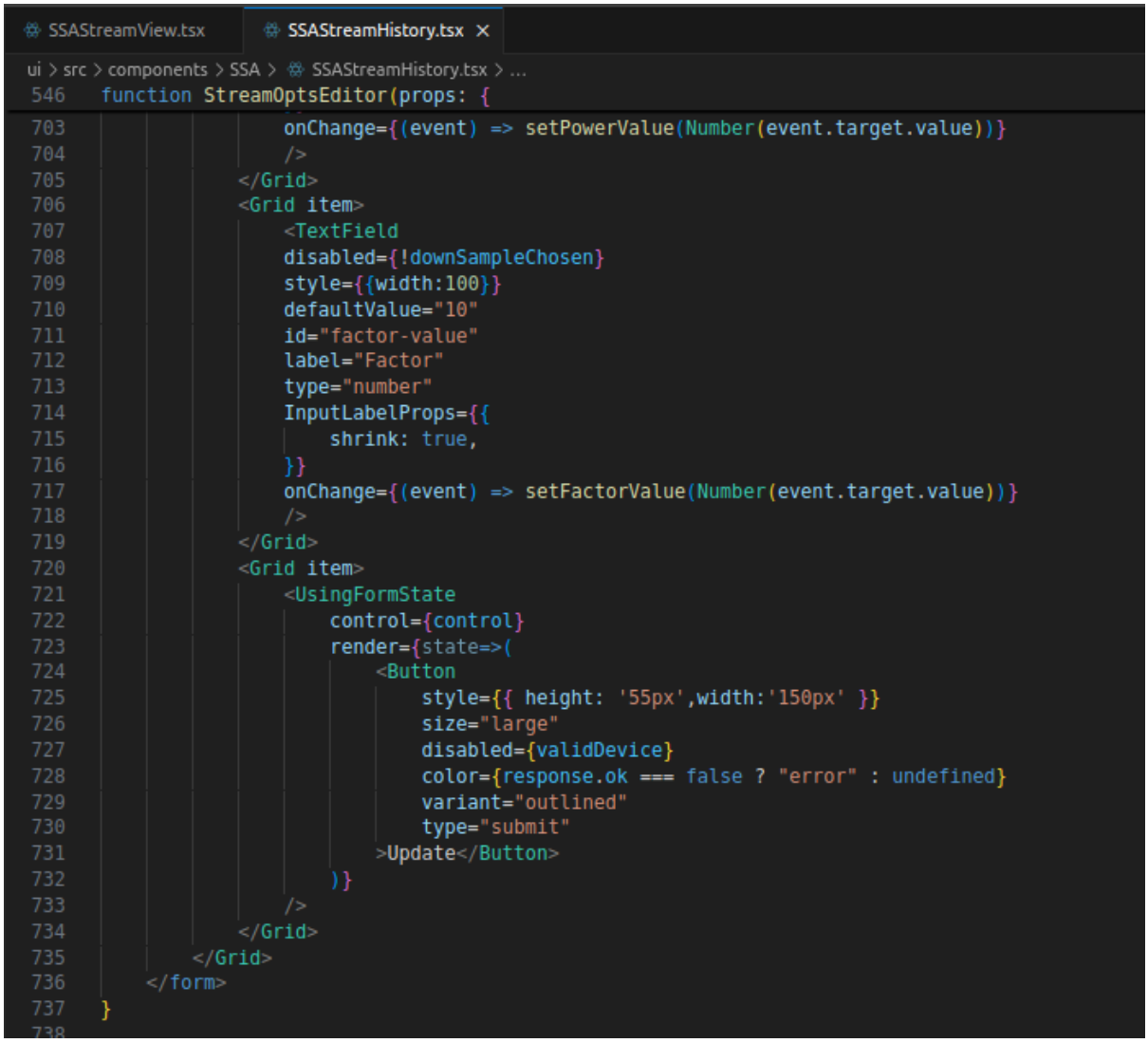


Figure 15 - React components to create the SSA history options.

I replaced the default value property in StreamOptsEditor from StreamOpts to SSAHistoryOpts and updated the useForm hook to use the new SSAHistoryOpts type instead of the StreamOpts type. I also added useState hooks for each option to enable value updates. By implementing a function called PowerFilterChange, I managed the checkboxes and the corresponding text fields, enabling or disabling them based on the selected filters.

Initially, I planned to add a React date picker component to select the start and end dates. However, I realised I could use the existing date picker in the app toolbar section of the dashboard, which has broader functionality. This date and time picker offers two modes: static and relative. Users can either backtrack on SSA data relative to the current time or choose specific start and end dates and times. To access the date and time I can use the useComplexTimeRange function which returns an object containing relative and static time in milliseconds, for now I am focusing on relative time. By reusing this component, I not only accelerated development but also maintained consistency across the application. This approach minimised redundant code, reduced the risk of bugs, and ensured a unified user experience, as updates to the date picker will automatically propagate throughout the dashboard.

```
const [timeRange] = useComplexTimeRange()

const startTime = timeRange[0].type === 'relative' ? timeRange[0].offset : timeRange[0].time
const endTime = timeRange[1].type === 'relative' ? timeRange[1].offset : timeRange[1].time
```

Figure 16 - Getting the start date/time and end date/time.

I created a function called formatTimestampToString which converts the Unix timestamp into a database compatible timestamp.

```
//function that converts unix timestamp into database compatible timestamp
function formatTimestampToString(timestamp: number) {
    const date = new Date(timestamp);
    const year = date.getFullYear();
    const month = (date.getMonth() + 1).toString().padStart(2, '0');
    const day = date.getDate().toString().padStart(2, '0');
    const dateString = `${year}-${month}-${day}`;
    const timeString = date.toLocaleTimeString('en-UK', { hour12: false });
    return `${dateString} ${timeString}`;
}
```

Figure 17 - Function that converts Unix timestamp into a database compatible timestamp.

I then updated the StreamOptsView function, which loads and sets the history options, to use the SSAHistoryOpts type. I created a function called getDefaultData as a temporary fix for the StreamOptsView function, since it returns the StreamOptsEditor which requires a default value. As the route had not yet been made, no data was being passed through, so when there is no data, it uses the values from the getDefaultData function.

```
//Loading and setting stream opts
function StreamOptsView(props: {
    setStreamEnabled: (en: boolean)=>void
}) {
    const {data, refetch} = useHTTPGet<SSAHistoryOpts>("/ssa/history", {interval_ms: 5000});

    // Temporarily set streamEnabled to true for testing
    useEffect(() => {
        props.setStreamEnabled(true);
    }, []);
    // Can delete this portion

    console.log("SSAHistoryOpts:",data)
    const editorKeyNum = useCountVariableChanges(
        data,
        (a,b)=>!deepCompare(a,b)
    );

    return <>
        <StreamOptsEditor
            key={editorKeyNum + "-Editor"}
            defaultValue={data || getDefaultData()} // delete || getdefaultdata
            onComplet={refetch}
            setStreamEnabled={props.setStreamEnabled}
        />
    </>
}
```

Figure 18 - StreamOptsView function.

```
//temporary - please delte
function getDefaultData(): SSAHistoryOpts {
    return {
        deviceId: 0,
        startDate: "",
        endDate: "",
        filters: [],
        power: 0,
        factor: 0
    };
}
```

Figure 19 - Default values function.

To test how the waterfall plot data worked, I created some mock data by analysing the data from the spectrum stream and matching its format. I then passed this mock data through the makeData function, and the result of the makeData function is used for the waterfall plot and the graph. The waterfall plot seemed to work, outputting a single line of data each time the update button was clicked, but the graph did not work. I need to conduct more research on the WaterfallDisplay and SpectrumCaptureView functions to understand how each uses data to output graphical data.

```
779  function SSAStreamHistory() {
780      const mockData = {
781          1: {
782              "1" : 10, "2" : 20, "3" : 30, "4" : 10, "5" : 20,
783              "6" : 30, "7" : 10, "8" : 20, "9" : 30, "10": 10,
784              "11": 20, "12": 30, "13": 10, "14": 20, "15": 30,
785              "16": 10, "17": 20, "18": 30, "19": 10, "20": 20,
786              "21": 30, "22": 10, "23": 20, "24": 30, "25": 10,
787              "26": 20, "27": 30, "28": 10, "29": 20, "30": 30,
788              "31": 10, "32": 20, "33": 30, "34": 10, "35": 20,
789              "36": 30, "37": 10, "38": 20, "39": 30, "40": 10,
790              "41": 20, "42": 30, "43": 10, "44": 20, "45": 30,
791              "46": 10, "47": 20, "48": 30, "49": 10, "50": 20, "51": 24
792          }
793      };
794
795      const chartData = makeData(mockData)
796
797      const [activeTab, setActiveTab] = useState(0);
798      const [, _forceRender] = useState(0);
799      const forceRender = ()=>_forceRender(prev=>prev+1);
800      const [run, setRun] = useState(true);
801
802      const [streamEnabled, setStreamEnabled] = useState<boolean>(false);
803      const {dataRef, statsRef} = useViewLoading(()=>{
804          if(run)
805              forceRender();
806      });
807
808      const data = mockData;
809      const chartDataRef = useRef<ChartData<"line">>({datasets: []});
810      useEffect(()=>{
811          const newData = makeData(data);
812          if(newData.datasets.length)
813              chartDataRef.current = newData;
814      }, [data])
815
```

Figure 20 - SSAStreamHistory function.

```
830     return <div>
831         <StreamOptsView setStreamEnabled={setStreamEnabled}/> <br />
832         <Divider />
833         <Grid container style={{marginTop: 10}}>
834             <Grid item>
835                 <Tabs value={activeTab} onChange={(_, val)=>setActiveTab(val)} style={{marginBottom:"10px"}}>
836                     <Tab label="Waterfall"/>
837                     <Tab label="Capture"/>
838                 </Tabs>
839             </Grid>
840 
841             <Grid item style={{marginTop: 5, marginLeft: 10}}>
842                 {/*Disabled if device not selected && no waterfall*/}
843                 <Button
844                     variant="outlined"
845                     onClick={()=>setRun(prev=>!prev)}
846                     >{run ? "Pause" : "Continue"}
847                 </Button>
848             </Grid>
849         </Grid>
850         {activeTab === 0 && <WaterfallPlot
851             data={chartData}
852         />}
853         {activeTab === 1 && <SpectrumCaptureView
854             data={chartData}
855         />}
856     </div>
857 }
858 
859 export default SSAStreamHistory;
```

Figure 21 - Return div of SSAHistory function.

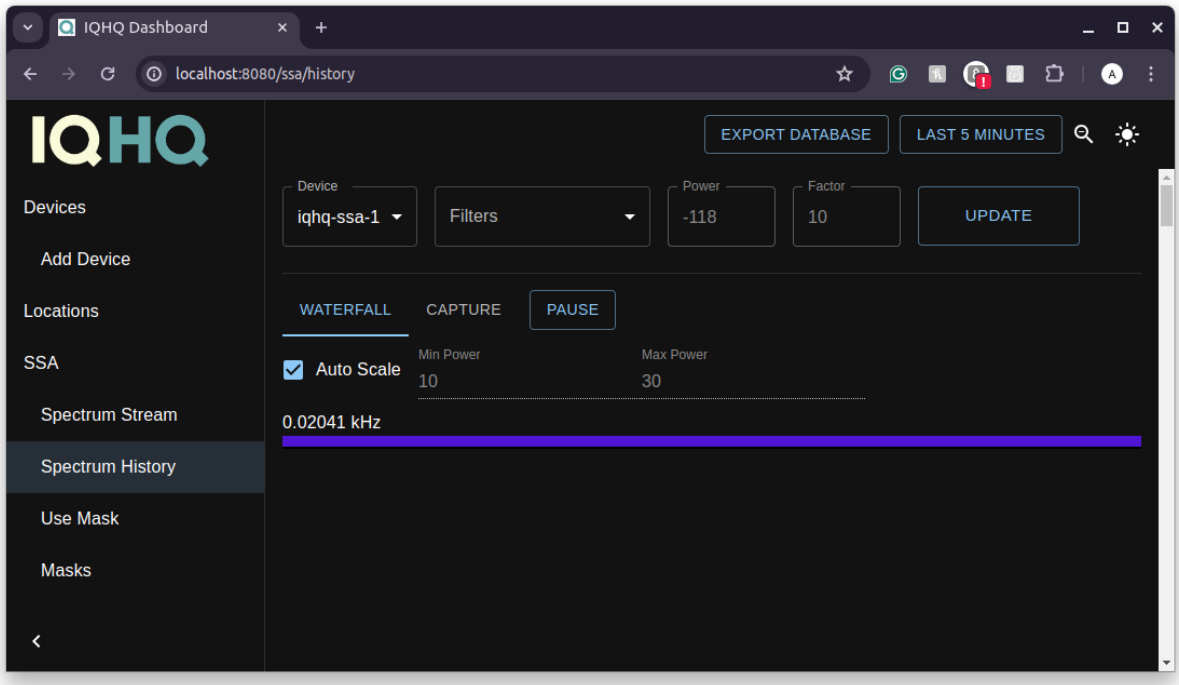


Figure 22 - Spectrum History Dashboard UI.

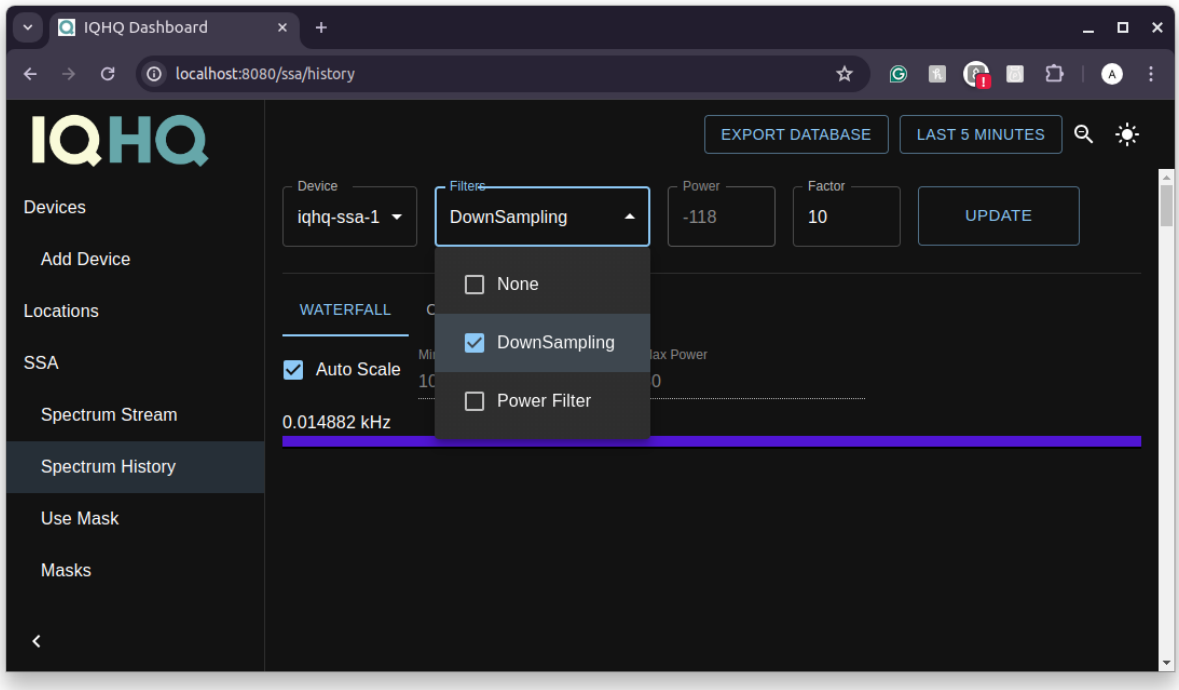


Figure 23 - Spectrum history dashboard UI.

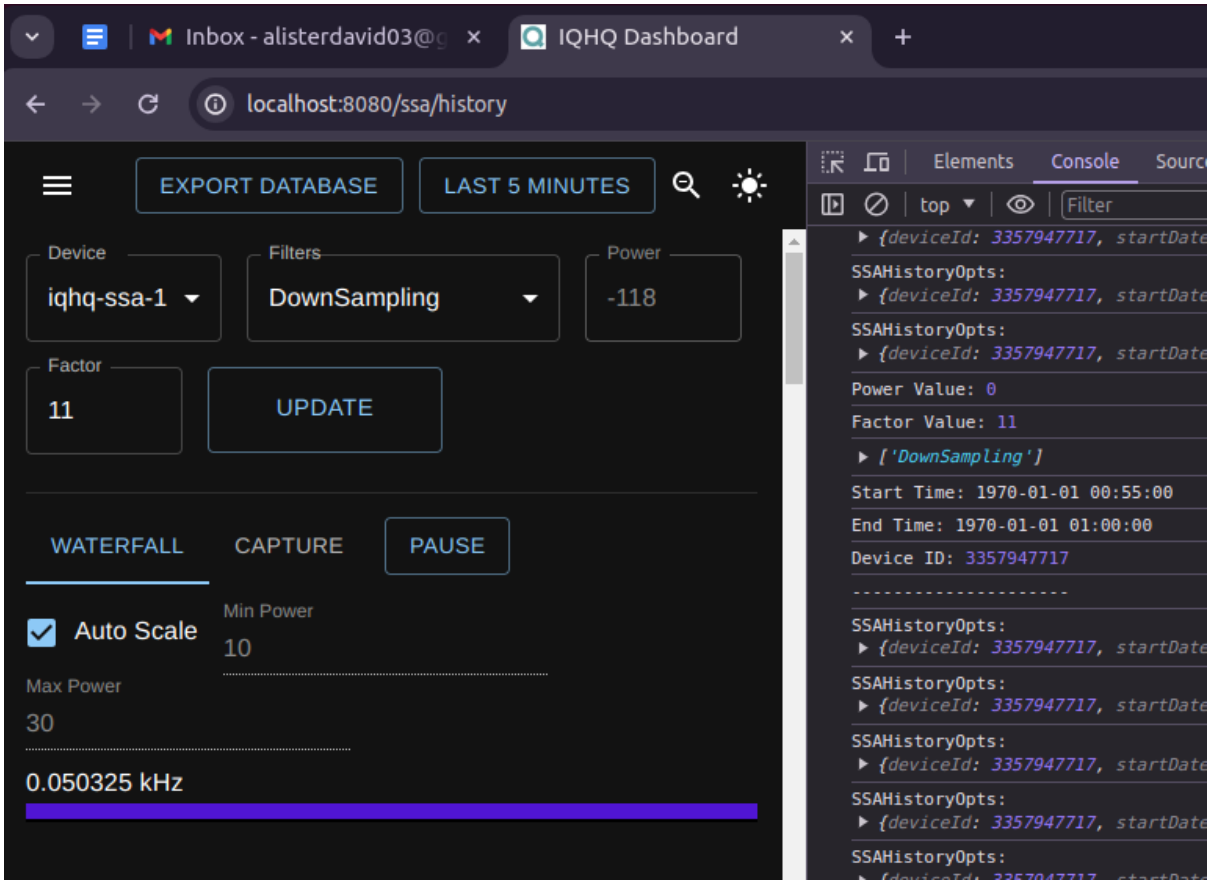


Figure 24 - Spectrum history log.

Initial back-end implementation of data retrieval:

I created a file called GetSSAHistory.ts and placed in the model's directory which is where the other back end SSA files are located. Within this file I created an async function called GetSSAHistory which at the moment takes a promise of any. This function returns SSA data from the database based on the given query. The query selects power and frequency from the ssa_metrics table from the database using deviceId and between the start time and end time. After testing this function and logging the result, it was clear that the SSA data was being retrieved.

```
57  //function that returns filtered ssa data from database
58  async function GetSSAHistory(): Promise<any[]> {
59    try {
60      const client = await pool.connect();
61      console.log('querying...')
62      const queryData: QueryResult = await client.query(`SELECT power, frequency FROM ssa_metrics
63      WHERE deviceID = $1
64      AND "time" >= $2
65      AND "time" <= $3
66    `, [deviceID, startDate, endDate]);
67      client.release();
68      let ssaData = queryData.rows
69      console.log("Raw Data Length:",ssaData.length)
70      const filteredData = applyFilters(ssaData, filter, powerValue, factor);
71      //console.log(filteredData)
72      console.log("Filtered Data Length:",filteredData.length)
73      return filteredData;
74    } catch (error) {
75      console.error('Error fetching data from database:', error);
76    }
77  }
```

Figure 25 - GetSSAHistory function that returns filtered SSA data.

The next step was to create the filters, I began by working on the down sampling filter. I created a function called downSampling which has two parameters, one being data and another being a factor. The function iterated over the data but instead of checking every item, it skips ahead by the specified factor each time. For each of these skipped steps, it picks the current item and adds it to the new list. It then returns a down sampled data set.

```
16  //functions that downsamples a list by factor of 10
17  function downSampling(data, factor) {
18    const filteredData = []
19    for (let i = 0; i < data.length; i += factor) {
20      filteredData.push(data[i])
21    }
22    return filteredData;
23  }
```

Figure 26 - downSampling function that down samples a data set by a given factor.

The thresholdFiltering function takes two parameters, data, and threshold. It contains a list to store the new values of SSA data. This function iterates through the data set and if the power is above the threshold, then it will be pushed to the new list. The list containing the filtered data is returned.

```
25  function thresholdFiltering(data, threshold) {
26    const filteredData = []
27    for (let i = 0; i < data.length; i++) {
28      if (data[i].power > threshold) {
29        filteredData.push(data[i])
30      }
31    }
32    return filteredData;
33  }
```

Figure 27 - thresholdFiltering function that filters out power frequency pairs where the power is below the threshold.

The applyFilters function processes a list of data by applying specified filters in sequence. It accepts the original data, an array of filter names, a power threshold, and a downsampling factor. It starts with the original data and iterates through the list of filters. If a filter is 'Power Filter', it uses the thresholdFiltering function to keep only items with a power value greater than the given threshold. If a filter is 'DownSampling', it uses the downSampling function to reduce the list by selecting every nth item based on the provided factor. The function returns the data after all specified filters have been applied. Now I can simply call on the applyFilters function in my GetSSAHistory function to apply the filters chosen by the user.

```
35  function applyFilters(data, filters, powerValue, factor) {
36    let filteredData = data;
37
38    for (const filter of filters) {
39      if (filter === 'Power Filter') {
40        filteredData = thresholdFiltering(filteredData, powerValue);
41      } else if (filter === 'DownSampling') {
42        filteredData = downSampling(filteredData, factor);
43      }
44    }
45
46    return filteredData;
47  }
```

Figure 28 - applyFilters function that applies the appropriate filters to SSA data.

To test that this function as well as the filters work, I logged the length of the raw data and the filtered data to make sure the results were as expected. I could also compare these results to the database, to ensure they are correct.

Testing the GetSSAHistory function:

I conducted tests to validate the GetSSAHistory function, ensuring it returns correctly filtered SSA data from the database. Each test focused on a specific filtering option, comparing the results against the raw data and the data from the database. Here are the results:

Test 1: Down Sampling

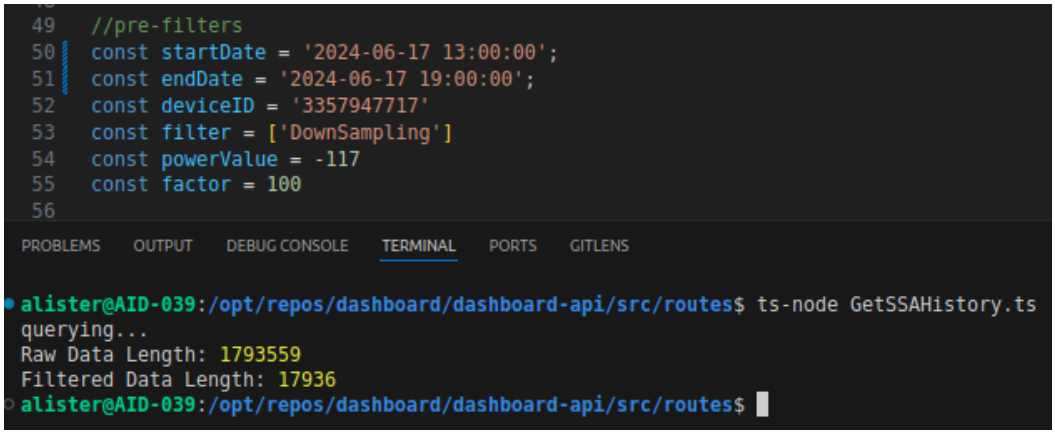


Figure 29 - Test result of down sampling.

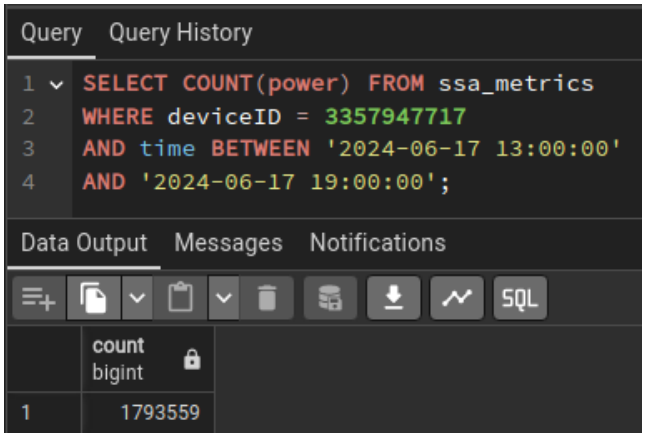


Figure 30 - Database query result of SSA data count.

Test Outcome: Success

Test 2: Threshold Filtering

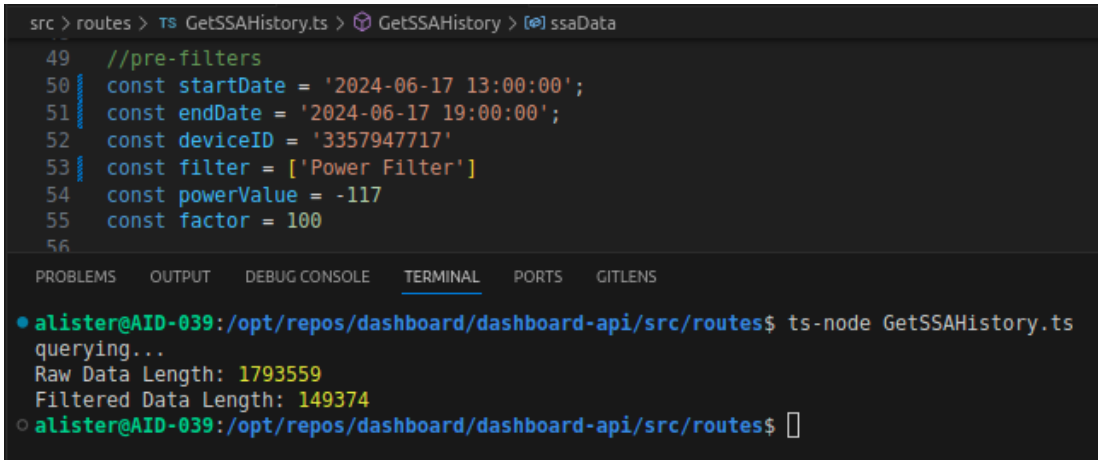


Figure 31 - The result of threshold filtering.

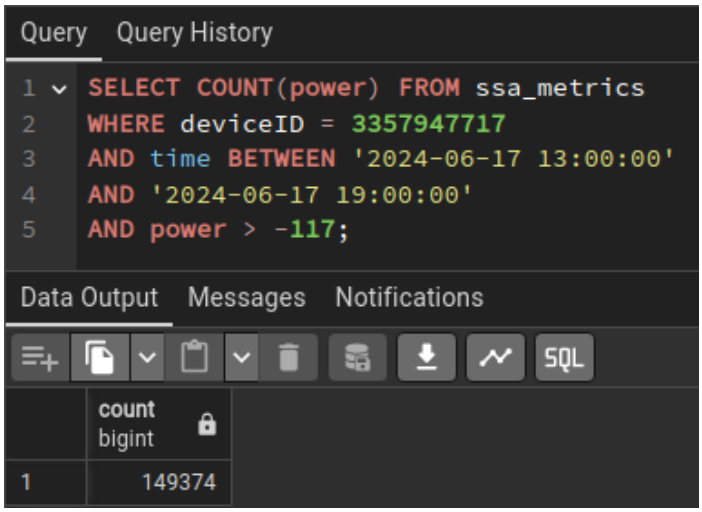


Figure 32 - Database query result of SSA data count where power is above the threshold value .

Test Outcome: Success

Test 3: Down Sampling & Threshold Filtering

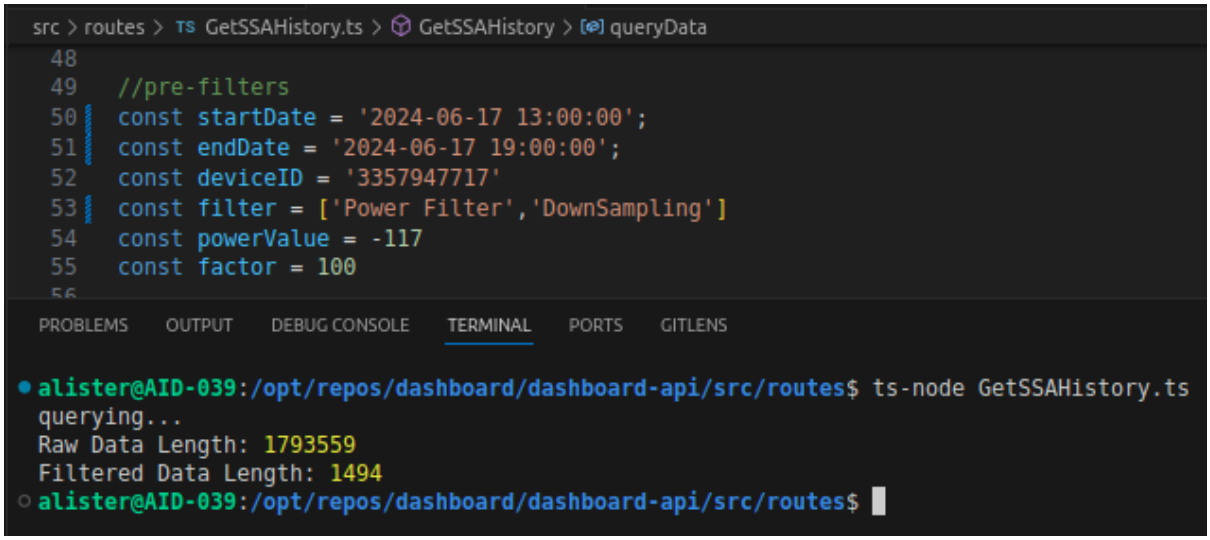


Figure 33 - The result of down sampling and threshold filtering.

Test Outcome: Success

Test 4: No Filters

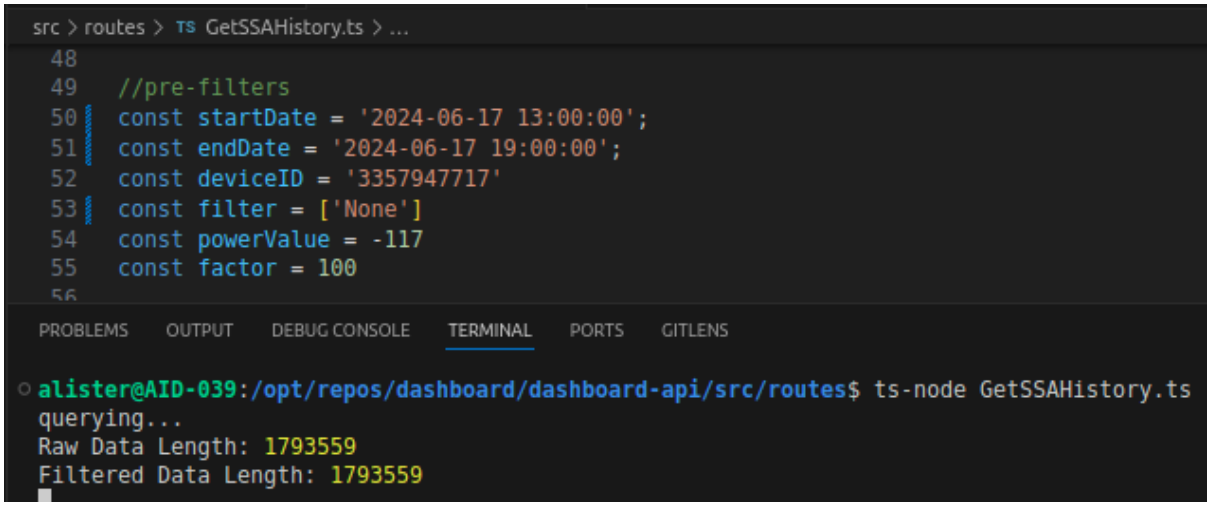


Figure 34 - The result of applying no filters.

Test Outcome: Success

These initial tests confirm that the GetSSAHistory function operates correctly, returning the expected results for each filtering option. The function effectively filters SSA data as specified, ensuring reliability and accuracy in the data retrieval process.

Creating a SSA history route:

The final step for this initial implementation of the data retrieval process was to establish to a route for the SSA history, enabling posting of spectrum history options and retrieving the corresponding SSA data.

In the Route.ts file within the SSA directory, which contains routes for posting and getting various SSA elements, I found the post route for the spectrum stream using /stream. This route sends the streaming options and uses a GET request to receive live SSA data from the spectrumStreamManager.

Seeing as though the spectrum stream and spectrum history functions in somewhat a similar manner, I decided to create an identical route post and get for the spectrum history. I used the route /history and replaced the post properties to suit the SSAHistoryOpts.

```
route.post("/history", ExpressBodyTypeCheck(yup.object({
    deviceId: yup.number().required(),
    startDate: yup.string().required(),
    endDate: yup.string().required(),
    filters: yup.array().required(),
    power: yup.number().required(),
    factor: yup.number().required(),
}).noUnknown(), async(body, req, res)=>{
    try{
        await spectrumStreamManager.setHistoryOpts(
            body,
        );
        res.status(200).send(body);
    }catch(e) {
        console.log("Failed to set stream opts: ", e);
        res.status(400).send();
    }
}))

route.get("/history", (req, res)=>{
    try{
        res.status(200).send(spectrumStreamManager.getHistoryOpts());
    }catch(e) {
        console.log("Failed to set stream opts: ", e);
        res.status(400).send();
    }
})
```

Figure 35 - Spectrum history POST and GET.

Since the spectrum stream and spectrum history functions operate similarly, I created identical routes for the spectrum history, using /history. I replaced the POST properties to suit SSAHistoryOpts. The spectrum stream post used a function called setStreamOpts from spectrumStreamManager.ts, and a corresponding getStreamOpts function. I created similar functions for the spectrum history route, named setHistoryOpts and getHistoryOpts.

Figure 36 - getHistoryOpts and setHistoryOpts function.

The spectrumStreamManager.ts file contained the SpectrumStreamManager class which has a private property called currentOpts. This private property contained the default values for the streaming options. I decided to create one for the history options called currentHistoryOpts, where I set some default values for the spectrum history options, as a test.

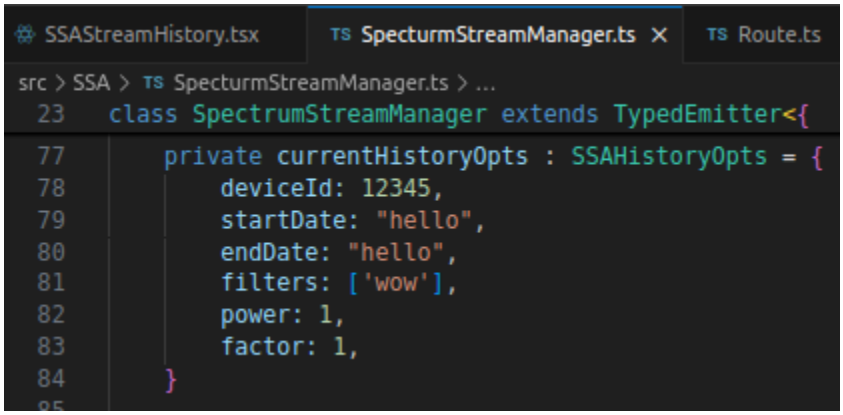


Figure 37 - Default values for spectrum history options.

I then added a custom hook, similar to the one found on SSAStreamView to post and get a response from the route into the front end.

```
const {
    loading,
    response,
    post: dotrigger,
} = useHTTPTrigger<
    any,
    SSAHistoryOpts
>("/ssa/history");
```

Figure 38 - useHTTPTrigger custom hook.

Issues with the history route:

During the development of the SSA data retrieval feature, I encountered a problem with the history route that required the expertise of a senior software engineer. This situation demonstrated the importance of clear and detailed communication with technical colleagues.

After creating the POST route for the spectrum history, I encountered an error with the body being passed to the setHistoryOpts function. The POST route was nearly identical to the streaming route, differing only in the options properties. Despite this similarity, the source of the error was unclear. I attempted to resolve the issue by searching online resources, but I could not find a solution.

With the rest of the software team on a trial in America, I couldn't consult them directly. My manager recommended using the Signal app to securely communicate

with my colleagues. This advice was particularly useful as it allowed for secure and efficient communication despite the geographic separation.

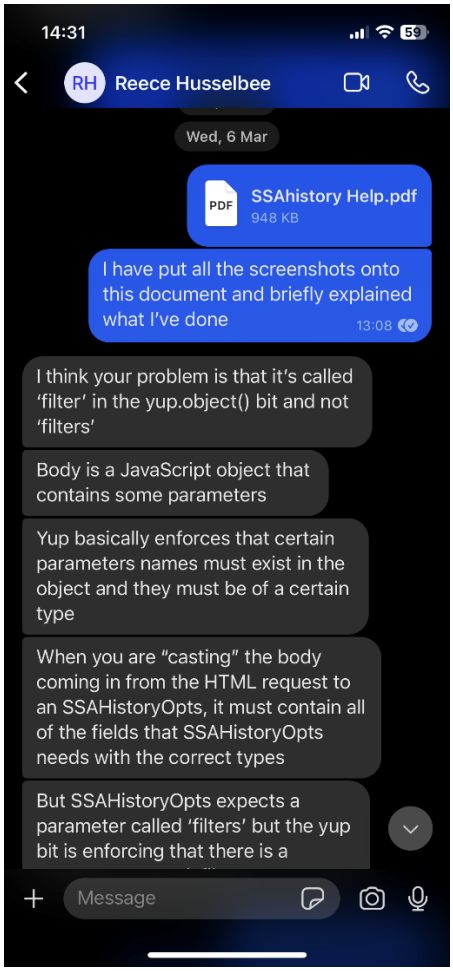
Communication Process:

To effectively communicate the issue to my senior software engineer colleague, I compiled a comprehensive document that included:

- **Detailed Description of the Issue**: I provided a clear explanation of the problem, including the specific error message and the circumstances under which it occurred.
- **Relevant Code Snippets**: I included the relevant portions of the code, highlighting the sections that were identical to the streaming route and the differences in the options properties.
- **Steps Taken to Resolve the Issue**: I outlined the steps I had already taken to try to resolve the issue, including my attempts to find solutions online.

I sent this document via Signal, along with a brief description of the task I was working on. This approach ensured that my colleague had all the necessary context to understand the problem and could provide informed advice.

Figure 39 - Error in the history route POST.



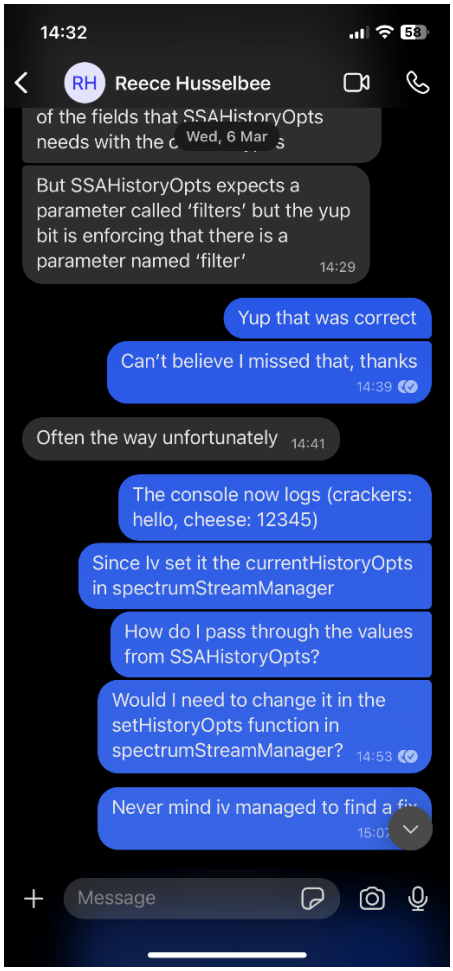


Figure 40 - Signal chat screenshot.

Figure 41 - Singal chat screenshot.

Thanks to the detailed information I provided, my colleague Reece quickly identified the issue as a minor spelling mistake. This simple oversight had caused the error, but without the fresh perspective and expertise of a senior engineer, it might have taken much longer to diagnose.

Once the spelling mistake was corrected, the history options were posted and received correctly, allowing me to proceed with the development of the feature.

Benefits of Using Signal for Technical Communication:

- **Security:** Signal's end-to-end encryption ensured that our communication was secure, protecting sensitive project details.
- **Efficiency:** The ability to send detailed documents and code snippets allowed for efficient troubleshooting and problem resolution.
- **Collaboration:** Despite being in different locations, we were able to collaborate effectively, leveraging the instant messaging capabilities of Signal to discuss and resolve the issue in real time.

In conclusion, the experience of resolving the history route issue highlighted the importance of clear, detailed communication with technical stakeholders. By using Signal to share comprehensive information, I was able to leverage my colleague's

expertise to quickly identify and fix the problem. This approach not only facilitated the resolution of the issue but also reinforced the value of effective communication in technical problem-solving.

Updating the UI and restructuring the history options:

To enhance the efficiency of tracking selected filters, I decided to switch from using arrays to using boolean values. This adjustment would simplify the code by eliminating the need for extensive array restructuring. Consequently, I opted to replace the dropdown filter options with checkboxes, enabling users to select the desired filters by ticking the corresponding boxes.

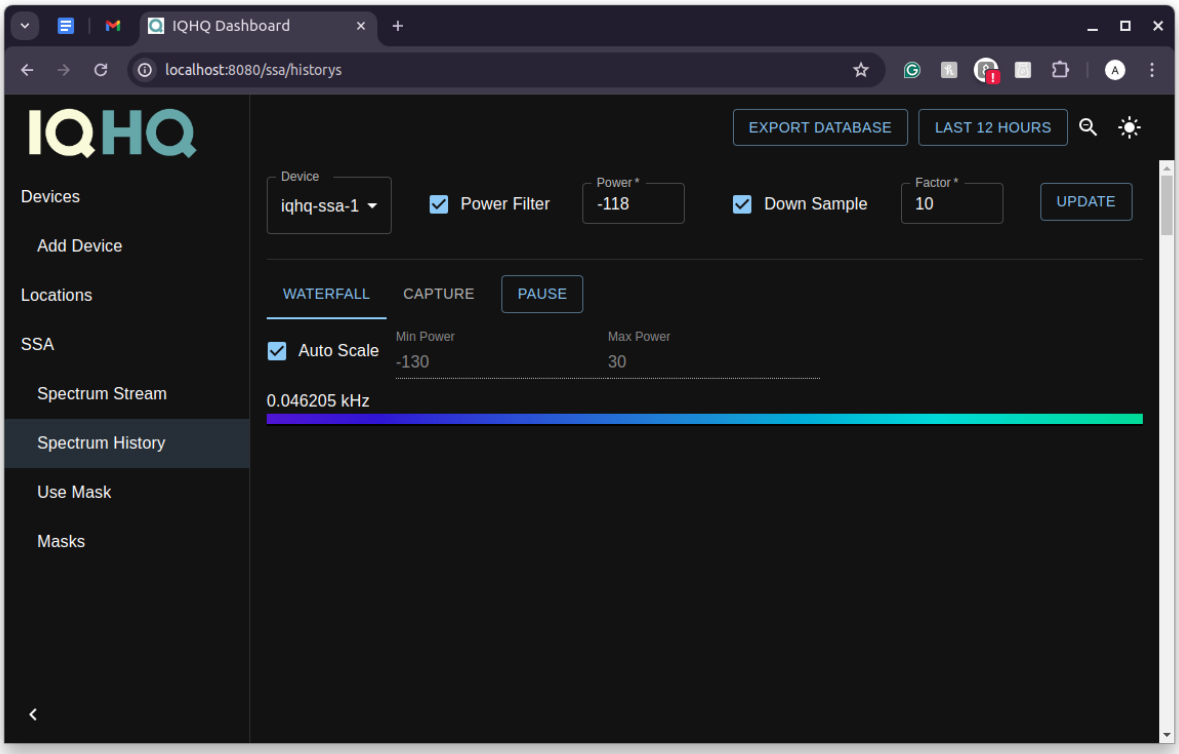


Figure 42 - Update spectrum history UI.

I began by removing the previously implemented React elements and starting fresh. I introduced a checkbox labelled "Power Filter" and a corresponding text field labelled "Power," allowing users to input a threshold value for filtering. Similarly, I added another checkbox labelled "Down Sample" and a text field labelled "Factor," enabling users to specify a down-sampling factor. To manage the state of these checkboxes, I utilised the useState hook.

```
//useState to handle power filter and down sample check boxes
const [powerFilterChecked, setPowerFilterChecked] = useState(false);
const [downSampleChecked, setDownSampleChecked] = useState(false);

const handlePowerFilterChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setPowerFilterChecked(event.target.checked);
};

const handleDownSampleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setDownSampleChecked(event.target.checked);
};
```

Figure 43 - useState to handle power filter and down sample check boxes.

```
<Grid item style={{marginTop: 6, marginLeft: 20}}>
    <FormControlLabel
        control={
            <Checkbox
                {...register("powerFilter")}
                onChange={handlePowerFilterChange}
                checked={powerFilterChecked}
            />
        }
        label="Power Filter"/>
</Grid>
<Grid item><TextField required
    type="number"
    inputProps={{
        min: -400,
    }}
    label="Power"
    disabled={!powerFilterChecked}
    size="small"
    style={{ width: '100px', marginTop:6}}
    {...register("power", {valueAsNumber: true})}
/></Grid>
```

Figure 44 - React check box and text field.

```
<Grid item style={{marginTop: 6, marginLeft: 30}}>
    <FormControlLabel
        control={
            <Checkbox
                {...register("downSampleFilter")}
                onChange={handleDownSampleChange}
                checked={downSampleChecked}
            />
        }
        label="Down Sample"/>
</Grid>
<Grid item><TextField required
    type="number"
    inputProps={{
        min: 2,
    }}
    label="Factor"
    disabled={!downSampleChecked}
    size="small"
    style={{ width: '100px', marginTop:6}}
    {...register("factor", {valueAsNumber: true})}
/></Grid>
```

Figure 45 - React check box and text field.

The history options were initially not being propagated through the program because they were not included in the POST request within the route. To resolve this, I added the history option properties to the dotrigger function, ensuring that each value is sent when the update button is clicked.

```
return <form
    onSubmit={handleSubmit(values=>{
        setError(undefined);

        let deviceId = isFinite(values.deviceId as number)
            ? values.deviceId as number
            : undefined
        dotrigger({
            ...values,
            power: values.power,
            factor: values.factor,
            powerFilter: powerFilterChecked,
            downSampleFilter: downSampleChecked,
            deviceId: deviceId,
            startDate: formattedStartTime,
            endDate: formattedEndTime,
        })
            .then(()=>{
                if(isMounted()) {
                    props.onComplet()
                    props.setStreamEnabled(deviceId !== undefined)
                    console.log("\nDEVICE ID          :", deviceId)
                    console.log("START DATE         :",formattedStartTime)
                    console.log("END DATE           :",formattedEndTime)
                    console.log("POWER FILTER       :",powerFilterChecked)
                    console.log("DOWN SAMPLE FILTER :",downSampleChecked)
                    console.log("POWER              :",values.power)
                    console.log("FACTOR             :",values.factor)
                    setPowerFilterChecked(false);
                    setDownSampleChecked(false);
                }
            })
    })}
>
```

Figure 46 - Added history option properties to the dotrigger function.

I then updated the SSAHistoryOpts type, replacing the array for filters with two new boolean properties: powerFilter and downSampleFilter. Subsequently, I updated all instances where SSAHistoryOpts was used throughout the codebase.

```
//function that applies the filters and returns the filtered data
function applyFilters(data, powerFilter, downSampleFilter, powerValue, factor) {
  let filteredData = data;
  if (powerFilter) {
      filteredData = thresholdFiltering(filteredData, powerValue);
  }
  if (downSampleFilter) {
      filteredData = downSampling(filteredData, factor);
  }
  return filteredData;
}
```

Figure 47 - Update applyFilters function.

```
type SSAHistoryOpts = {
    //No device should be undefined, need to find a way around to remove undefined
    deviceId: number | undefined,
    startDate: string,
    endDate: string,
    powerFilter: boolean,
    downSampleFilter: boolean,
    power: number,
    factor: number,
}

export type {
    SSAHistoryOpts
}
```

Figure 48 - Updated SSAHistoryOpts type.

```
34  route.post("/history", ExpressBodyTypeCheck(yup.object({
35      deviceId: yup.number(),
36      startDate: yup.string().required(),
37      endDate: yup.string().required(),
38      powerFilter: yup.boolean().required(),
39      downSampleFilter: yup.boolean().required(),
40      power: yup.number().required(),
41      factor: yup.number().required(),
42  }).noUnknown(), async(body, req, res)=>{
43      try{
44          await spectrumStreamManager.setHistoryOpts(
45              body,
46          );
47          res.status(200).send(body);
48      }catch(e) {
49          console.log("Failed to set stream opts: ", e);
50          res.status(400).send();
51      }
52  }))
```

Figure 49 - Updated route.

```
49  //pre-filters
50  const startDate = '2024-02-26 16:31:00';
51  const endDate = '2024-02-26 16:33:00';
52  const deviceId = '3357947717'
53  const powerFilter = true
54  const downSampleFilter = true
55  const power = -117
56  const factor = 100
57
58  //const GetSSAHistory = async (deviceId,startDate,endDate,powerFilter,
59  //async function GetSSAHistory(): Promise<SpectrumStreamsData> {
60
61  const GetSSAHistory = async (deviceId,startDate,endDate,powerFilter,
62    downSampleFilter,power,factor): Promise<SpectrumStreamsData> => {
63    try {
64      const client = await pool.connect();
65      console.log('querying...')
66      const queryData: QueryResult = await client.query(
67        `SELECT power, frequency FROM ssa_metrics
68        WHERE deviceID = $1
69        AND "time" >= $2
70        AND "time" <= $3`,
71        [deviceId, startDate, endDate]      You, 3 months ago • front en
72      );
73      client.release();
74      const ssaData = queryData.rows;
75      const filteredData = applyFilters(ssaData, powerFilter,
76        downSampleFilter, power, factor)
77
78      //converting filtered data to match SpectrumStreamsData type
79      const spectrumStreamsData: SpectrumStreamsData = {}
80      filteredData.forEach((streamId, Key) => {
81        spectrumStreamsData[Key] = streamId
82      })
83
84      console.log(ssaData.length)
85      console.log(spectrumStreamsData)
86      console.log(filteredData.length)
87
88      //return filteredData;
89      return spectrumStreamsData;
90    } catch (error) {
91      console.error('Error fetching data from database:', error);
92    }
93  }
94
95  GetSSAHistory(deviceId,startDate,endDate,powerFilter,
96    downSampleFilter,power,factor);
97
98  //export default GetSSAHistory;
99
100 export {
101   GetSSAHistory
102 }
```

Figure 50 - Updated GetSSAHistory function.

This restructuring and UI update streamlined the process of managing filter options, enhancing both the user experience and the maintainability of the code.

Connecting the front end to the back end:

The next stage in the development process was to connect the front end to the back end, ensuring that when the update button is clicked, it triggers the GetSSAHistory function.

I started by creating a new route in the SSAStreamHistory file, /streamHistory, using the useHTTPTrigger hook. This allows the front end to post history options so they can be accessed in the back end. I then included all the history option values in the onSubmit function, within the post to ensure they are sent when the update button is clicked.

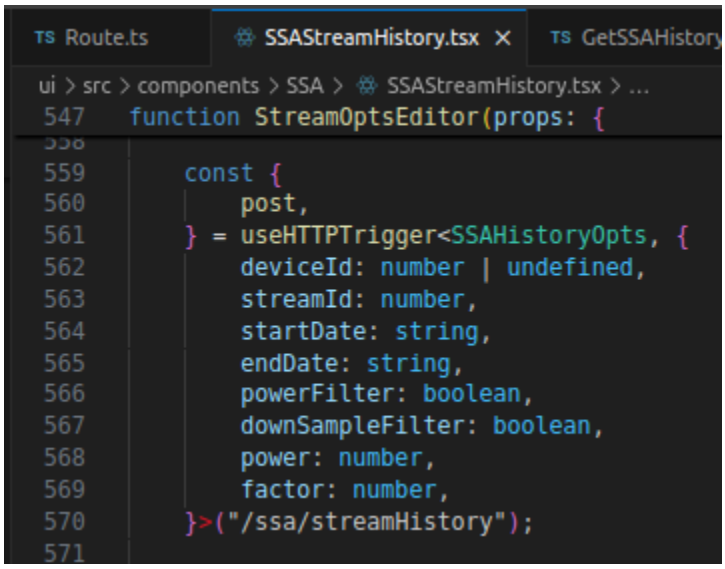


Figure 51 - useHTTPTrigger hook to post history options.

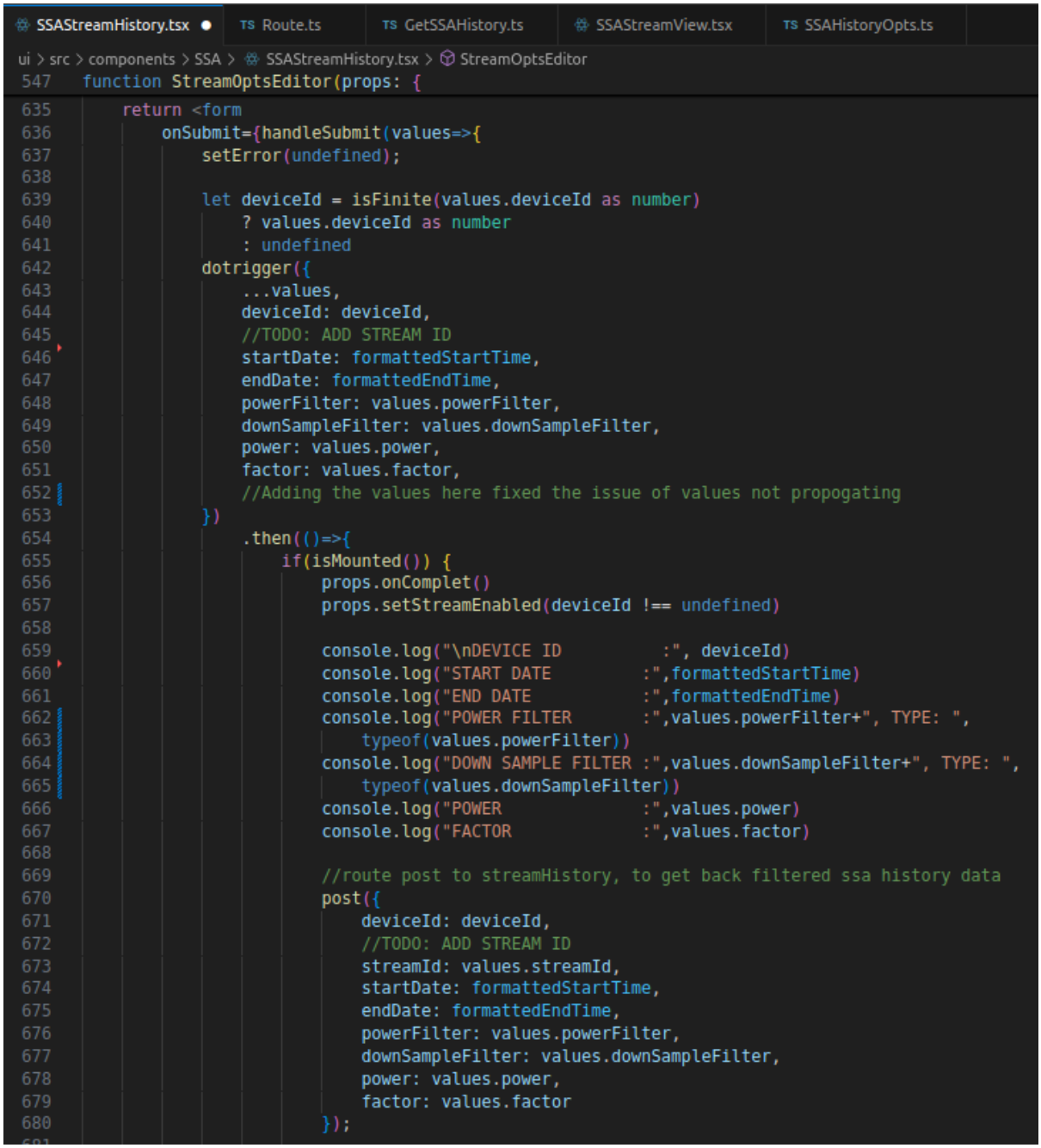


Figure 52 - Posting history options when the update button is clicked.

In the Route.ts file, I created a new POST route using the /streamHistory endpoint to retrieve the filtered data. This route is similar to the initial history route POST but differs in that it awaits and uses the GetSSAHistory function within the try block to fetch the filtered data. With this update, the filtered SSA data gets logged in the back end when the options are set and updated from the front end.

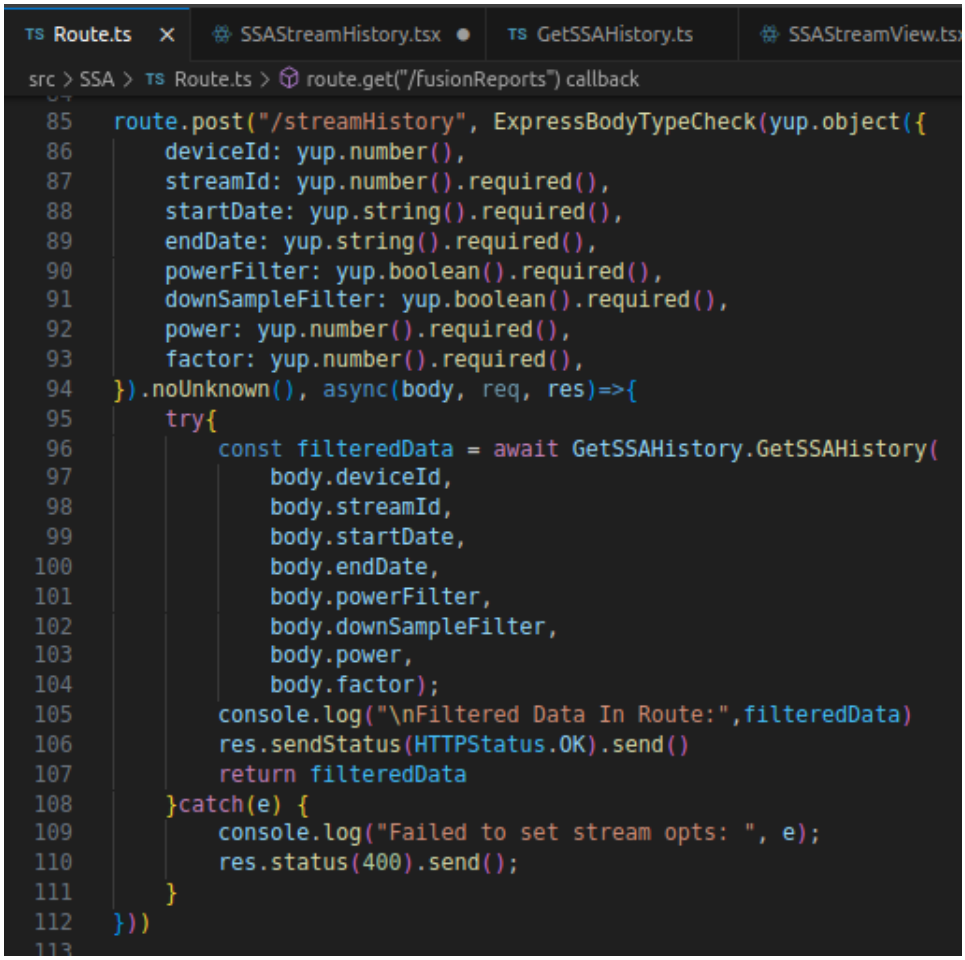


Figure 53 - streamHistory route.

This integration effectively links the user interface with the back-end logic, ensuring that all the specified filter options are correctly applied and processed. The next step is being able to access the filtered data in the front end from the back end.

Formatting the filtered SSA data:

To ensure the filtered SSA data is properly formatted before it is received on the front end, I aimed to match the format expected by the existing makeData function. This function is used to convert data for use by the waterfall and graph components. By studying the SSAStreamView code, I determined that the data was of the type SpectrumStreamsData, a nested dictionary where each stream ID maps to another object containing frequency-power level pairs. I verified this format by logging the live raw data received by the SSAStreamView function.

```
type SpectrumStreamsData = {
    [streamId: number]: {[key: string]: number,}
}
```

Figure 54 - SpectrumStreamsData type.

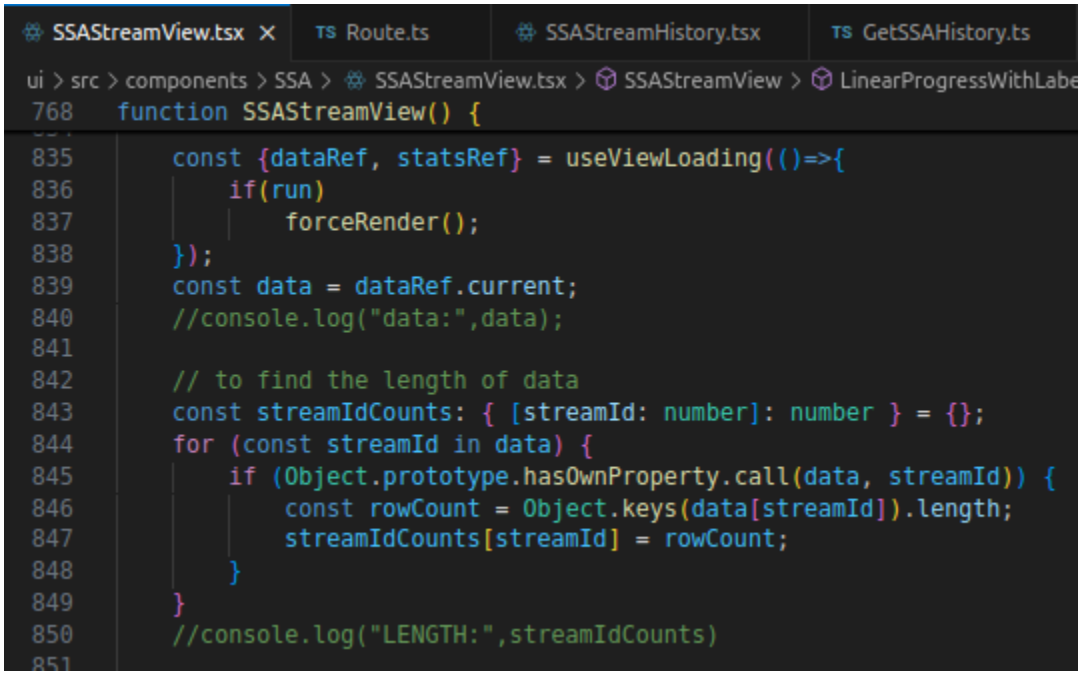


Figure 55 - Logging live raw SSA data.

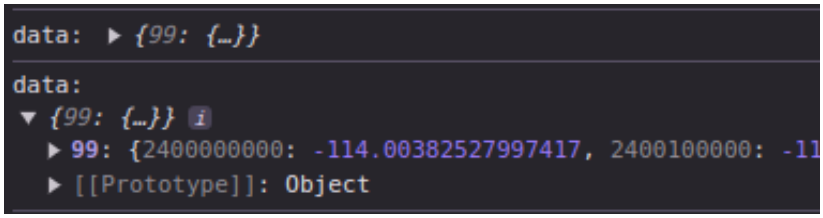


Figure 56 - Live raw SSA data log.

I then created an identical nested dictionary object named HistoryStreamsData. Since this type requires a stream ID, I added a stream ID property as a number to the SSAHistoryOpts type and updated all instances where this type was used. For the time being, I assigned a random number to the stream ID, as this option was not yet configurable from the front end.

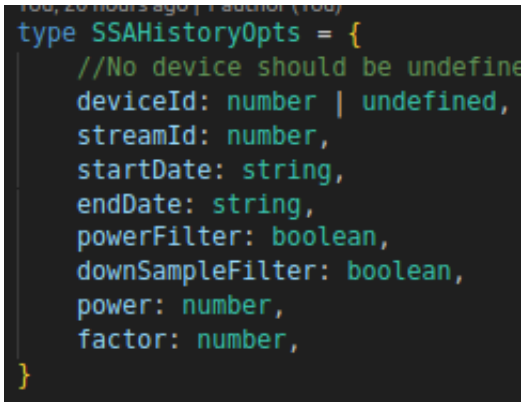


Figure 57 - Updated SSAHistoryOpts type.

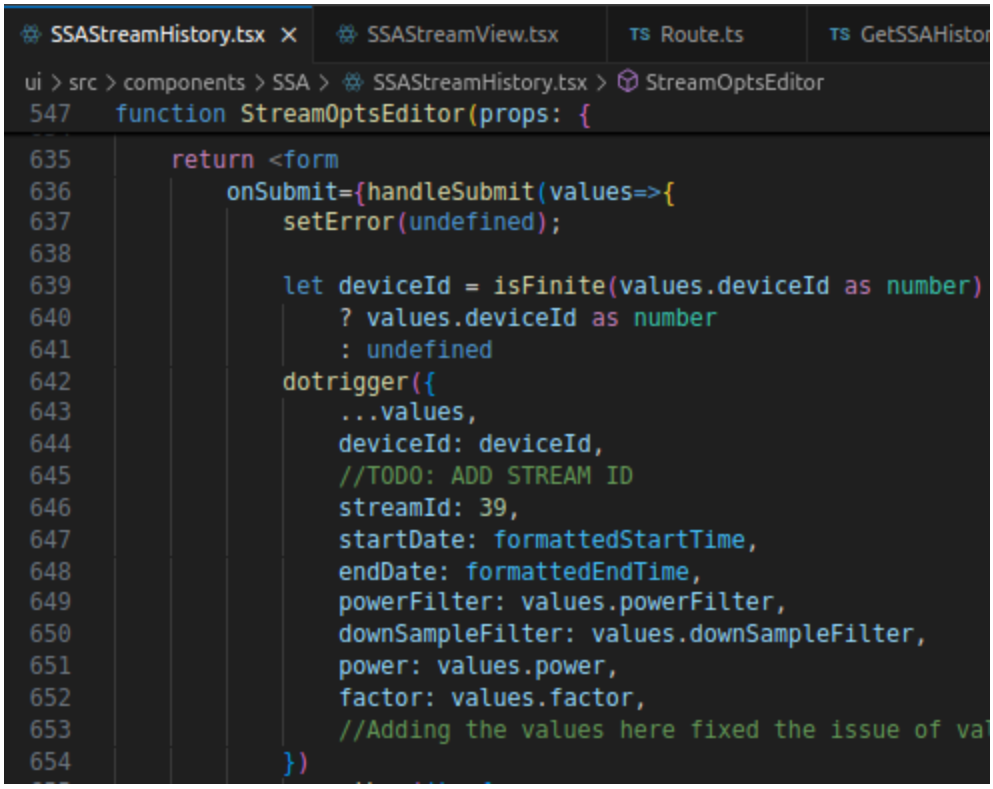


Figure 58 - Adding streamId to post the history options.

In the GetSSAHistory function I added code to format the filtered data into the HistoryStreamsData type. For each data point, it ensures that there is an object entry for the specified streamId. Then, it adds an entry within that object where the key is the frequency (as a string), and the value is the power. This results in a nested object structure where each stream ID maps to an object of frequency-power pairs.

```
//converting filtered data to match HistoryStreamsData type
const HistoryStreamsData: HistoryStreamsData = {};

filteredData.forEach(dataPoint => {
  const { frequency, power } = dataPoint;
  if (!HistoryStreamsData[streamId]) {
    HistoryStreamsData[streamId] = {};
  }
  HistoryStreamsData[streamId][frequency.toString()] = power;
});
```

Figure 59 - Formatting filtered SSA data into the HistoryStreamsData type.

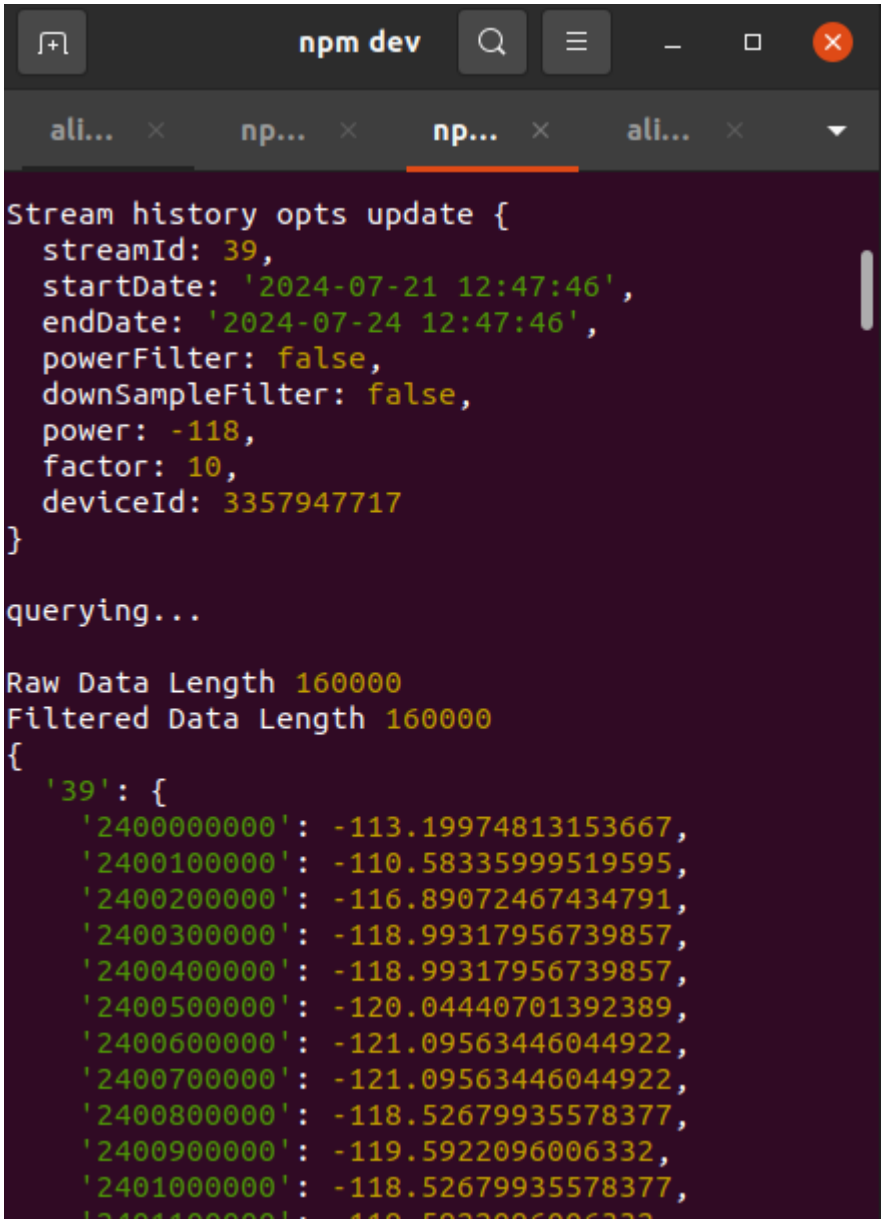


Figure 60 - GetSSAHistory function log.

This update ensures that the filtered SSA data is formatted correctly for seamless integration with the front-end components, thereby maintaining consistency and ease of data handling across the application.

Getting filtered data into the UI:

With the back-end processing of filtered data functioning correctly, the next logical step is to integrate this data into the front end for display in the waterfall plot and power frequency graph. Callum, a senior software engineer and the original developer of the dashboard, provided guidance on how to accomplish this.

In the useHTTPTrigger hook, I added a response property in addition to post. I passed in the parameters for the expected response (HistoryData) and the data expected to be sent in the request body (SSAHistoryOpts).

I implemented the onComplete and onResponse callback functions passed through the props object:

- The onComplete function is invoked upon successfully fetching the history data, passing the resulting data back to the parent component.
- The onResponse callback handles any kind of response from the operations within the component, such as errors or status messages, allowing the parent component to appropriately react to those responses.

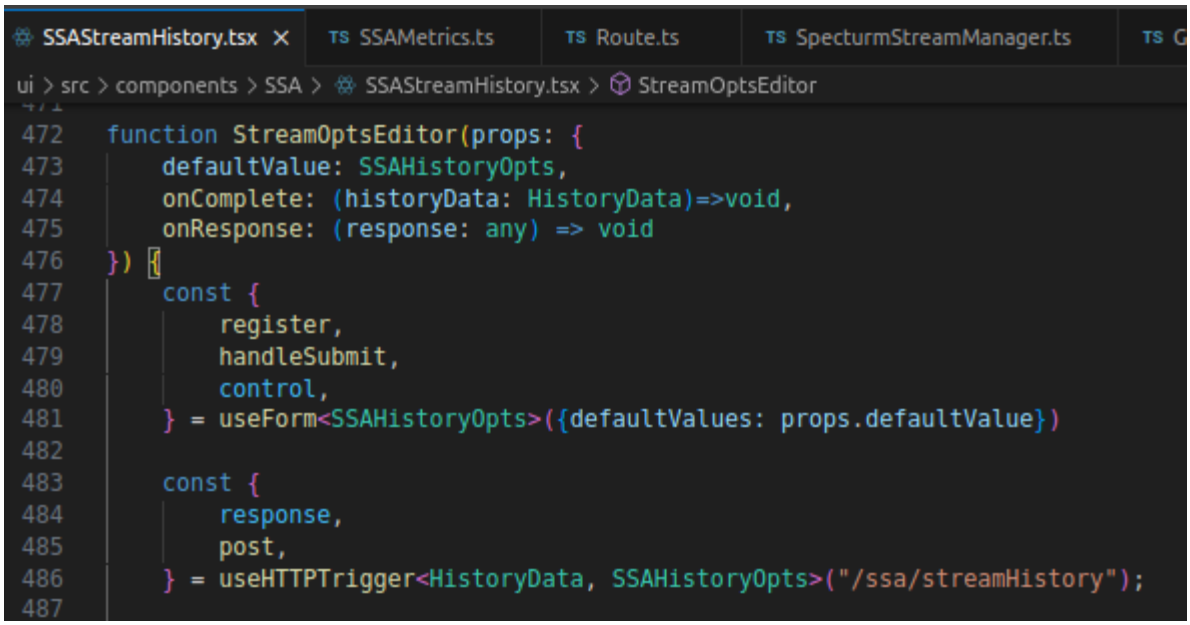


Figure 61 - Updated useHTTPTrigger hook and updated StreamOptsEditor function with callbacks.

When a response is received successfully the onResponse function is used to set the data as a prop so that it can be accessed elsewhere. I removed the dotrigger function from the handleSubmit method, which was part of the previous useHTTPTrigger hook that is no longer in use. Consequently, I deleted the StreamOptsView function, which was initially used to load and set the history options.

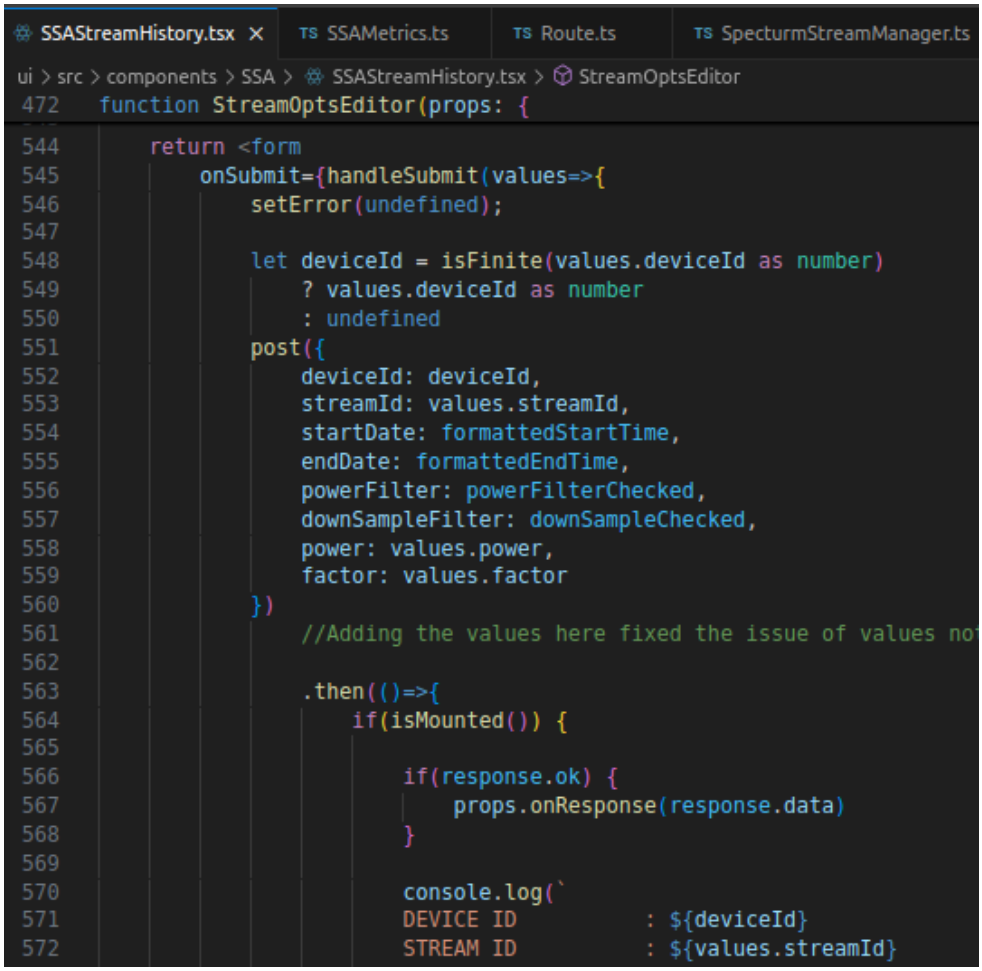


Figure 62 - Updated handleSubmit method.

In the SSAStreamHistory function, I accessed the filtered data through the onReponse function. After logging the data, I confirmed that the process was now working correctly.

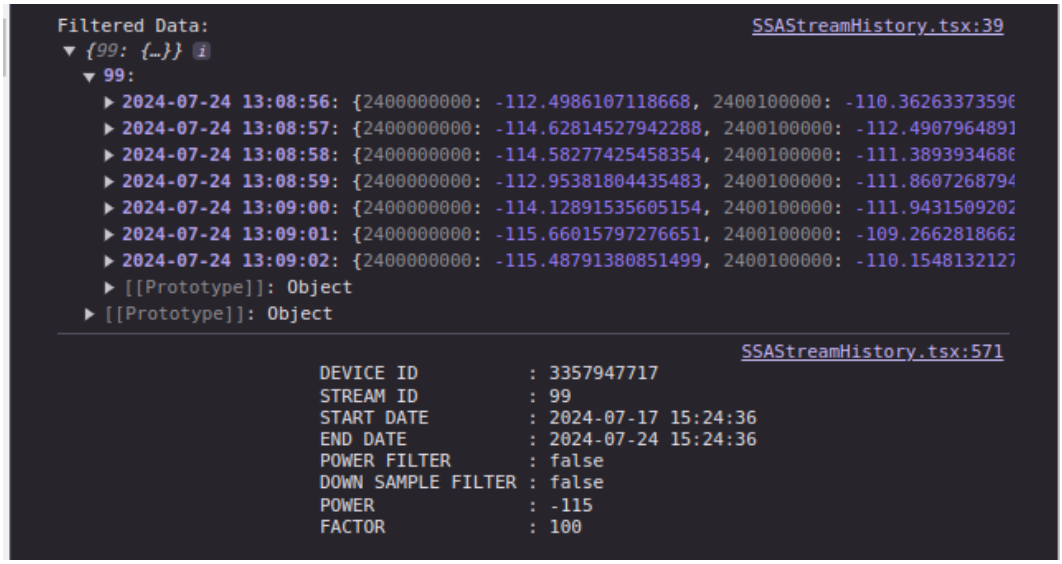


Figure 63 - Log from SSAStreamHistory function.

Adding stream Id to the database:

The stream Id is currently not part of the ssa_metrics table, therefore I updated the SSAMetrics.ts file to include stream Id. I first updated the SSAMetric type to include stream Id, then I consequently updated all the area where this type was being used. I updated the createTableIfNotExists function to add stream Id as a INT. I was able to access the stream Id from the WebSocketInterfacePacket type within the insertSSAMetrics function. Finally, I dropped the old ssa_metrics table and run the dashboard application to induce the creation of the updated table. This ensured that stream Id was being added to the database when SSA data was being captured.

```
type SSAMetric = {
    deviceId: number,
    streamId: number
    time: number,
    power: number,
    frequency: number,
}
```

```
function createTableIfNotExists() {
    return connection.runQuery(`
        CREATE TABLE IF NOT EXISTS ${tableName} (
            deviceId BIGINT NOT NULL,
            streamId INT NOT NULL,
            time TIMESTAMPTZ NOT NULL,
            power FLOAT NOT NULL,
            frequency FLOAT NOT NULL
        );
    `, []);
}
```

Figure 64 - Updated SSAMetric type. Figure 65 - Updated ssa_metrics table.

```
//function to insert ssametric into database
async function insertSSAMetrics(
    metrics: WebSocketInterfacePacket
): Promise<SSAMetric | SSAMetric[]> {
    const values: any[] = [];
    const useValue = (val: any): string => {
        values.push(val);
        return `$${values.length}`;
    };

    //const metricsArray = Array.isArray(metrics) ? metrics : [metrics];
    const time = new Date().getTime()
    const device = metrics.deviceId
    const stream = metrics.streamId

    let ssaMetrics : SSAMetric[] = Object.entries(metrics.powerLevels).map(([key, value])=>
        ({
            deviceId: device,
            streamId: stream,
            time: time,
            power: value,
            frequency: Number(key)
        })
    )
```

Figure 66 - Updated insertSSAMetrics function.

Updating the HistoryData type:

After a brief code review with Callum, we decided to use a more integrable format for the filtered data to be incorporated with the waterfall plot and graphing function. The HistoryData type which was previously called HistoryStreamsData was updated to also use time within the nested structure.

Top-Level Object (HistoryData):

- The top-level object is a dictionary where the keys are stream Ids
- Each key maps to another object that represents time-based records.

Second-Level Object (Time Records):

- Each stream Id maps to an object where the keys are time values.
- Each time value key maps to another object that represents frequency-based records.

Third-Level Object (Frequency Records):

- Each time value key maps to an object where the keys are frequency values.
- Each frequency value key maps to a power level.

```
type HistoryData = {
    [streamId: number]: {
        [time: string]: {
            [frequency: string]: number;
        };
    };
};
```

Figure 67 - HistoryData type.

I added a function called formatFilteredData to the GetSSAHistory.ts file. This function formats the filtered data into the HistoryData type.

Function Breakdown

1. **Initialisation**:
   - count is initialised to keep track of the number of processed data entries.
   - historyData is initialised as an empty object that will be populated with the formatted data.
2. **Iterating Over Filtered Data**:
   - The function loops through each object (row) in the filteredData array.
3. **Extracting and Formatting Data**:
   - streamId is extracted and converted to an integer.
   - time is extracted, converted to an ISO string, and formatted to exclude the milliseconds and replace the 'T' with a space.
   - frequency is extracted and converted to a string.
   - power is extracted as-is.
4. **Initialising Nested Structure**:
   - If historyData does not have an entry for the current streamId, it initialises an empty object for that streamId.
   - If the streamId entry does not have an entry for the current time, it initialises an empty object for that time.
5. **Populating the Data**:
   - The power value is assigned to the appropriate frequency key within the time and streamId entries.
   - The count is incremented to track the number of data points processed.
6. **Logging and Returning the Result**:
   - The function logs the total number of data points formatted.

- The formatted historyData object is returned.

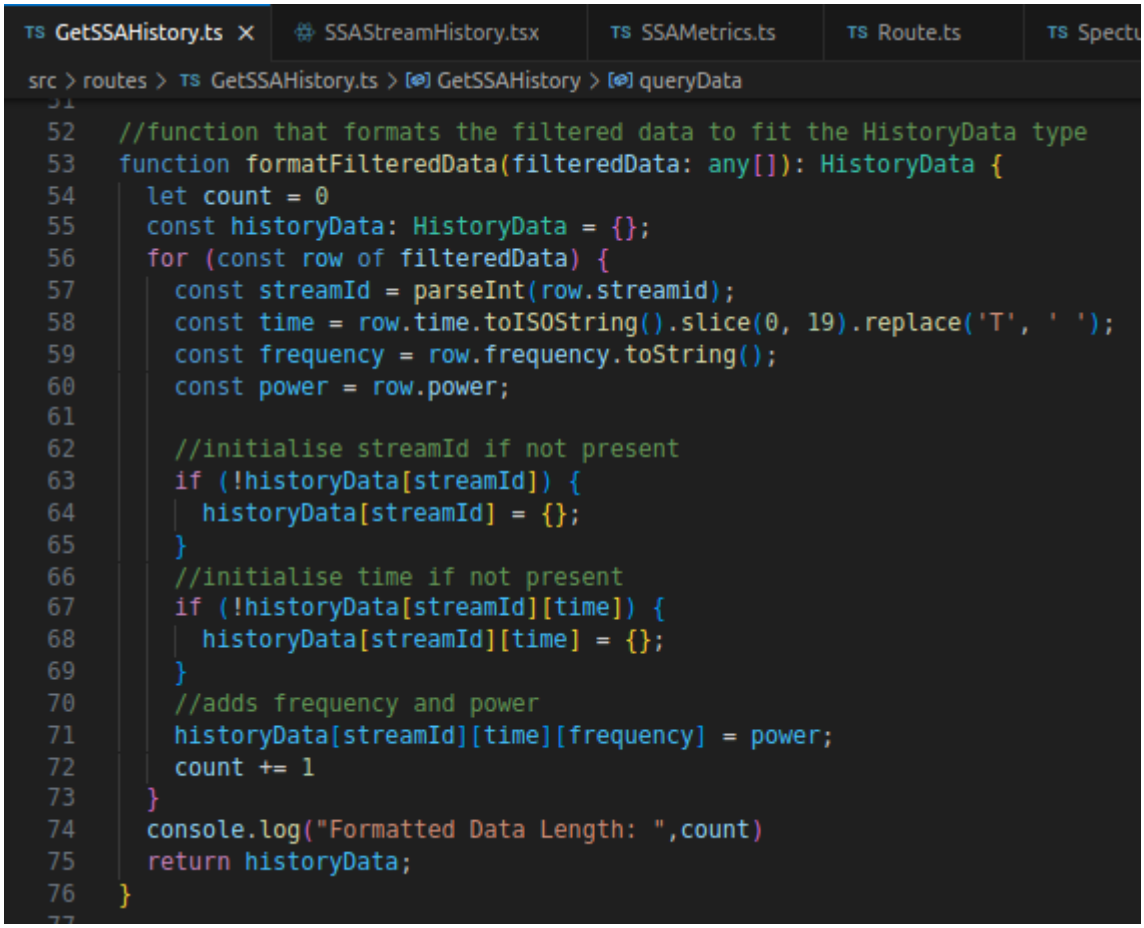


Figure 68 - Function that formats the filtered data into the historyData type.

I add time and stream Id to the database query to ensure the raw data returned time and stream Id as well so that it can be used in the formatting function. I also updated the query to order the data by time to ensure the data would be displayed in chronological order. After running some tests this function is working as intended. Now I can simply call on this function after applying the filters to format the data and return it.

```
    const queryData: QueryResult = await client.query(
      `SELECT streamId, "time", frequency, power FROM ssa_metrics
      WHERE deviceId = $1
      AND "time" BETWEEN $2 AND $3
      ORDER BY "time" ASC;`,
      [deviceId, startDate, endDate]
    );
    client.release()

    const ssaData = queryData.rows

    console.log("Raw Data Length:",ssaData.length)

    const filteredData = applyFilters(ssaData, powerFilter, downSampleFilter, power, factor)
    const formattedData = formatFilteredData(filteredData)

    console.log(formattedData)
    return formattedData
  } catch (error) {
    console.error('Error fetching data from database:', error);
  }
}
```

Figure 69 - Applying the formatFilteredData function.

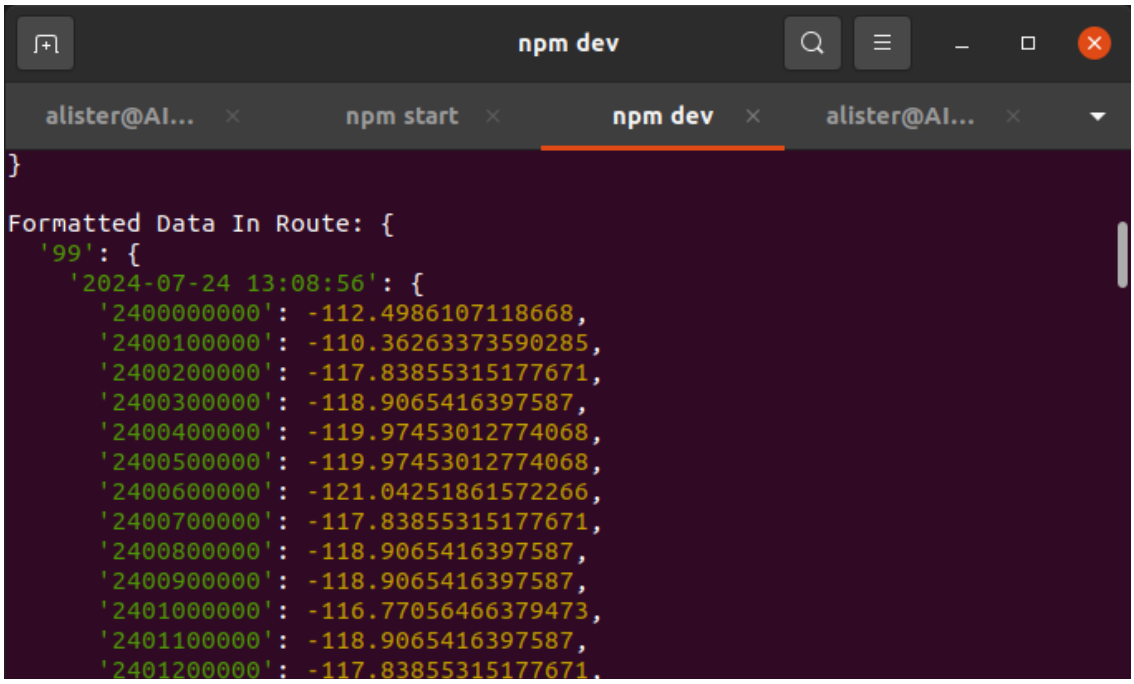


Figure 70 - Logging formatted filtered data.

Enabling the use of both static and relative time:

At the moment only relative time is being used, meaning the user can choose to track back from the current time. This would set the end date to the current time and time. However, I want to enable to choose their own start date as well as end date. The UI to do this already exists however I have not yet implemented it.

I am able to access both the static and relative time from the time picker UI using the useComplexTimeRange function. The useComplexTimeRange function is a custom hook that interacts with a TimeRangeContext to provide the current complex time range, a method to update the time range, an optional method to undo the last change. This custom hook abstracts the logic for managing complex time ranges, making it easier to use within React components.

The timeRange variable is an array containing two elements. If the user chooses to use relative time, then the first element of the array will contain the offset in milliseconds, with the second elements offset being 0. If the user chooses to use static time, then the first element will contain the start time in milliseconds and the second element will contain the end time in milliseconds. I implemented a simple if statement to check the type of timeRange and set the start date and end date.

```
// Converts static time to be passed onto formatDate function
if (timeRange[0].type === 'static') {
  const startDate = new Date(timeRange[0].time);
  const endDate = new Date((timeRange[1] as { time: number }).time);
  startTime = startDate;
  endTime = endDate;
}

// Converts relative time to be passed onto formatDate function
if (timeRange[0].type === 'relative') {
  const timeDifference = (timeRange[0].offset)*-1;
  const timeNow = new Date();
  startTime = new Date(timeNow.getTime() - timeDifference);
  endTime = timeNow;
}
```

Figure 71 - Finding start and end date using the timeRange variable.

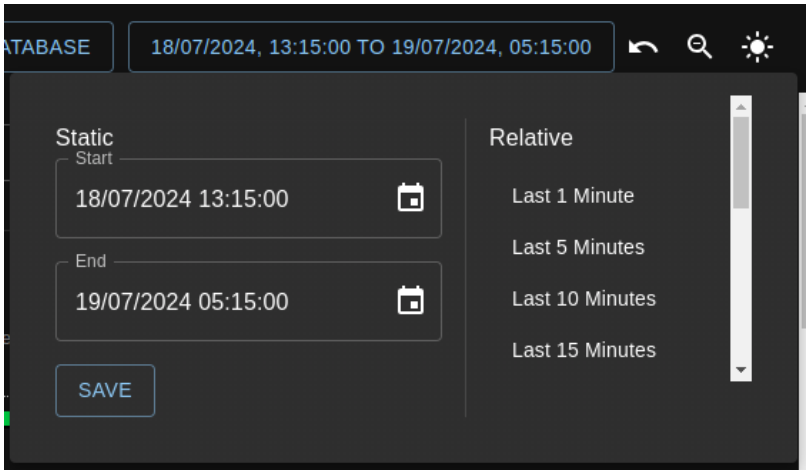


Figure 72 - Date/time picker UI on the dashboard.

Adding a react circular progress element to indicate data retrieval:

During the continuous testing phase of development, the absence of any indication that data retrieval was in progress became a recurring annoyance. Given the potentially large volume of SSA data, the retrieval process can take several seconds,

varying based on the options set. To address this issue, I decided to implement a loading circle to visually indicate when data retrieval is underway. This addition not only enhances user experience by providing feedback during data processing but also eliminates uncertainty about whether a query has successfully returned any SSA data.

I started by adding a useState variable called isSubmitting in the StreamOptsEditor function to track whether the form is currently being submitted or is waiting for a response. The useState was set to true within the handleSubmit method when the update button is clicked. If a response is received and it is successful, the useState is set to false. I imported the CircularProgress element from React and placed it next to the update button.

Figure 73 - Implementing the isSubmitting useState.

To prevent multiple clicks on the update button, which could trigger multiple queries, I used the isSubmitting useState to manage disabling and enabling the update button.

This approach ensures that while a submission is in progress, the update button is disabled, preventing additional submissions until the current one is complete.

Figure 74 - CircularProgress react element.

Using integer for time instead of string:

Initially, dates were stored as strings, representing the actual date values. However, this approach can lead to several issues, such as lack of type safety, parsing complexity, and sorting inefficiencies. To mitigate these problems, I decided to store time as milliseconds since the Unix epoch, keeping it as an integer. This method offers several advantages:

- **Improved Type Safety**: Using integers to represent time ensures type safety, reducing the risk of errors related to type mismatches and improving overall code robustness.
- **Simplified Parsing**: Integers are straightforward to work with and do not require complex parsing operations, unlike strings which need to be parsed into date objects for manipulation and comparisons.
- **Efficient Sorting**: Sorting integers is inherently more efficient than sorting date strings. This leads to better performance, especially when dealing with large datasets.

- **Consistency**: Storing time as integers provides a consistent format across different parts of the application, ensuring uniformity in data handling and processing.
- **Ease of Calculation**: Performing arithmetic operations on integers is simpler and faster, facilitating easy calculations for time differences, durations, and comparisons.

Overall, using integers to represent time enhances the application's performance, maintainability, and reliability.

To implement this change, I began by updating the SSAHistoryOpts type. I changed the startDate and endDate data types to number. In the GetSSAHistory.ts file I added a function called formatDate. This function converts milliseconds to a string date value that can be used to perform the SQL query. I use the formatDate function in the GetSSAHistory function to convert the date before querying.

```
// Function that formats milliseconds to date string for sql query
function formatDate(milliseconds: number): string {
  const date = new Date(milliseconds)
  const year = date.getFullYear();
  const month = String(date.getMonth() + 1).padStart(2, '0');
  const day = String(date.getDate()).padStart(2, '0');
  const hours = String(date.getHours()).padStart(2, '0');
  const minutes = String(date.getMinutes()).padStart(2, '0');
  const seconds = String(date.getSeconds()).padStart(2, '0');
  return `${year}-${month}-${day} ${hours}:${minutes}:${seconds}`;
}
```

Figure 75 - formatDate function used to convert milliseconds to a date string value.

In the StreamOptsEditor function within the SSAStreamHistory.tsx file I imported the SimplifyRange custom function from the code base. This function uses the timeRange variable to return the start date and end date since the epoch, whether relative time or static time is used. I can then use the JavaScript getTime method to convert the date into milliseconds. This has simplified the process of getting the start and end date. Finally, I updated the route so that it expects the startDate and endDate to be a number.

```
const simpleTimeRange = SimplifyRange(timeRange)

let deviceId = isFinite(values.deviceId as number)
    ? values.deviceId as number
    : undefined
post({
    deviceId: deviceId,
    streamId: values.streamId,
    startDate: simpleTimeRange[0].getTime(),
    endDate: simpleTimeRange[1].getTime(),
```

Figure 76 - Using the SimplifyRange function to extract start date and end date.

Implementing the HistoryCapture type:

After consulting with Callum, I had a much better Idea of how the waterfall graph is displayed. On the SSA live streaming feature the WaterfallDisplay function outputs a line of data which represents the captures within that second, then shifts the canvas up so there is a continuous flow of the waterfall chart. Knowing this I had a better understanding of how I could display the retrieved SSA data chronologically.

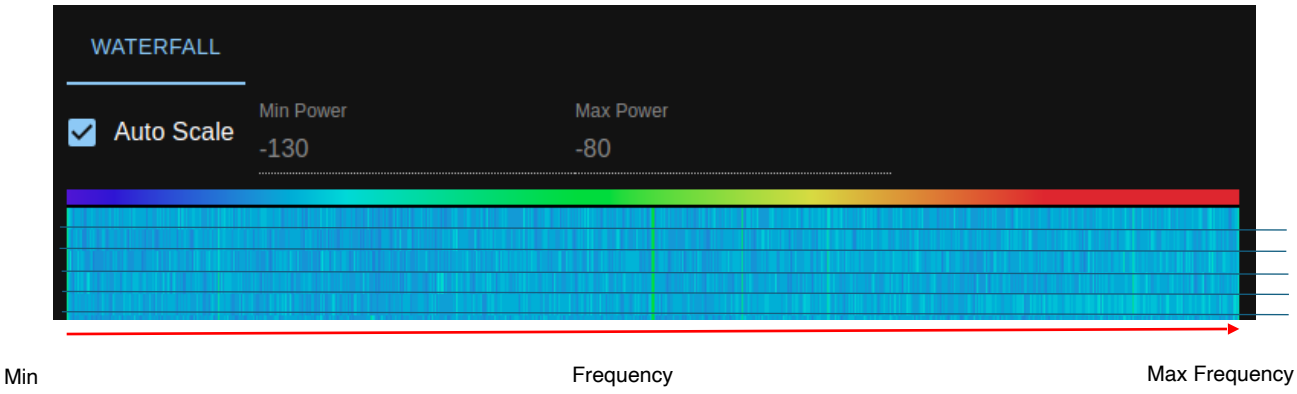


Figure 77 - Waterfall plot.

I started by defining a new type called HistoryCapture for the data sent to the front end, replacing the old HistoryData type. The HistoryCapture type maps each stream Id to an object containing an array of captures. Each capture includes a timestamp, and an array of power levels, where each power level consists of a frequency and a power value. With this data type it is much easier to manage, extract and display the SSA data since each capture represents a line on the waterfall plot.

```
type HistoryCapture = {
  [streamId: number]:{
    captures: {
      time:number,
      powerLevels:{frequency:number, power:number}[]
    }[]
  }
}
```

Figure 78 - HistoryCapure type.

I updated the formatFilteredData function in the GetSSAHistory.ts file, so that it would format the filtered data into the HistoryCapture type. It takes in the filtered data containing stream Id, time, frequency, and power. The function initialises an empty object, historyCaptures, to store the formatted data.

It then iterates over each data object in the filteredData array. For each data object, it checks if an entry for the given streamId exists in historyCaptures. If not, it creates a

new entry with an empty captures array. Next, it searches for a capture within the current stream ID's captures array that matches the time property of the data object. If such a capture is not found, it creates a new capture with the given time and an empty powerLevels array, and then adds this new capture to the captures array.

The function then adds the frequency and power from the data object to the powerLevels array of the corresponding capture. It also increments a counter, count, to keep track of the number of processed data points. Finally, it logs the total number of formatted data points and returns the formatted data.

```
// Function that formats the filtered data to fit the HistoryCapture type
function formatFilteredData(filteredData: {streamid: number, time: Date,
  frequency: number, power: number}[]): { [key: number]: { captures: Capture[] } } {
  let count = 0;
  const historyCaptures: { [key: number]: { captures: Capture[] } } = {};

  for (let i = 0; i < filteredData.length; i++) {
    const data = filteredData[i];

    // If streamId doesnt exist in historyCaptures, create it
    if (!historyCaptures[data.streamid]) {
      historyCaptures[data.streamid] = { captures: [] };
    }

    // Find the capture with the same time
    let capture = historyCaptures[data.streamid].captures.find
    (capture => capture.time === data.time.getTime());

    // If capture does not exist, create it and add it to captures
    if (!capture) {
      capture = { time: data.time.getTime(), powerLevels: [] };
      historyCaptures[data.streamid].captures.push(capture);
    }

    // Add powerLevel to the corresponding capture
    capture.powerLevels.push({frequency: data.frequency, power: data.power});
    count += 1;
  }
  console.log("Formatted Data Length:",count)
  return historyCaptures;
}
```

Figure 79 - Updated formatFilteredData function.

Displaying the SSA data on a waterfall plot:

Now that I have the HistoryCapture type, I can extract the SSA data easily for displaying the waterfall plot. I updated the WaterfallDisplay function in the SSAStreamHistory.tsx file by setting the dataset parameter to use the HistoryCapture type and restructuring how the waterfall is drawn.

```
function WaterfallDisplay({
    dataset,
    powerRange,
    onSizeChange,
    onMouseMove,
    ...canvasProps
}: {
    dataset: HistoryCapture,
    powerRange: [number,number],
    onSizeChange: (width: number)=>void,
    onMouseMove: (t: number | undefined /* 0-1 */)=>void,

    width?: number,
    height?: number,
    //Should be included in the HTMLCanvasElement
} & Omit<React.DetailedHTMLProps<
    React.HTMLAttributes<HTMLCanvasElement>, HTMLCanvasElement
>, "onMouseMove">) {
    const canvasRef = useRef<HTMLCanvasElement | null>(null)
```

Figure 80 - WaterfallDisplay function.

In the useEffect within the WaterfallDisplay function, the goal is to draw visual representations of the SSA data onto an HTML canvas element. First, it references the canvas element using useRef, and retrieves the 2D drawing context of the canvas. If the context is successfully obtained, it then extracts the width and height of the canvas.

For now, the function focuses on a specific stream within the dataset, identified by the stream Id 99. Before proceeding, it clears the canvas to ensure no previous drawings remain. If the canvas dimensions are invalid or the specified stream is undefined, the function exits early.

Next, the function calculates the height of each sample by dividing the canvas width by the number of captures in the current stream. It iterates over each capture in the stream, determining the vertical position for each sample based on its index. For each capture, it computes the minimum and maximum frequencies and calculates the width of each frequency sample by dividing the canvas width by the number of power levels in the capture.

For each power level within a capture, the function calculates the horizontal position on the canvas by mapping the frequency value to a pixel position within the canvas width. It then determines the colour corresponding to the power value using a helper function and fills a rectangle at the computed position with the appropriate colour.

This process is repeated for all captures and power levels, effectively rendering a visual representation of the dataset on the canvas.

```
//Draw a new line
const powerRangeRef = useLatest(powerRange);
useEffect(()=>{
    if(!canvasRef.current)
        return;

    const ctx = canvasRef.current.getContext("2d");

    if(ctx) {
        const {width, height} = ctx.canvas;
        const currentStream = dataset[99]

        //clears the screen
        ctx.clearRect(0, 0, width, height)

        // Invalid canvas
        if((!width || !height) || currentStream == undefined)
            return;

        const sampleHeight = Math.ceil(width / currentStream.captures.length);

        currentStream.captures.forEach((capture, i)=>{
            const y = i * sampleHeight
            const freqMin = capture.powerLevels[0].frequency
            const freqMax = capture.powerLevels[capture.powerLevels.length-1].frequency
            const sampleWidth = Math.ceil(width / capture.powerLevels.length)

            capture.powerLevels.forEach((e)=>{
                const x = Math.floor(MapValue(e.frequency, [freqMin,freqMax],
                    [0, canvasRef.current!.width]));
                ctx.fillStyle = getPowerColour(powerRangeRef.current, e.power)
                ctx.fillRect(x, y, sampleWidth, sampleHeight);
            })
        })
    }
}, [dataset, powerRangeRef])
```

Figure 81 - Code to draw the waterfall plot.

The function WaterfallPlot is used to display the selected frequency based on the user's cursor, the waterfall colour range which shows the frequency range and the waterfall plot itself. Since I had restructures the WaterfallDisplay there were some features which were now not working. There was a useEffect used to remove any extreme spikes from the live streaming SSA data, however this is not needed since it is vital to show the SSA data as is to keep data integrity. The selected frequency finder had some errors; to test the waterfall plot I commented out any insignificant bits of code in this function that caused errors.

I also had to disable the spectrum capture view tab which showed the graph since I no longer had the makeData function that is used to format the data for the SpectrumCaptureView function. After running some tests, I was glad to see that the waterfall plot was successfully displaying.

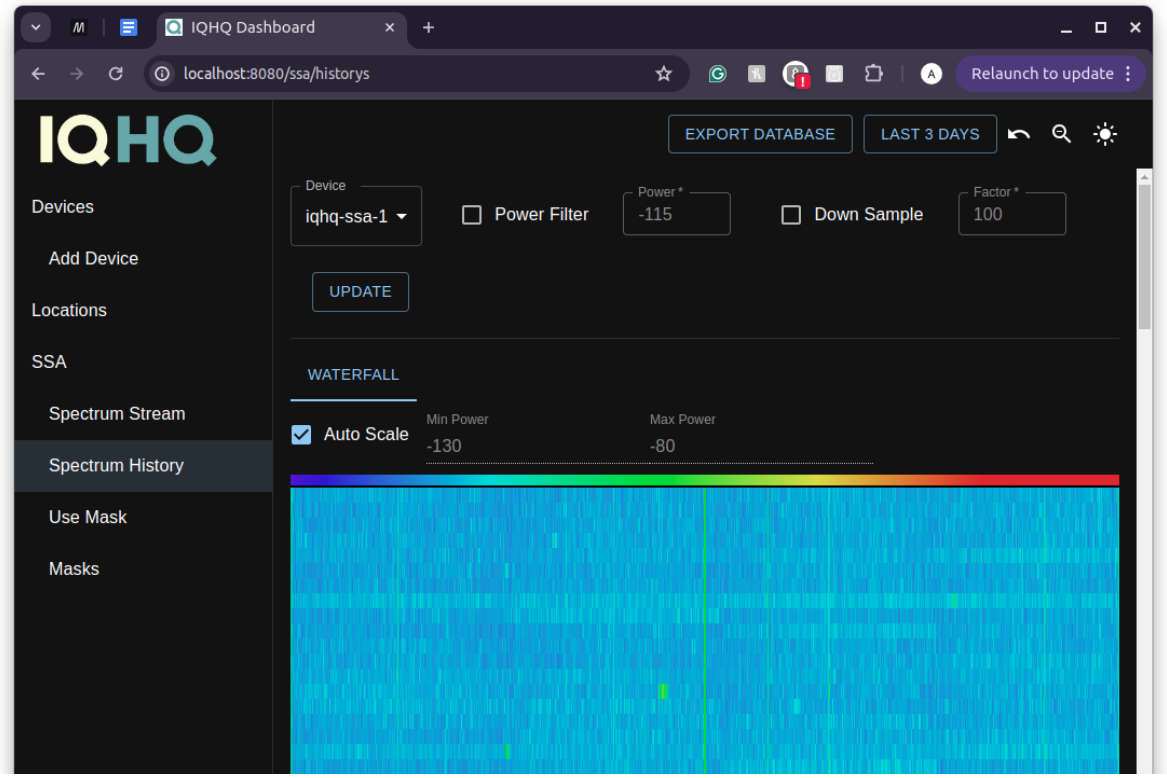


Figure 82 - Waterfall plot on the dashboard.

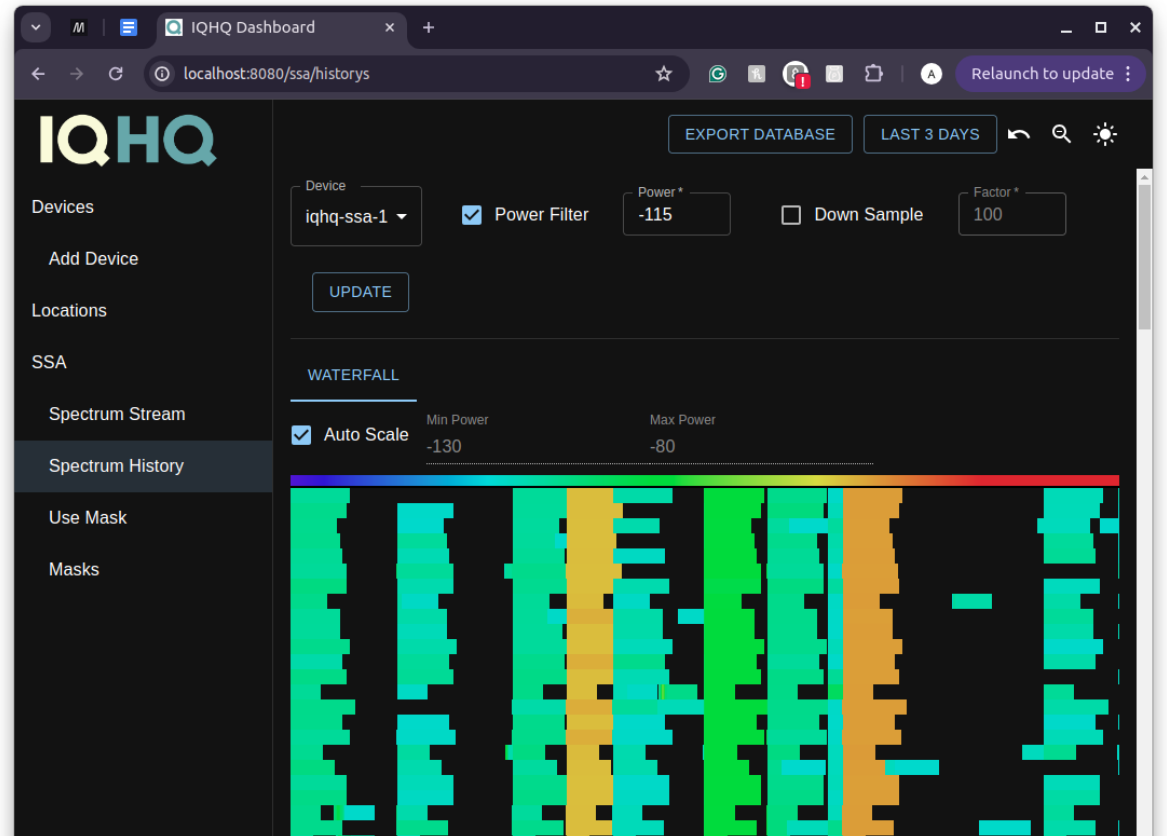


Figure 83 - Waterfall plot on the dashboard.

Implementing Alerts for Data Retrieval Feedback

To enhance user feedback on the dashboard, I added two types of React Alerts: an error alert and an info alert. The error alert notifies users when no data is found based on their provided options, while the info alert displays the number of individual power and frequency points retrieved. This implementation provides clear, immediate feedback on the data retrieval process, improving the user experience.

1. **Error Alert for No Data Found:**

   - **Purpose**: The error alert informs users when their query returns no data. This helps users quickly identify and rectify any issues with their search criteria.

   - **Implementation**: The alert is triggered when the data retrieval function returns an empty dataset. It provides a clear message to the user, indicating that no matching data was found for their specified options.

2. **Info Alert for Data Retrieval Summary:**

   - **Purpose**: The info alert gives users a summary of the data retrieved, including the length of the data and the number of individual power and frequency points. This helps users understand the scope of the data retrieved and ensures transparency in the data processing.

   - **Implementation**: This alert is displayed upon successful data retrieval and provides specific details about the dataset, such as the total number of data points. This information can be useful for users to verify the comprehensiveness of the data fetched.

I started by importing the alert element from React. In the SSAStreamHistory function, I created a new useState called length to track the length of the retrieved data. Within the onResponse callback, I added code to find the length of the filteredData object, which is then assigned to length.

To ensure accurate feedback, I added a conditional statement to check the value of length. If length is 0, a "No Data Available" alert is displayed. If length is greater than 0, an info alert is shown, displaying the length of the SSA data.

Initially, the "No Data Available" error was shown by default, even when the update button had not been clicked. To fix this, I set the default value of length to -1, ensuring that no alerts are displayed until the data retrieval process is triggered.

```
const [length, setLength] = useState<number>(-1);
```

Figure 84 - useState to track the length of the retrieved SSA data.

```
return (
    <div>
        <StreamOptsEditor
            defaultValue={defaultValues}
            onResponse={(filteredData: HistoryCapture) => {

                // Finding the length of the filteredData
                let length = 0;
                Object.values(filteredData).forEach(stream => {
                    stream.captures.forEach(capture => {
                        length += capture.powerLevels.length;
                    });
                });
                setLength(length)
                waterfallDataRef.current = filteredData;
                forceRender();
            }}
        />
        <br />
        <Grid item>
            {length == 0 &&
               <Alert variant="outlined" severity="error"
               style={{width:200, height:50, marginBottom: 20}}
               >No Data Available</Alert>}
            {length > 0 &&
               <Alert variant="outlined" severity="info"
               style={{width:200, height:50, marginBottom: 20}}
               >SSA Length: {length}</Alert>}
        </Grid>
```

Figure 85 - React alerts to display feedback from the data retrieval process.

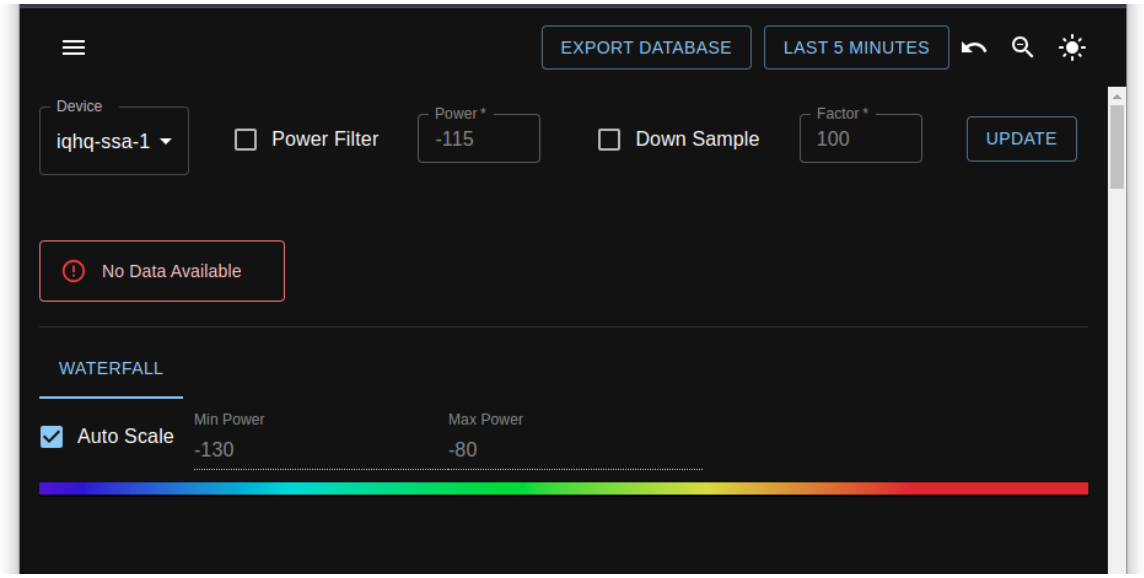


Figure 86 - No data available, error alert.

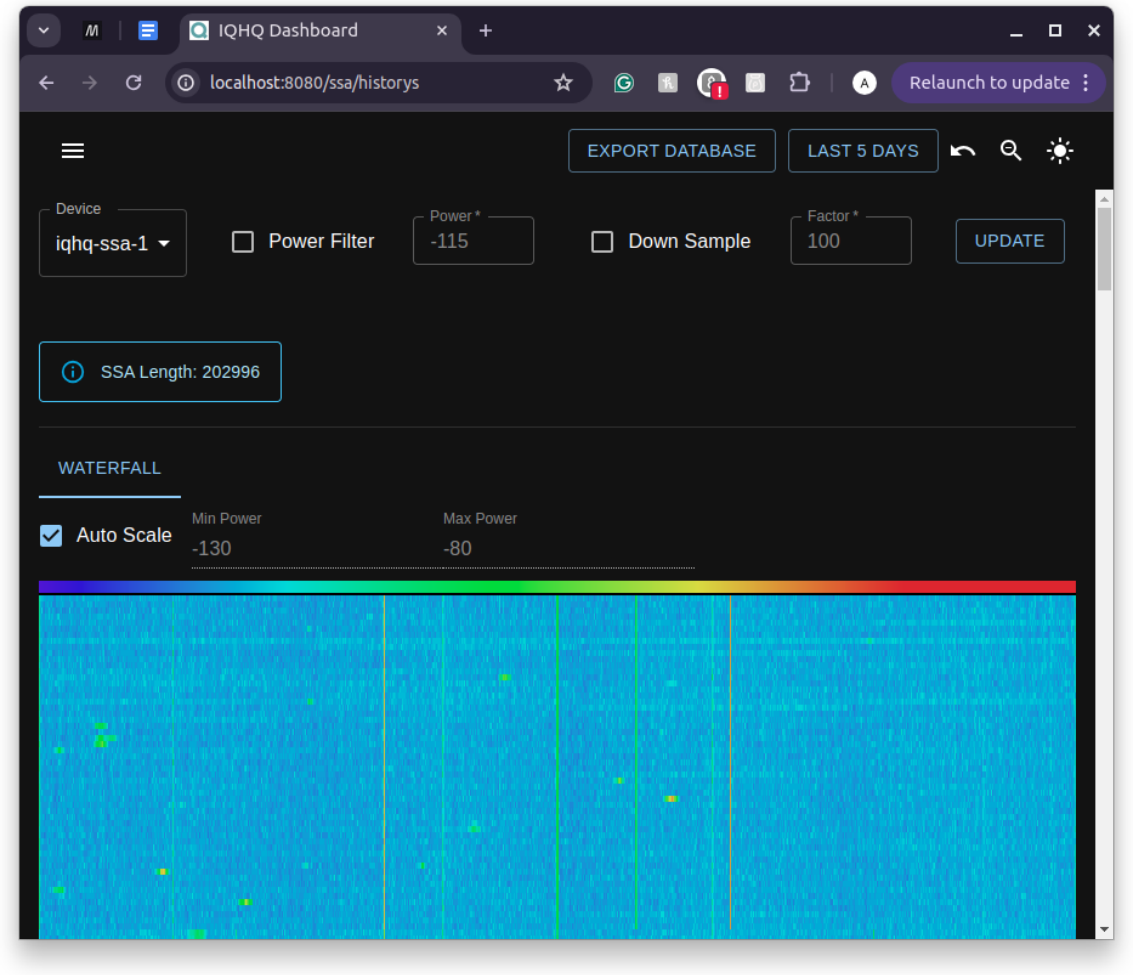


Figure 87 - SSA data length, info alert.

By implementing these alerts, the dashboard provides a more informative and responsive interface, helping users to better understand and interact with the data retrieval process.

Implementing the Frequency Power Graph

My next focus was to display the capture graph, which shows the power level at each frequency interval. This graph is crucial for identifying points of interest within a known frequency range. The graph is displayed using the spectrumCaptureView function, which utilises a dataset provided by the makeData function. I needed to modify makeData to use the HistoryCapture type and return the same dataset format expected by spectrumCaptureView.

Figure 88 - SpectrumCaptureView function.

In the new makeData function, I extracted all the powerLevels from all the captures in the queried stream from the HistoryCapture object. These powerLevels were then mapped to Chart.js data points and sorted by frequency. By ensuring the return format matches the original makeData function, the spectrumCaptureView function remains compatible.

SSAStreamHistory.tsx

ui > src > components > SSA > SSAStreamHistory.tsx > SpectrumCaptureView > chartRef

```
function makeData(data: HistoryCapture): ChartData<"line"> {
    const streams = Object.entries(data);

    const streamData = streams.map(([streamId, streamData]) => {
        // Extract all powerLevels from all captures in the current stream
        const powerLevels = streamData.captures.flatMap(capture => capture.powerLevels);

        // Map powerLevels to Chart.js data points and sort by frequency
        const chartData = powerLevels
            .map(({ frequency, power }) => ({ x: frequency, y: power }))
            .sort((a, b) => a.x - b.x);

        return { streamId, data: chartData };
    });

    return {
        datasets: streamData.map(({ streamId, data }) => ({
            data,
            label: streamId + "",
            borderColor: '#2460bf',
            tension: 0,
            // borderJoinStyle: "round"
        })),
    };
}
```

Figure 89 - makeData function.

In the SSAStreamHistory function, I implemented the chartDataRef useRef. I applied the makeData function on the filteredData from getSSAHistory and assigned the return value to chartDataRef.current. I enabled the Capture tab and passed chartDataRef.current to spectrumCaptureView to display the graph.

```
const chartDataRef = useRef<ChartData<"line">>({ datasets: [] });
```

Figure 90 - chartDataRef useRef hook.

```
chartDataRef.current = makeData(filteredData)
```

Figure 91 - Formatting the filtered data.

```
<Grid container style={{ marginTop: 10 }}>
    <Grid item>
        <Tabs
            value={activeTab}
            onChange={(_, val) => setActiveTab(val)}
            style={{ marginBottom: "10px" }}
        >
            <Tab label="Waterfall" />
            <Tab label="Capture" />
        </Tabs>
    </Grid>
</Grid>
{activeTab === 0 && <WaterfallPlot
    data={waterfallDataRef.current}
 />}
{activeTab === 1 && <SpectrumCaptureView
    data={chartDataRef.current}
/>}
```

Figure 92 - Enabling the capture tab.

Enabling display of the selected frequency:

Enabling display of the selected frequency involved finding the minimum and maximum frequency from the dataset, as well as determining the length of the data. Given the wide range of frequencies in the dataset, I implemented code to retrieve these values, which were then used to set the selectedFrequency_kHz variable in the WaterfallPlot function.

To find the length of the dataset, I added code that loops through the dataset, counting the total number of powerLevels, and assigning this count to the variable length. I created two variables, max and min, initialised to negative infinity and infinity, respectively. These variables track the highest and lowest frequency values by comparing each frequency value against max and min. I implemented a loop that iterates through each streamId in the data object, and for each powerLevel within each capture of each streamId, the current power is compared to max and min, updating these variables accordingly. After logging the results and comparing them against the database, I verified that length, minimum frequency, and maximum frequency were all calculated correctly.

SSAStreamHistory.tsx

ui > src > components > SSA > SSAStreamHistory.tsx > WaterfallPlot

```
function WaterfallPlot({data}: {data: HistoryCapture}) {
    // @ts-ignore
    // const dataset : {x: number, y: number}[] = data.datasets?

    // Finds the length of data
    let length = 0;
    Object.values(data).forEach(stream => {
        stream.captures.forEach(capture => {
            length += capture.powerLevels.length;
        });
    });

    //maximum and minimum power value from data
    let maxFreq = -Infinity;
    let minFreq = Infinity;

    // Iterate through each streamId in the data object
    for (const streamId in data) {
        data[streamId].captures.forEach(capture => {
        capture.powerLevels.forEach(powerLevel => {
            // Update max if current power is higher
            maxFreq = Math.max(maxFreq, powerLevel.frequency);
            // Update min if current power is lower
            minFreq = Math.min(minFreq, powerLevel.frequency);
        });
        });
    }
```

Figure 93 - Code to find the maximum and minimum frequency values from the dataset.

Next, I uncommented the code used to find the selectedFrequency_kHz and updated the minimum and maximum values. I also uncommented the typography element used to display the selected frequency on the UI. With these changes, the selected frequency was now displayed and updated with the user's cursor movement. The WaterfallPlot function now includes the logic to find the dataset length, track minimum and maximum frequencies, and calculate the selectedFrequency_kHz based on cursor position. The selected frequency is displayed using a typography element that updates dynamically with cursor movements, providing real-time feedback to the user.

```
    let selectedFrequency_kHz : number | undefined = undefined;
    if(length && mousePos !== undefined) {
        const minimum = minFreq;
        const maximum = maxFreq;

        selectedFrequency_kHz = ((maximum - minimum) * mousePos + minimum) / 1000;
    }
```

Figure 94 - Calculating the selected frequency based on cursor position.

```
339             <Stack direction="column" spacing={0}>
340                 {selectedFrequency_kHz !== undefined && <Typography>
341                     {HumanFrequency(selectedFrequency_kHz)}
342                 </Typography>}
343                 <WaterfallColourRange
344                     canvasWidth={canvasWidth}
345                     powerRange={range}
346                     style={{width: "100%"}}
347                 />
```

Figure 95 - Displaying the selected frequency on the UI.

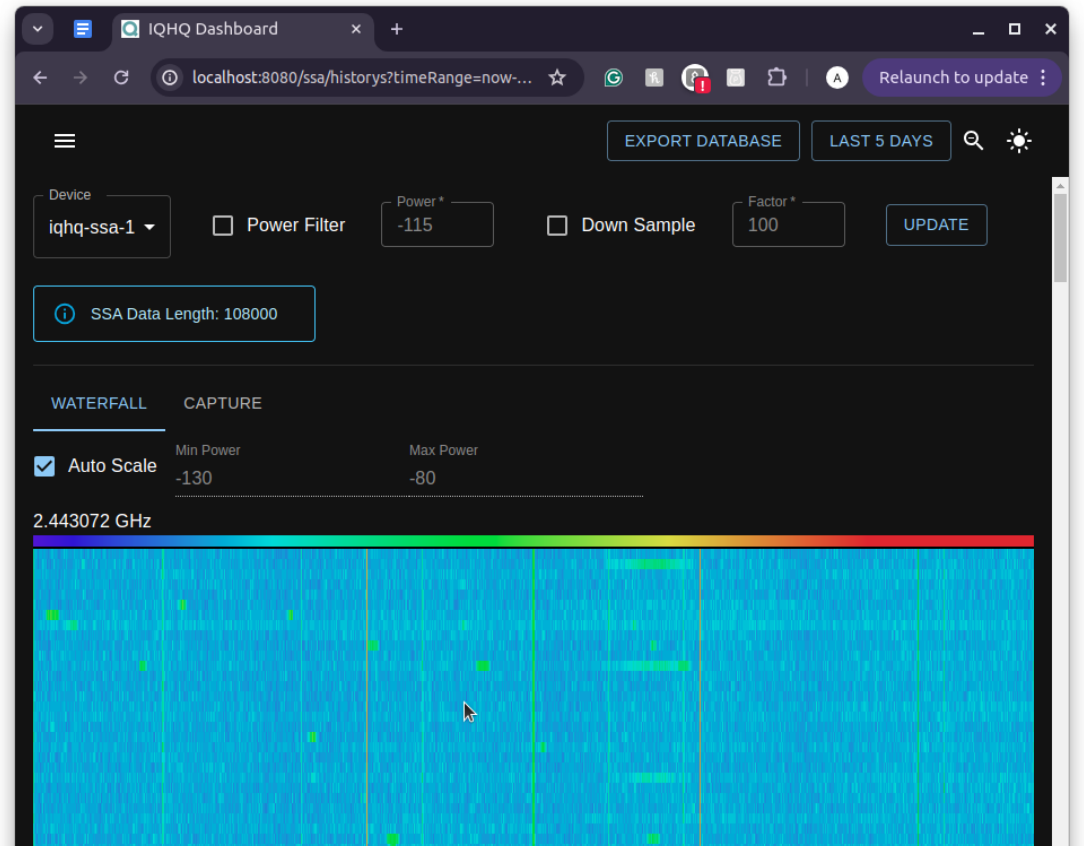


Figure 96 - Selected frequency based on cursor position being displayed.

Filtering SSA data by highest and lowest power levels on the capture graph.

Currently the capture graph displays all the powerLevel pairs from the dataset. This creates a clustered graph where each frequency has multiple power levels, creating a messy and unusable graph. Considering the usage of the graph I thought it would be useful for the user to be able to view the highest power level or lowest power level for each frequency. This feature would work in conjunction with the power filter option, the user can filter the SSA data based on a power level and then view the highest or lowest power level where there are multiple power levels for a frequency value.

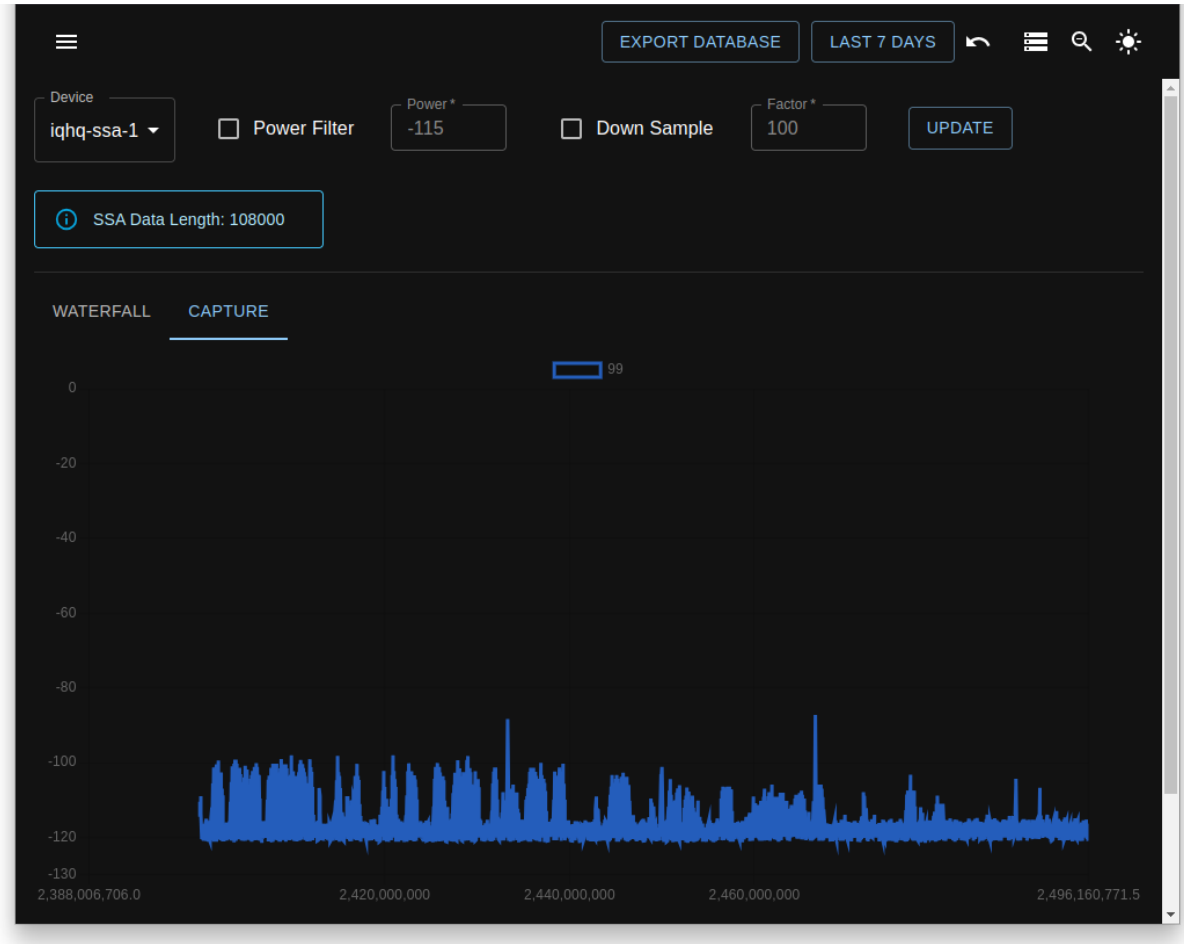


Figure 97 - Capture graph displaying all SSA data.

To implement this feature, I began by creating two new functions in the SSAStreamHistory.tsx file called highestPower and lowestPower. These functions are similar to the makeData function with only difference being that they either return the highest power levels or lowest power levels.

The highestPower function processes the HistoryCapture data to determine the highest power level for each frequency across all captures within each stream. It starts by converting the streams in the dataset into an array of entries. For each stream, it extracts all power levels from its captures and uses a map to track the highest power for each frequency. As it iterates over the power levels, it updates the map if a higher power value is found for a frequency. After processing, the map is converted back into an array of data points suitable for Chart.js, sorted by frequency. Finally, the function returns an object structured for Chart.js, containing datasets for each stream with corresponding highest power levels and appropriate styling.

```
34  // Function that returns the highest power level for each frequency
35  function highestPower(data: HistoryCapture): ChartData<"line"> {
36      const streams = Object.entries(data);
37
38      const streamData = streams.map(([streamId, streamData]) => {
39          // Extract all powerLevels from all captures in the current stream
40          const powerLevels = streamData.captures.flatMap(capture => capture.powerLevels);
41
42          // Use a Map to track the highest power for each frequency
43          const frequencyMap = new Map<number, number>();
44
45          powerLevels.forEach(({ frequency, power }) => {
46              if (!frequencyMap.has(frequency) || power > frequencyMap.get(frequency)!) {
47                  frequencyMap.set(frequency, power);
48              }
49          });
50
51          // Convert the map back to an array of Chart.js data points and sort by frequency
52          const chartData = Array.from(frequencyMap.entries())
53              .map(([frequency, power]) => ({ x: frequency, y: power }))
54              .sort((a, b) => a.x - b.x);
55
56          return { streamId, data: chartData };
57      });
58
59      return {
60          datasets: streamData.map(({ streamId, data }) => ({
61              data,
62              label: streamId + "",
63              borderColor: '#2460bf',
64              tension: 0,
65              // borderJoinStyle: "round"
66          })),
67      };
68  }
```

Figure 98 - Function that returns the highest power level for each unique frequency from a dataset.

Similarly, the lowestPower function works the same but returns a dataset for each stream with corresponding lowest power levels.

```
// Functions that returns the lowest power level for each frequency
function lowestPower(data: HistoryCapture): ChartData<"line"> {
    const streams = Object.entries(data);

    const streamData = streams.map(([streamId, streamData]) => {
        const powerLevels = streamData.captures.flatMap(capture => capture.powerLevels);

        const frequencyMap = new Map<number, number>();

        powerLevels.forEach(({ frequency, power }) => {
            if (!frequencyMap.has(frequency) || power < frequencyMap.get(frequency)!) {
                frequencyMap.set(frequency, power);
            }
        });

        const chartData = Array.from(frequencyMap.entries())
            .map(([frequency, power]) => ({ x: frequency, y: power }))
            .sort((a, b) => a.x - b.x);

        return { streamId, data: chartData };
    });

    return {
        datasets: streamData.map(({ streamId, data }) => ({
            data,
            label: streamId + "",
            borderColor: '#2460bf',
            tension: 0,
            // borderJoinStyle: "round"
        })),
    };
}
```

Figure 99 - Function that returns the lowest power level for each unique frequency from a dataset.

I decided to use react switches to allow the user to toggle on either the highest power filter or lowest power filter. To track the state of the switch I added two new useStates useHighestPower and useLowestPower. I then created a function called handleResponse to set the length of the data for the alert and to display the correct graph data based on the user's selection.

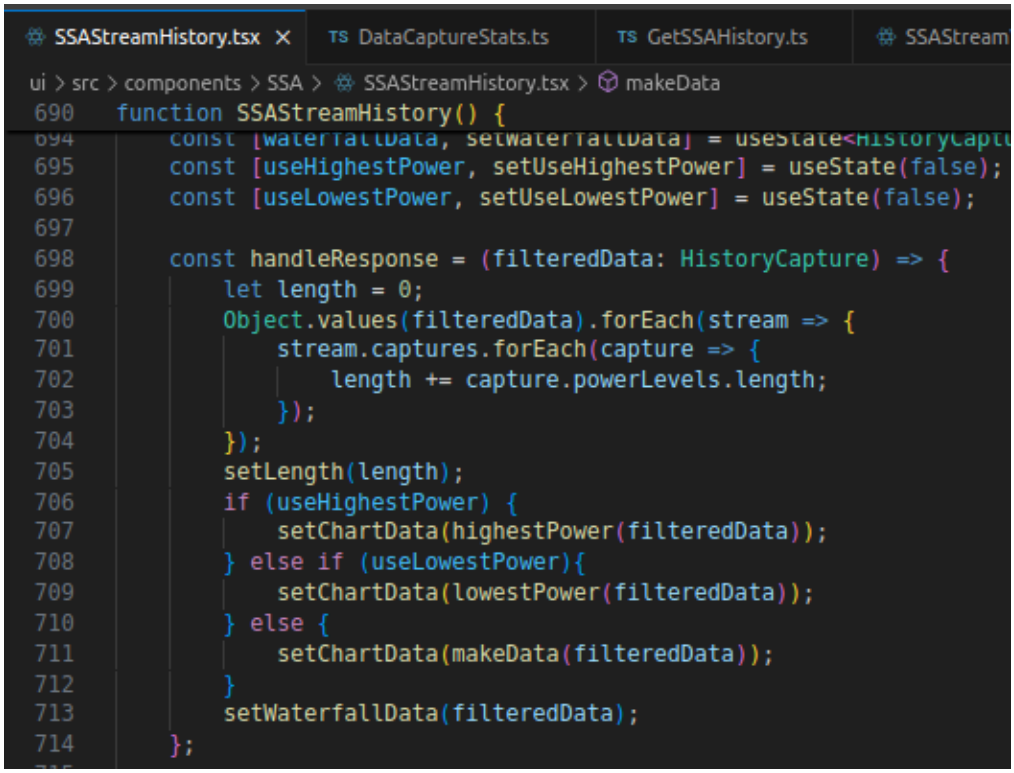


Figure 100 - Function that handles the retrieved data response.

Finally, I imported the switch element from react and created two toggles within the capture graph tab. One toggle was labelled ‘Highest Power Levels’ and the other ‘Lowest Power Levels.’ I ensured that only one switch could be enabled at a time. If no switches are enabled that all the powerLevels are shown using the makeData function.

SSAStreamHistory.tsx | DataCaptureStats.ts | GetSSAHistory.ts | SSAStreamView.tsx | Route.ts

ui > src > components > SSA > SSAStreamHistory.tsx > makeData

```
690   function SSAStreamHistory() {
749               <div style={{ display: activeTab === 1 ? 'block' : 'none' }}>
750                   <SpectrumCaptureView data={chartData} />
751                   <FormGroup style={{ margin: 20 }}>
752                       <FormControlLabel
753                           control={<Switch checked={useHighestPower} onChange={(e) => {
754                               setUseHighestPower(e.target.checked);
755                               if (e.target.checked) setUseLowestPower(false);
756                           }} />}
757                           label="Highest Power Levels"
758                       />
759                       <FormControlLabel
760                           control={<Switch checked={useLowestPower} onChange={(e) => {
761                               setUseLowestPower(e.target.checked);
762                               if (e.target.checked) setUseHighestPower(false);
763                           }} />}
764                           label="Lowest Power Levels"
765                       />
766                   </FormGroup>
767               </div>
768           </div>
769       );
770   }
```

Figure 101 - React switches to toggle highest and lowest power levels.

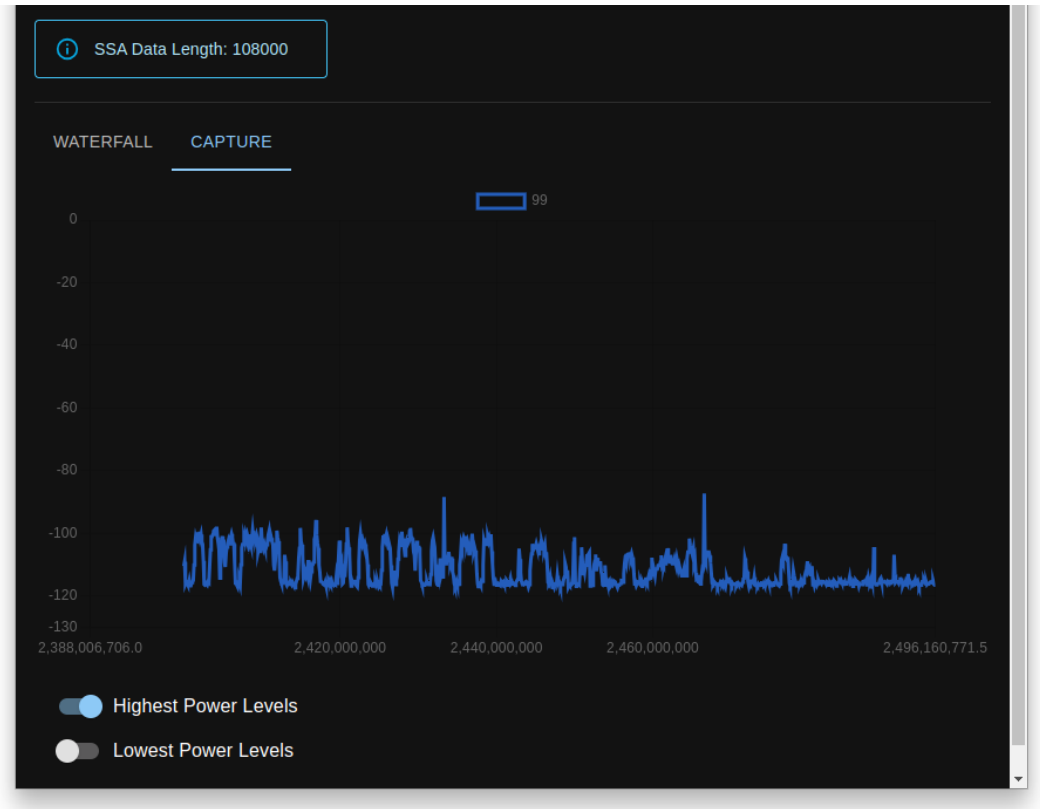


Figure 102 - Capture graph with the highest power level toggled.

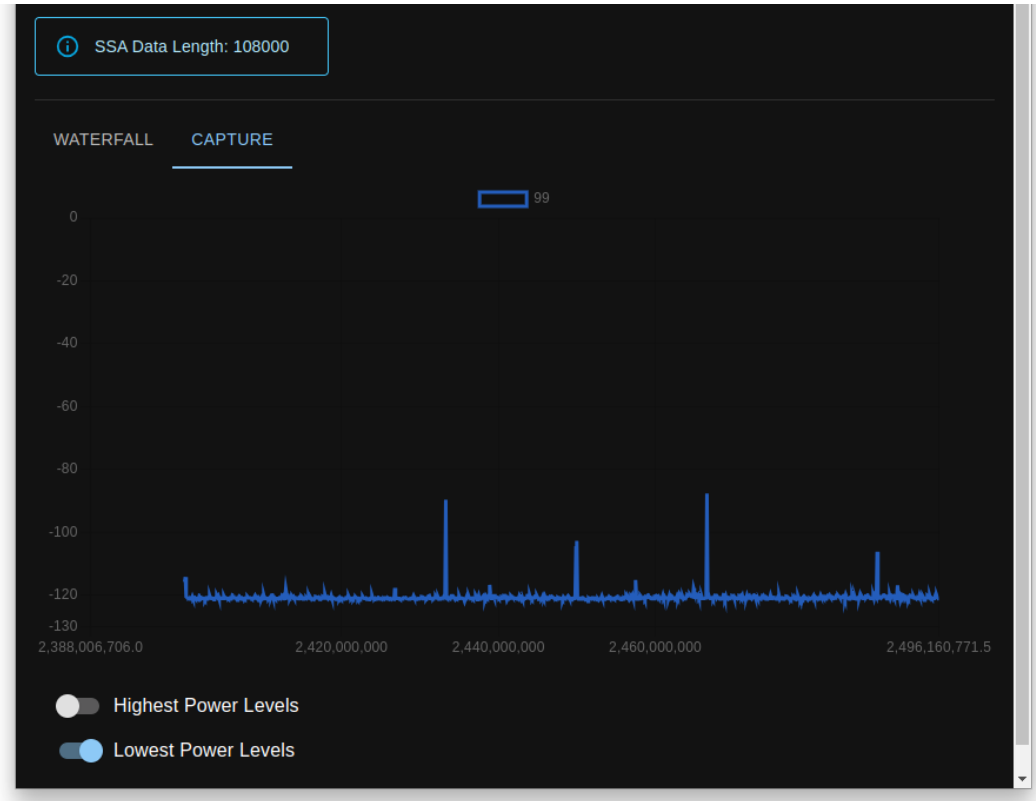


Figure 103 - Capture graph with the lowest power level toggled.

Retrieving all unique device Ids from the database:

In line with the requirements, I wanted to ensure that users were able to query the history SSA data using streamId. In the future intend to have multiple streams running at once from a single device, therefore it is crucial that they are able to isolate any one stream. I also realised a major flow with the current UI design. The device drop down uses the DeviceSelect element which only shows the device that the dashboard is currently connected too. I want to allow users to select any device and stream from the database, so they can view specific SSA data. In order to achieve this, I need to retrieve each unique device and stream from the database, and display these in a device dropdown and stream dropdown.

I began by creating a function called getDevices which was temporarily placed in the DataCaptureStats.ts file. This function queries the ssa_metrics table from the database to return a list of the devices from the database table.

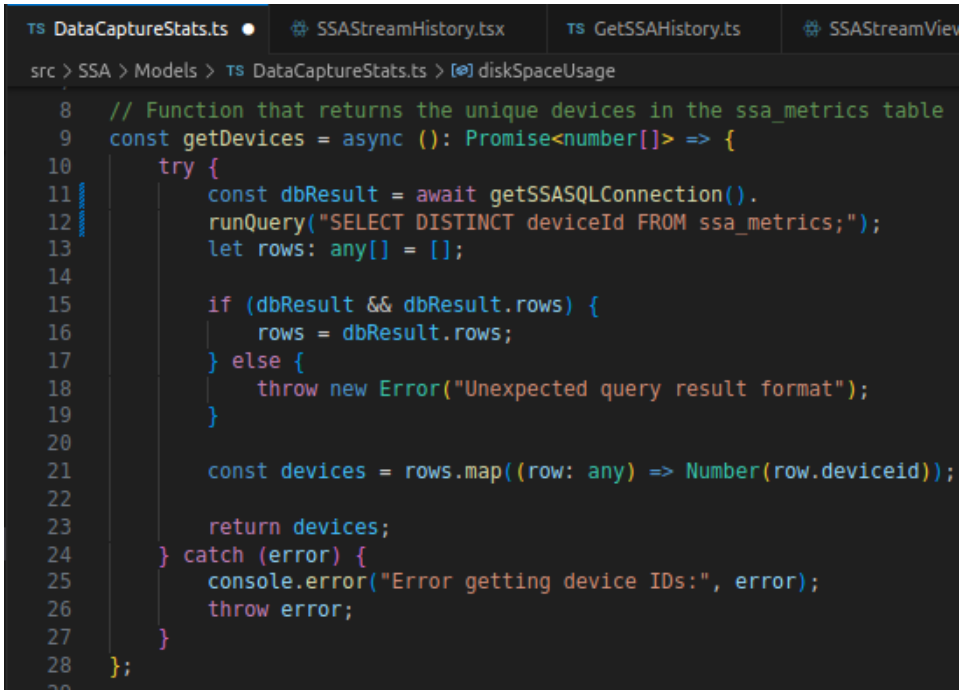


Figure 104 - Function that returns each unique devices from the ssa_metrics database table.

I then added a route GET using the route /devices. This route would return the devices from the getDevices function.

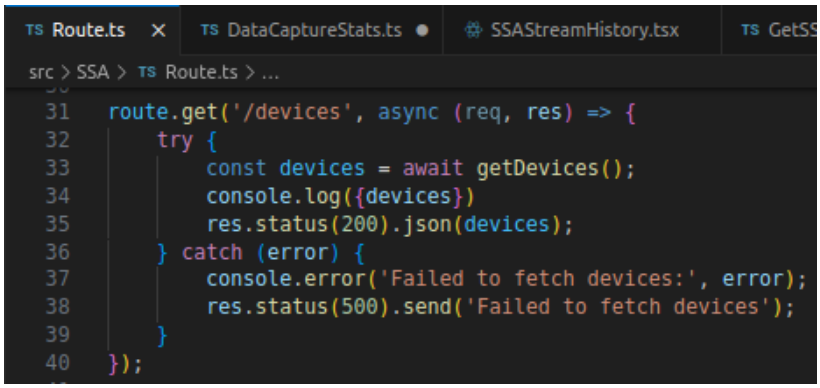


Figure 105 - devices GET route.

In order to test the function, I logged the result in the index.ts file which would execute tasks on startup. I added random devices to the database table to ensure that all unique devices were being retrieved.

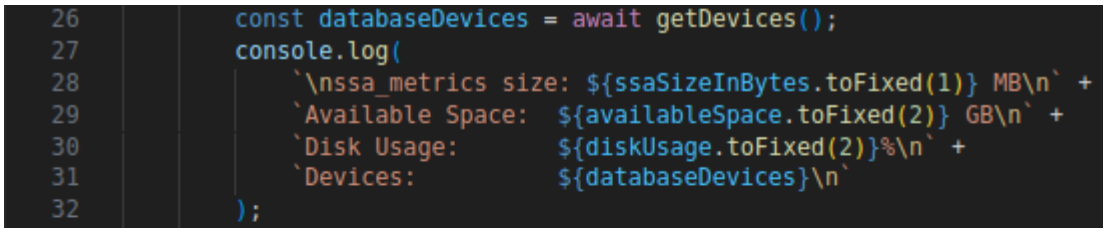


Figure 106 - Loggin the result of getDevices function.

Query Query History

```
1  INSERT INTO ssa_metrics
2  VALUES
3  (1111111111,11,'2024-07-31 11:06:58.687+00',-111,1111111111),
4  (2222222222,22,'2024-07-31 11:06:58.687+00',-222,2222222222),
5  (3333333333,33,'2024-07-31 11:06:58.687+00',-333,3333333333);
6
7  SELECT DISTINCT deviceId FROM ssa_metrics;
8
```

Data Output Messages Notifications

	deviceid bigint
1	3357947717
2	2222222222
3	3333333333
4	1111111111

Figure 107 - Inserting device Ids to test the getDevices function.

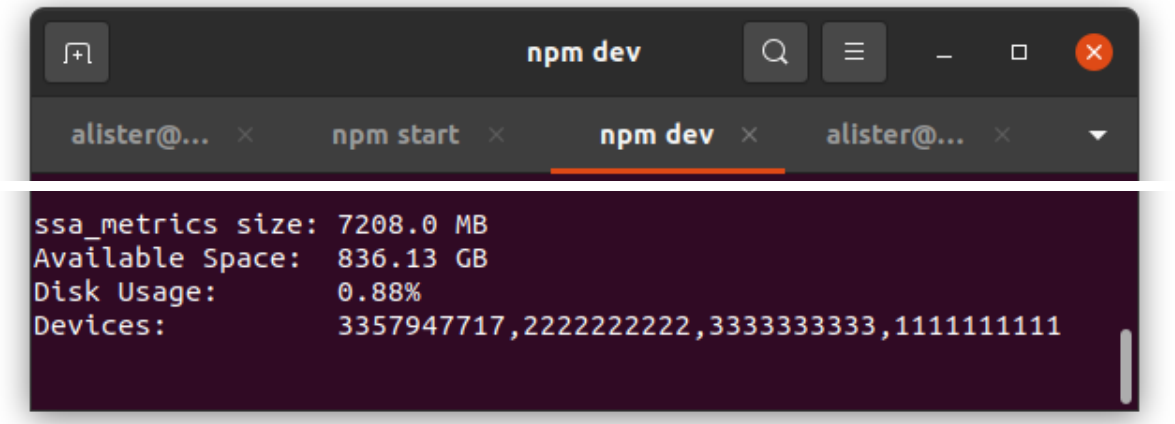


Figure 108 - Output of the getDevice function.

GetDeviceandStream function:

Now that I have a function to retrieve all the unique device IDs from the ssa_metrics table, I decided to enhance it so that the function would also retrieve unique device IDs and stream IDs within a specific time range. To achieve this, I started by creating a new file called GetDeviceandStream.ts. Creating a new file for this functionality was necessary to keep the code organised and modular.

I defined a type called deviceOptions that contains two arrays, one for devices and one for streams. This type ensures that the return value of the function is structured and clear.

```
type deviceOptions = {
    devices: number[],
    streams: number[]
}
```

Figure 109 - deviceOptions type.

I then added an asynchronous function called fetchDistinctValues which takes a column name, start time, and end time as parameters and returns an array of numbers. This function runs a database query to retrieve either the unique device IDs or stream IDs between the specified start and end times from the ssa_metrics table, based on the column name provided. To avoid errors due to mismatching column names, I converted the column name to lowercase. Additionally, I handled any potential errors that could arise from the query execution.

```
//Could add a time filter to show specific deviceId and streamId from a time range
async function fetchDistinctValues(columnName: string, startTime: number, endTime: number): Promise<number[]> {

    const oneHour = 3600000
    const startDate = new Date(startTime + oneHour)
    const endDate = new Date(endTime + oneHour)

    console.log("BEFORE new Date:",startTime,endTime)
    console.log("AFTER new Date:",startDate,endDate,"\n")
    console.log(`UTC Time: ${startDate.getUTCHours()}:${startDate.getUTCMinutes()}:${startDate.getUTCSeconds()}`);

    const queryResult: QueryResult = await getSSASQLConnection().runQuery(
        `SELECT DISTINCT ${columnName} FROM ssa_metrics WHERE "time" BETWEEN $1 AND $2;`,
        [startDate, endDate]
    );
    if (!queryResult || !queryResult.rows) {
        throw new Error("Unexpected query result format");
    }
    const lowerCaseColumnName = columnName.toLowerCase();
    return queryResult.rows.map((row: any) => Number(row[lowerCaseColumnName]));
}
```

Figure 110 - Function to retrieve all unique devices and streams from the database.

Next, I created another asynchronous function called GetDeviceandStream that returns the devices and streams using the deviceOptions type from the fetchDistinctValues function. This function uses the previously defined fetchDistinctValues function to get the necessary data. I then exported this function so that it could be accessed in the route. This setup ensures that the function can be easily utilised wherever needed in the application.

```
const GetDeviceandStream = async (
    startTime: number,
    endTime: number
): Promise<deviceOptions> => {
    try{
    const devices = await fetchDistinctValues("deviceId",startTime,endTime);
    const streams = await fetchDistinctValues("streamId",startTime,endTime);
    console.log({devices}, {streams})
    return { devices, streams };
    }
    catch (error) {
        console.error('Error fetching data from database:', error);
    }
}

export {
    GetDeviceandStream
}
```

Figure 111 - Function that returns the devices and streams as an array.

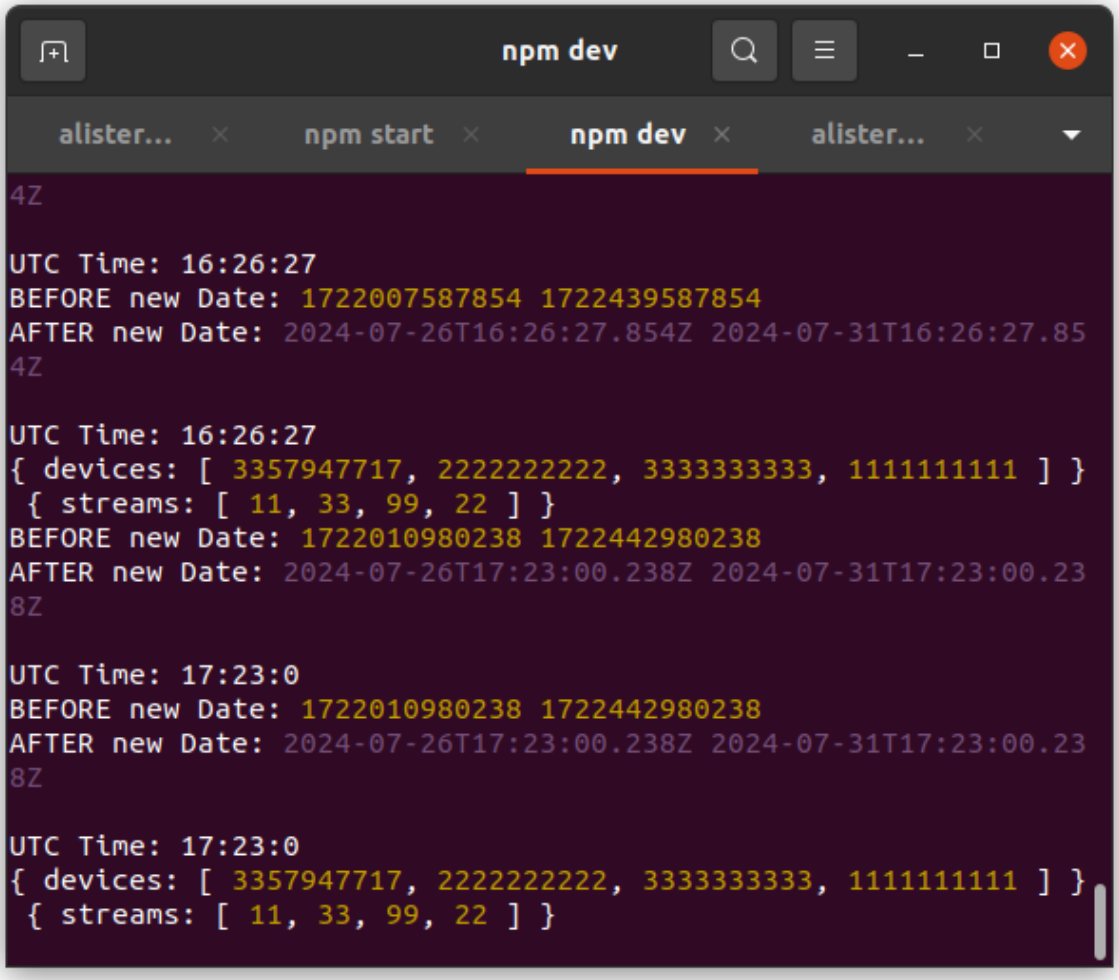


Figure 112 - Logging the result of the GetDeviceandStream function.

Displaying unique device Ids and stream Ids from the ssa_metrics database table:

To ensure the retrieved data could be filtered using stream Ids, I added stream Id to the database query in the GetSSAHistory function.

```
const queryData: QueryResult = await getSSASQLConnection().runQuery(
  `SELECT streamId, "time", frequency, power FROM ssa_metrics
  WHERE deviceId = $1 AND streamId = $2
  AND "time" BETWEEN $3 AND $4
  ORDER BY "time" ASC;`,
  [deviceId, streamId, startTime, endTime]
);
```

Figure 113 - Updated database query.

To allow the user to select devices and streams from the database within a selected time range, I needed to modify the SSA history UI. I began by updating the /devices route to a POST route that requires a start date and end date. The route returns the result of the GetDeviceandStream function if the HTTP status indicates success.

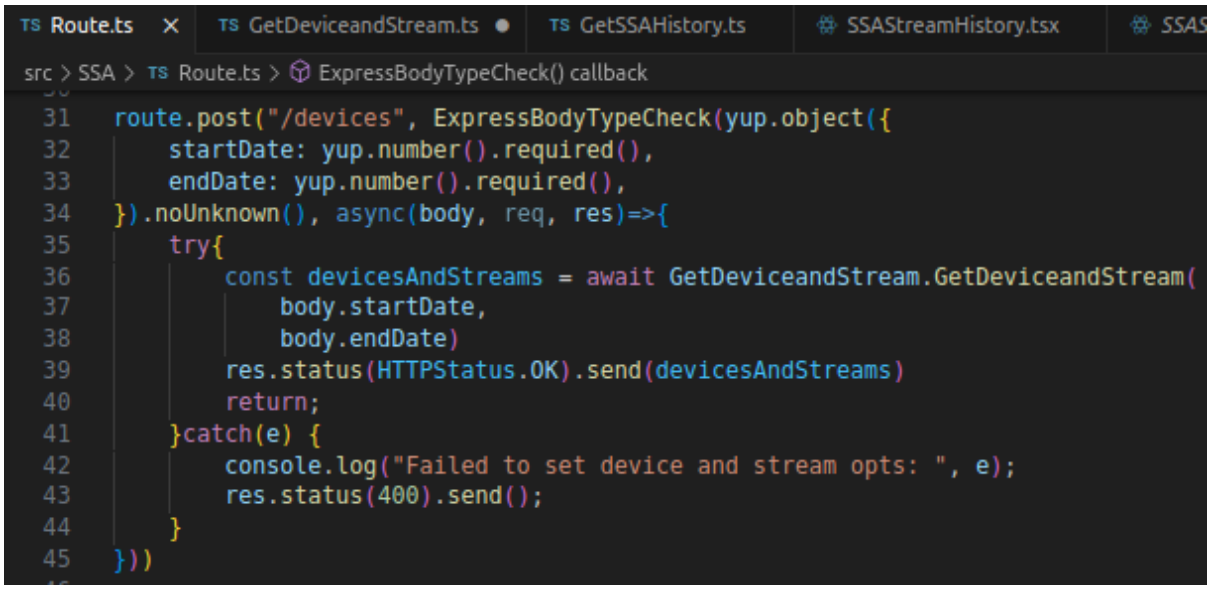


Figure 114 - Updates devices route.

In the StreamOptsEditor function, I added a useHTTPTrigger hook to retrieve the devices and streams. I created a useState called myResult to track the result from the route. When the update button is clicked, myResult is updated to retrieve the correct data based on the start and end time selected by the user.

```
const {
    response: devices,
    post: postTime,
} = useHTTPTrigger<deviceOptions, any>("/ssa/devices");
```

Figure 115 - useHTTPTrigger to access devices and streams list.

```
551     const [myResult, setMyResult] = useState<deviceOptions | undefined>(undefined);
552     const dbDevices = myResult?.devices
553     const dbStreams = myResult?.streams
554
555     const fetchData = async () => {
556         try {
557             console.log(simpleTimeRange[0].getTime(),simpleTimeRange[1].getTime())
558
559             const result = await postTime({
560                 startDate: simpleTimeRange[0].getTime(),
561                 endDate: simpleTimeRange[1].getTime(),
562             });
563             console.log({result})
564             if (isMounted()) {
565                 setMyResult(result);
566             }
567         } catch (error) {
568             console.error("Error fetching data:", error);
569         }
570     };
571
572     useEffect(() => {
573         fetchData();
574     }, []);
```

Figure 116 - Posting time and setting result.

I replaced the DeviceSelect custom element with a React MenuItem to create a drop-down menu for the devices. The devices from the route are mapped onto the MenuItem. I also created another MenuItem component for the stream Id drop down.

```
623                 <LabeledSelect
624                     style={{minWidth: "100px"}}
625                     label="Device"
626                     type="number"
627                     inputProps={{min: 520, max: 2000}}
628                     defaultValue={props.defaultValue.deviceId}
629                     {...register("deviceId", {valueAsNumber: true})}
630                 >
631                     {dbDevices && dbDevices.map(size => (
632                         <MenuItem value={size} key={size}>
633                             {size}
634                         </MenuItem>
635                     ))}
636                 </LabeledSelect>
637             </Grid>
638             <Grid item>
639                 <LabeledSelect
640                     style={{minWidth: "100px"}}
641                     label="Stream"
642                     type="number"
643                     inputProps={{min: 520, max: 2000}}
644                     defaultValue={props.defaultValue.streamId}
645                     {...register("streamId", {valueAsNumber: true})}
646                 >
647                     {dbStreams && dbStreams.map(size => (
648                         <MenuItem value={size} key={size}>
649                             {size}
650                         </MenuItem>
651                     ))}
652                 </LabeledSelect>
```

Figure 117 - React MenuItem to select device and stream.

After testing, both drop downs worked correctly, allowing the user to select a device Id and stream Id from within a specific time range from the database. This enhancement makes it easier to filter data.

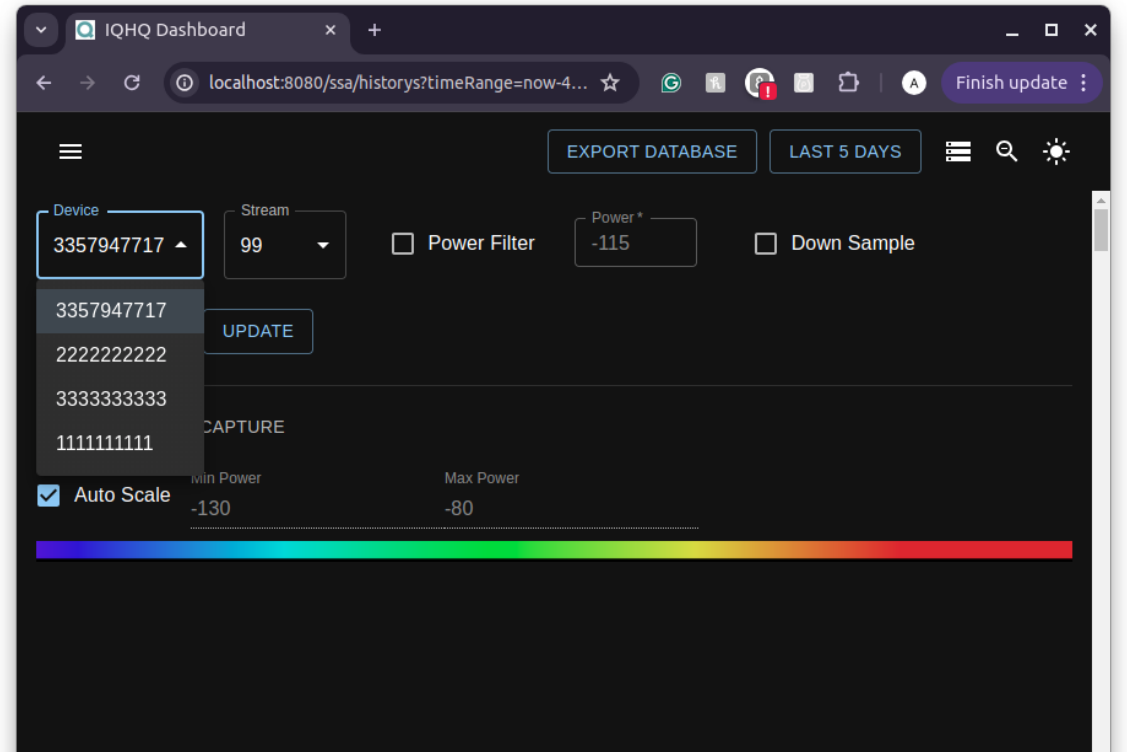


Figure 118 - Dashboard with updated device drop down.

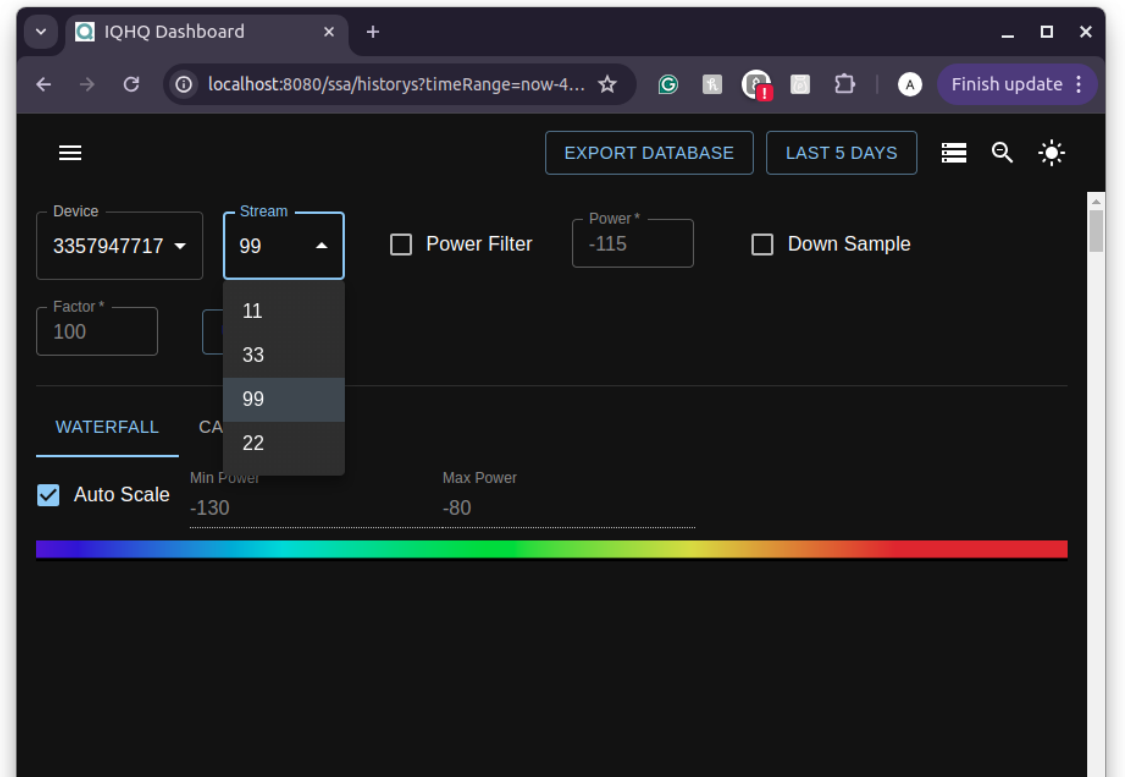


Figure 119 - Dashboard with stream drop down added.

Refetching Devices and Streams on Time Range Update:

Currently, the dashboard requires reloading each time a new time range is selected to access the corresponding devices and streams, which is inconvenient for users. To improve user experience, I implemented a refetch of the devices and streams each time the time range is updated.

To achieve this, I created a variable called offset to track any changes in relative time. Within my useEffect hook, I added a conditional statement to call fetchData if offset was not a null value. This ensures devices and streams are refetched each time the time value changes.

```
const offset = timeRange[0].type === "relative" ? timeRange[0].offset : null;
```

Figure 120 – Variable to track any change in time.

```
// Refetches device and str
useEffect(() => {
    if (offset !== null) {
        fetchData();
    }
}, [offset]);
```

Figure 121 - Refetching devices and streams when time value changes.

Displaying Alerts for No Devices Found:

I added an alert to notify users when no devices are found for the selected time range. The alert is displayed when the length of the device list is zero. This implementation enhances usability by dynamically updating the available devices and streams without requiring a page reload and providing feedback when no devices are available for the selected time range.

```
<Grid item>
    {dbDevices?.length === 0 &&
        <Alert variant="outlined" severity="error"
            style={{ width: 210, height: 50, marginBottom: 20 }}
        >No Devices Available</Alert>}
</Grid>
```

Figure 122 - React alert to notify users when no device is found.

Updating the HistoryCapture type:

During this project, I was tasked with updating my previous project, the SSA data capture process, to make it more efficient. Power and frequency were now being stored in arrays rather than individual values, which will make querying the database faster. Other advantages of this change include:

- **Reduced Redundancy**: Storing power and frequency in arrays eliminates the need for repeated entries, thus reducing redundancy in the database.
- **Improved Data Integrity**: Grouping related data together ensures that power and frequency values remain associated, enhancing data integrity.
- **Optimised Storage**: Using arrays optimises storage space by minimising the overhead associated with storing individual records.
- **Enhanced Query Performance**: Array-based storage allows for more efficient querying, as it reduces the number of database operations needed to retrieve related values.

To accommodate this change, I had to rewire the insertion of the SSA data into the database. This also necessitated updating the data retrieval process.

I began by updating the HistoryCapture type. I realised that streamId was not necessary to include in this type. This is because the user can only choose a single streamId on the dashboard; therefore, including streamId in the data that is sent back is redundant since it can only be the streamId that was chosen by the user. As a result, I removed the streamId property from HistoryCapture.

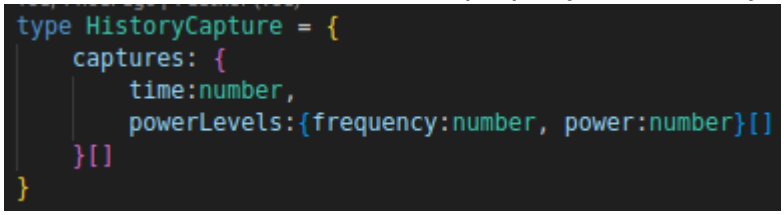


Figure 123 - Updated HistoryCapture type.

Updating the GetSSAHistory.ts file:

In the GetSSAHistory.ts file I updated the downSampling function, so that for each row in the input data, it initialises an empty array to store the downsampled power levels. It then iterates through the power levels of the row, selecting every factorth element and adding it to the downsampled power levels array. It then returns a new object containing the timestamp and the downsampled power levels. This process is repeated for all rows in the input data. Finally, the function returns an object with a single property, captures, which contains the array of downsampled data objects.

```
// Function that downsamples the data by a factor
function downSampling(data, factor: number): HistoryCapture {
  const downsampledData = data.map(row => {
    const downsampledPowerLevels = [];

    // Iterate through the power array with the given factor
    for (let i = 0; i < row.power.length; i += factor) {
      downsampledPowerLevels.push({
        frequency: row.frequency[i],
        power: row.power[i]
      });
    }

    return {
      time: new Date(row.time).getTime(),
      powerLevels: downsampledPowerLevels
    };
  });

  return {
    captures: downsampledData
  };
}
```

Figure 124 - Updated downSampling function.

Similarly, I updated the thresholdFiltering function so that for each row in the input data, it creates a new array of power levels where each element is an object containing the corresponding frequency and power from the row. It then filters this array to include only those power levels where the power is greater than or equal to the threshold value. After filtering, it returns a new object containing the timestamp and the filtered power levels for the row. This process is applied to all rows in the input data. Finally, the function returns an object with a single property, captures, which contains the array of filtered data objects.

```
// Function that filters data based on a power threshold
function thresholdFiltering(data, threshold: number): HistoryCapture {
  const filteredData = data.map(row => {
    const filteredPowerLevels = row.power
      .map((power, index) => ({
        frequency: row.frequency[index],
        power: power
      }))
      .filter(level => level.power >= threshold);

    return {
      time: new Date(row.time).getTime(),
      powerLevels: filteredPowerLevels
    };
  });

  return {
    captures: filteredData
  };
}
```

Figure 125 - Updated thresholdFiltering function.

I created a new function called formatData which transforms input data into the HistoryCapture format. It processes each row of the input data, creating a new array of objects where each object contains a timestamp and an array of power levels. For each row, it converts the time property to a timestamp in milliseconds and maps the power values to corresponding frequency-power pairs by iterating through the power values and their indices. The function returns an object with a single property, captures, which contains the array of formatted data objects.

```
// Function that formats the ssa data
function formatData(data): HistoryCapture {
  const formattedData = data.map(row => ({
    time: new Date(row.time).getTime(),
    powerLevels: row.power.map((power, index) => ({
      frequency: row.frequency[index],
      power: power
    }))
  }));

  return {
    captures: formattedData
  };
}
```

Figure 126 - Updated formatData function.

I then updated the applyFilters function so that it would apply any filter that was selected by the user. If no filters are chosen the retrieved data is formatted. The functions downSampling and thresholdFiltering already format the filtered data.

```
// Function that applies the filters and returns the filtered data
function applyFilters(data, powerFilter, downSampleFilter, powerValue, factor): HistoryCapture {
  let filteredData = data;
  if (powerFilter) {
    filteredData = thresholdFiltering(filteredData, powerValue);
  }
  if (downSampleFilter) {
    filteredData = downSampling(filteredData, factor);
  } else {
    filteredData = formatData(filteredData)
  }
  return filteredData;
}
```

Figure 127 - Updated applyFilters function.

I had to create a function to calculate the length of the raw SSA data called getLength. This function calculates the total number of power values in the filtered data. It initialises a counter, totalPowerValues, to zero. It then iterates through each row of the filteredData. For each row, it retrieves the power array and checks if it exists and is an array. If so, it increments the counter by the length of the power array. After processing all rows, the function returns the total count of power values.

```
// Function that returns the length of the ssa
function getLength(filteredData){
  let totalPowerValues = 0;

  filteredData.forEach(row => {
    const powerArray = row.power;
    if (powerArray && Array.isArray(powerArray)) {
      totalPowerValues += powerArray.length;
    }
  });
  return totalPowerValues
}
```

Figure 128 - Function that returns the length of raw SSA data.

I then created a function to calculate the length of the formatted SSA data called getLengthOf. This function calculates the total number of power levels in a HistoryCapture object. It initialises a counter, totalPowerValues, to zero. It then iterates over each capture in the historyCapture.captures array. For each capture, it adds the length of the powerLevels array to the counter. After processing all captures, the function returns the total count of power levels. I can now reuse this function in the front end to display to total number of power levels from the SSA data retrieval process.

```
function getLengthOf(historyCapture: HistoryCapture): number {
  let totalPowerValues = 0;

  // Iterate over each capture
  historyCapture.captures.forEach(capture => {
    // Add the length of the powerLevels array to the total
    totalPowerValues += capture.powerLevels.length;
  });

  return totalPowerValues;
}
```

Figure 129 - Function that returns the length of a HistoryCapture object.

Updating the SSAStreamHistory.tsx file:

To accommodate for the updated HistoryCapture type, I had to restructure the highestPower, lowestPower and makeData functions.

```
// Function that returns the highest power level for each frequency
function highestPower(data: HistoryCapture): ChartData<"line"> {
    // Assuming data is directly the captures array now
    const powerLevels = data.captures.flatMap(capture => capture.powerLevels);

    // Use a Map to track the highest power for each frequency
    const frequencyMap = new Map<number, number>();

    powerLevels.forEach(({ frequency, power }) => {
        if (!frequencyMap.has(frequency) || power > frequencyMap.get(frequency)!) {
            frequencyMap.set(frequency, power);
        }
    });

    // Convert the map back to an array of Chart.js data points and sort by frequency
    const chartData = Array.from(frequencyMap.entries())
        .map(([frequency, power]) => ({ x: frequency, y: power }))
        .sort((a, b) => a.x - b.x);

    // Since there's no streamId, use a placeholder of '99' for the label
    return {
        datasets: [{
            data: chartData,
            label: "99",
            borderColor: '#2460bf',
            tension: 0,
            // borderJoinStyle: "round"
        }],
    };
}
```

Figure 130 - Updated highestPower function.

```
function lowestPower(data: HistoryCapture): ChartData<"line"> {
    // Directly use the captures array from HistoryCaptures
    const powerLevels = data.captures.flatMap(capture => capture.powerLevels);

    // Use a Map to track the lowest power for each frequency
    const frequencyMap = new Map<number, number>();

    powerLevels.forEach(({ frequency, power }) => {
        if (!frequencyMap.has(frequency) || power < frequencyMap.get(frequency)!) {
            frequencyMap.set(frequency, power);
        }
    });

    // Convert the map back to an array of Chart.js data points and sort by frequency
    const chartData = Array.from(frequencyMap.entries())
        .map(([frequency, power]) => ({ x: frequency, y: power }))
        .sort((a, b) => a.x - b.x);

    // Use a placeholder of '99' for the label since there's no streamId
    return {
        datasets: [{
            data: chartData,
            label: "99",
            borderColor: '#2460bf',
            tension: 0,
            // borderJoinStyle: "round"
        }],
    };
}
```

Figure 131 - Updated lowestPower function.

```
function makeData(data: HistoryCapture): ChartData<"line"> {

        // Extract all powerLevels from all captures in the current streeam
    const powerLevels = data.captures.flatMap(capture => capture.powerLevels);

    // Map powerLevels to Chart.js data points and sort by frequency
    const chartData = powerLevels
        .map(({ frequency, power }) => ({ x: frequency, y: power }))
        .sort((a, b) => a.x - b.x);

    return {
        datasets: [{
            data: chartData,
            label: "99",
            borderColor: '#2460bf',
            tension: 0,
            // borderJoinStyle: "round"
        }],
    };
}
```

Figure 132 - Updated makeData function.

In the WaterFallDisplay function, I was accessing the power levels from within a dataset by calling on the streamId, however since the dataset does not contain a streamId, I can just call on the dataset itself.

```
if(ctx) {
    const {width, height} = ctx.canvas;

    //clears the screen
    ctx.clearRect(0, 0, width, height)

    // Invalid canvas
    if((!width || !height) || dataset === undefined)
        return;

    const sampleHeight = Math.ceil(width / dataset.captures.length);

    dataset.captures.forEach((capture, i)=>{
        const y = i * sampleHeight
        const sampleWidth = Math.ceil(width / capture.powerLevels.length)
```

Figure 133 - Updated WaterFallDisplay function.

I reused the getLengthOf function to find the length of the SSA data, so that it can be used to display the selected frequency in the WaterfallPlot function. I also used the function so that the length of the SSA data can be displayed to the user on the UI.

```
function WaterfallPlot({data}: {data: HistoryCapture}) {

    // Finds the length of data
    const length = getLengthOf(data)
```

Figure 134 - Reusing getLengthOf function.

```
const handleResponse = (filteredData: HistoryCapture) => {
    const length = getLengthOf(filteredData)
    setLength(length);
```

Figure 135 - Reusing getLengthOf function.

Finalising SSA history capture feature:

With all the requirements for the SSA history capture feature now implemented, my focus shifted to refining the codebase to enhance both readability and maintainability. This final stage involved addressing minor bugs, ensuring proper error handling, and optimising the code for clarity.

Fixing getting ssa_metrics table error:

One critical issue I encountered was related to the database table creation process. Each time the ssa_metrics table was dropped, and the dashboard application was restarted, an error would appear stating, "Error getting ssa_metric size: error: relation 'ssa_metrics' does not exist." This error indicated that the table was not being created as expected. Upon investigation, I discovered that the problem was due to the order of operations in the index.ts file. The createTableIfNotExists function, responsible for creating necessary tables, was being executed after database information logging, which caused the table creation to be overlooked. By reordering the operations to ensure that table creation occurred before any logging, I resolved this issue and ensured that the ssa_metrics table was correctly created on startup.

```
try{
    await MasksModel.createTableIfNotExists();
    await FusionReportModel.createTableIfNotExists();
    await SSAMetricsModel.createTableIfNotExists();
    const ssaSizeInBytes = await getSSASize();
    const availableSpace = await getDiskSpace();
    const diskUsage = await diskSpaceUsage();
    console.log(
        `\nssa_metrics size: ${ssaSizeInBytes.toFixed(1)} MB\n` +
        `Available Space:  ${availableSpace.toFixed(2)} GB\n` +
        `Disk Usage:       ${diskUsage.toFixed(2)}%\n`
    );
}catch(e) {
    console.error("Error making SSA Tables")
}
```

Figure 136 - Reordered the creation of tables and logging the database information.

Fixing waterfall data useState:

Another issue arose with the useState hook used to manage waterfall data. The state was initially set to expect a HistoryCapture object, but when data retrieval was incomplete or failed, it led to an error and caused the dashboard to display a blank screen. To address this, I introduced a defaultWaterfall variable of the HistoryCapture type, initialised with an empty captures list. By setting this variable as the default state, I prevented errors related to mismatched data structures and improved the robustness of the dashboard.

```
const defaultWaterfall: HistoryCapture = {
    captures:[]
};
```

Figure 137 - Default waterfall data.

```
const [waterfallData, setWaterfallData] = useState<HistoryCapture>(defaultWaterfall);
```

Figure 138 - Setting default waterfall data.

Displaying waterfall plot chronologically:

I also identified and corrected a problem with data chronology in the waterfall plot. The plot was displaying data out of order because the database query was not sorting the results correctly. I resolved this by updating the query to order results in descending chronological order, which corrected the display issue and ensured that data was presented accurately.

```
const queryData: QueryResult = await getSSASQLConnection().runQuery(
  `SELECT streamId, "time", frequency, power FROM ssa_metrics
  WHERE deviceId = $1 AND streamId = $2
  AND "time" BETWEEN $3 AND $4
  ORDER BY "time" DESC;`,
  [deviceId, streamId, startTime, endTime]
);
```

Figure 139 - Updating database query.

Fixing maximum update depth exceeded error:

During some brief testing I ran into an issue where the chrome page would crash after displaying the waterfall plot. I inspected the console to see the following warning was displayed multiple times: “SSAStreamHistory.tsx:531 Warning: Maximum update depth exceeded. This can happen when a component calls setState inside useEffect, but useEffect either does not have a dependency array, or one of the dependencies changes on every render.” This issue was caused by an infinite loop in the React component. This happens when a component continually re-renders because of repetitive updates triggered by state changes or effects.

In my case, the problem arose because the useEffect hook was set to execute whenever the offset value changed. However, the function responsible for fetching data, fetchData, was recreated on every render, which meant that useEffect always saw a new version of fetchData. This constant change in fetchData led to an infinite loop as useEffect was repeatedly triggered, causing the "Maximum update depth exceeded" error.

To fix the issue, I introduced a way to track the previous value of offset using a React ref. This ref allowed us to compare the current offset value with the previous one. With this comparison, I ensured that certain actions, like logging or fetching data,

only occurred when offset actually changed. Additionally, I used useCallback to ensure that fetchData had a stable reference across renders, preventing unnecessary updates that could trigger re-renders. This approach effectively stopped the infinite loop by controlling when useEffect.

```
// Ref to keep track of the previous offset value
const prevOffsetRef = useRef<null | number>(null);

// Fetch data function
const fetchData = useCallback(async () => {
    try {
        const result = await postTime({
            startDate: simpleTimeRange[0].getTime(),
            endDate: simpleTimeRange[1].getTime(),
        });
        if (isMounted()) {
            setMyResult(result);
        }
    } catch (error) {
        console.error("Error fetching data:", error);
    }
}, [postTime, simpleTimeRange]);

// Effect to handle offset change
useEffect(() => {
    if (offset !== null && prevOffsetRef.current !== offset) {
        console.log("Offset changed to:", offset);
        prevOffsetRef.current = offset;
        fetchData();
    }
}, [offset, fetchData]);
```

Figure 140 - Updated refetch of devices and streams from the database.

Tidying up code:

In addition to fixing bugs and optimising functionality, I devoted time to tidying up the code. This involved removing redundant code, simplifying complex functions, and improving overall code structure. Code readability is crucial as it makes the codebase easier to understand and maintain, facilitating quicker debugging and future enhancements. I also added comprehensive comments throughout the code, making it easier for future developers (or myself) to grasp the logic and intent behind the code. This practice not only aids in debugging but also ensures that the code remains accessible and comprehensible for anyone who might work on it later.

In summary, these final refinements and enhancements were essential in ensuring that the SSA history capture feature was not only functional but also maintainable and easy to understand. This final stage of development was crucial in delivering a reliable and efficient feature that meets user needs while maintaining high standards of code quality.

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.

GetSSAHistory.ts:

downSampling():

Test Case 1: Valid Down-sampling Factor

Input:

```
const data = {
  captures: [
    { time: 1620000000000, powerLevels: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] },
    { time: 1620000001000, powerLevels: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] }
  ]
};
const factor = 2;
```

Expected Output:

```
{
  captures: [
    { time: 1620000000000, powerLevels: [1, 3, 5, 7, 9] },
    { time: 1620000001000, powerLevels: [10, 8, 6, 4, 2] }
  ]
}
```

Description: Test the function with a valid down-sampling factor to ensure it correctly down-samples the powerLevels array by taking every 2nd element.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 2: Down-sampling Factor of 1 (No Down-sampling)

Input:

```
const data = {
  captures: [
    { time: 1620000000000, powerLevels: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] },
    { time: 1620000001000, powerLevels: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] }
  ]
};
const factor = 1;
```

Expected Output:

```
{
  captures: [
    { time: 1620000000000, powerLevels: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] },
    { time: 1620000001000, powerLevels: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] }
  ]
}
```

Description: Test the function with a down-sampling factor of 1 to ensure it returns the original powerLevels array without any down-sampling.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 3: Down-sampling Factor Greater than Array Length

Input:

```
const data = {
  captures: [
    { time: 1620000000000, powerLevels: [1, 2, 3] },
    { time: 1620000001000, powerLevels: [10, 9, 8] }
  ]
};
const factor = 5;
```

Expected Output:

```
{
  captures: [
    { time: 1620000000000, powerLevels: [1] },
    { time: 1620000001000, powerLevels: [10] }
  ]
}
```

Description: Test the function with a down-sampling factor greater than the length of the powerLevels array to ensure it returns only the first element of each array.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 4: Empty powerLevels Array

Input:

```
const data = {
  captures: [
    { time: 1620000000000, powerLevels: [] },
    { time: 1620000001000, powerLevels: [] }
  ]
};
const factor = 2;
```

Expected Output:

```
{
  captures: [
    { time: 1620000000000, powerLevels: [] },
    { time: 1620000001000, powerLevels: [] }
  ]
}
```

Description: Test the function with an empty powerLevels array to ensure it handles the empty array correctly without errors.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

thresholdFiltering():

Test Case 1: Valid Threshold Filtering

Input:

```
const data = {
  captures: [
    { time: 1620000000000, powerLevels: [{ power: 5 }, { power: 15 }, { power: 25 }] }
    { time: 1620000001000, powerLevels: [{ power: 10 }, { power: 20 }, { power: 30 }]
  ]
};
const threshold = 15;
```

Expected Output:

```
{
  captures: [
    { time: 1620000000000, powerLevels: [{ power: 15 }, { power: 25 }] },
    { time: 1620000001000, powerLevels: [{ power: 20 }, { power: 30 }] }
  ]
}
```

Description: Test the function with a valid threshold value to ensure it correctly filters the powerLevels array by retaining only those elements with power greater than or equal to the threshold.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 2: Threshold Higher than All Power Levels

Input:

```
const data = {
  captures: [
    { time: 1620000000000, powerLevels: [{ power: 5 }, { power: 10 }, { power: 15 }] }
    { time: 1620000001000, powerLevels: [{ power: 10 }, { power: 20 }, { power: 30 }]
  ]
};
const threshold = 35;
```

Expected Output:

```
{
  captures: [
    { time: 1620000000000, powerLevels: [] },
    { time: 1620000001000, powerLevels: [] }
  ]
}
```

Description: Test the function with a threshold value higher than all the power levels in the array to ensure it returns an empty array for each capture.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 3: Threshold Lower than All Power Levels

Input:

```
const data = {
  captures: [
    { time: 1620000000000, powerLevels: [{ power: 5 }, { power: 10 }, { power: 15 }] },
    { time: 1620000001000, powerLevels: [{ power: 10 }, { power: 20 }, { power: 30 }]
  ]
};
const threshold = 1;
```

Expected Output:

```
{
  captures: [
    { time: 1620000000000, powerLevels: [{ power: 5 }, { power: 10 }, { power: 15 }] },
    { time: 1620000001000, powerLevels: [{ power: 10 }, { power: 20 }, { power: 30 }]
  ]
}
```

Description: Test the function with a threshold value lower than all the power levels in the array to ensure it returns the original powerLevels array for each capture.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 4: Empty powerLevels Array

Input:

```
const data = {
  captures: [
    { time: 1620000000000, powerLevels: [] },
    { time: 1620000001000, powerLevels: [] }
  ]
};
const threshold = 10;
```

Expected Output:

```
{
  captures: [
    { time: 1620000000000, powerLevels: [] },
    { time: 1620000001000, powerLevels: [] }
  ]
}
```

Description: Test the function with an empty powerLevels array to ensure it handles the empty array correctly without errors.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

formatData():

Test Case 1: Valid Data Formatting

Input:

```
const data = [
  { time: "2023-07-15T12:00:00Z", power: [10, 20], frequency: [50, 60] },
  { time: "2023-07-15T13:00:00Z", power: [30, 40], frequency: [70, 80] }
];
```

Expected Output:

```
{
  captures: [
    {
      time: 1689422400000,
      powerLevels: [
        { frequency: 50, power: 10 },
        { frequency: 60, power: 20 }
      ]
    },
    {
      time: 1689426000000,
      powerLevels: [
        { frequency: 70, power: 30 },
        { frequency: 80, power: 40 }
      ]
    }
  ]
}
```

Description: Test the function with valid data to ensure it correctly formats the time and pairs the power and frequency values.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 2: Empty Data Array

Input:

```
const data = [];
```

Expected Output:

```
{
  captures: []
}
```

Description: Test the function with an empty data array to ensure it handles the empty array correctly without errors.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 3: Data with Inconsistent Array Lengths

Input:

```
const data = [
  { time: "2023-07-15T12:00:00Z", power: [10, 20, 30], frequency: [50, 60] }
];
```

Expected Output:

```
{
  captures: [
    {
      time: 1689422400000,
      powerLevels: [
        { frequency: 50, power: 10 },
        { frequency: 60, power: 20 }
      ]
    }
  ]
}
```

Description: Test the function with data where the lengths of the power and frequency arrays are not equal to ensure it only pairs values up to the shortest array length.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 4: Data with Invalid Time Format

Input:

```
const data = [
  { time: "invalid-time-format", power: [10, 20], frequency: [50, 60] }
];
```

Expected Output:

```
{
  captures: [
    {
      time: NaN,
      powerLevels: [
        { frequency: 50, power: 10 },
        { frequency: 60, power: 20 }
      ]
    }
  ]
}
```

Description: Test the function with an invalid time format to ensure it handles the invalid time and returns NaN for the time value.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

getLength():

Test Case 1: Valid SSA Data

Input:

```
const ssaData = [
  { power: [10, 20, 30] },
  { power: [40, 50] },
  { power: [60] }
];
```

Expected Output: 6

Description: Test the function with valid SSA data to ensure it correctly sums the lengths of all power arrays.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 2: Empty SSA Data Array

Input:

```
const ssaData = [];
```

Expected Output: 0

Description: Test the function with an empty SSA data array to ensure it handles the empty array correctly and returns 0.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 3: SSA Data with Missing Power Arrays

Input:

```
const ssaData = [
  { power: [10, 20] },
  { time: "2023-07-15T12:00:00Z" },
  { power: [30] }
];
```

Expected Output: 3

Description: Test the function with SSA data where some entries do not have power arrays to ensure it correctly skips those entries.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 4: SSA Data with Non-Array Power Fields

Input:

```
const ssaData = [
  { power: [10, 20] },
  { power: 30 },
  { power: [40, 50] }
];
```

Expected Output: 4

Description: Test the function with SSA data where some power fields are not arrays to ensure it only counts the lengths of valid power arrays.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

getLengthOf():

Test Case 1: Valid HistoryCapture Data

Input:

```
const data: HistoryCapture = {
  captures: [
    { time: 1622505600000, powerLevels: [{ frequency: 1, power: 10 }, { frequency: 2,
    { time: 1622509200000, powerLevels: [{ frequency: 1, power: 30 }] }
  ]
};
```

Expected Output: 3

Description: Test the function with valid HistoryCapture data to ensure it correctly sums the lengths of all power levels arrays.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 2: Empty HistoryCapture Object

Input:

```
const data: HistoryCapture = { captures: [] };
```

Expected Output: 0

Description: Test the function with an empty HistoryCapture object to ensure it handles the empty captures array correctly and returns 0.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 3: HistoryCapture with Captures Missing Power Levels

Input:

```
const data: HistoryCapture = {
  captures: [
    { time: 1622505600000, powerLevels: [{ frequency: 1, power: 10 }] },
    { time: 1622509200000, powerLevels: [] }
  ]
};
```

Expected Output: 1

Description: Test the function with HistoryCapture data where some captures have empty power levels arrays to ensure it correctly counts the lengths of the power levels arrays.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

Test Case 4: HistoryCapture with Mixed Power Levels Lengths

Input:

```
const data: HistoryCapture = {
  captures: [
    { time: 1622505600000, powerLevels: [{ frequency: 1, power: 10 }, { frequency: 2,
    { time: 1622509200000, powerLevels: [{ frequency: 1, power: 30 }, { frequency: 2,
  ]
};
```

```
power: 20 }] },
power: 40 }, { frequency: 3, power: 50 }] }
```

Expected Output: 5

Description: Test the function with HistoryCapture data containing captures with varying lengths of power levels arrays to ensure it sums all lengths correctly.

Actual Output: The actual output was the same as the expected output.

Evaluation: This test case has passed.

GetSSAHistory():

Test Case 1: Valid Data Retrieval and Filtering

Input:

```
deviceId: 1,
streamId: 1,
startDate: 1622505600000,
endDate: 1622592000000, /
powerFilter: true,
downSampleFilter: true,
power: 10,
factor: 2
```

Expected Output: Filtered HistoryCapture object with data matching the specified criteria

Description: Test the function with valid inputs to ensure it retrieves the data, applies filters, and returns the filtered HistoryCapture object correctly.

Actual Output: Filtered HistoryCapture object that matches the specified criteria.

Evaluation: This test case has passed.

Test Case 2: No Filters Applied

Input:

```
deviceId: 1,
streamId: 1,
startDate: 1622505600000,
endDate: 1622592000000,
powerFilter: false,
downSampleFilter: false,
power: 0,
factor: 0
```

Expected Output: Unfiltered HistoryCapture object with raw data matching the specified criteria

Description: Test the function with no filters applied to ensure it retrieves the raw data and returns it correctly without any modifications.

Actual Output: Raw HistoryCapture object with no filtering applied.

Evaluation: This test case has passed.

Test Case 3: Invalid Device or Stream ID

Input:

```
deviceId: -1,
streamId: -1,
startDate: 1622505600000,
endDate: 1622592000000,
powerFilter: false,
downSampleFilter: false,
power: 0,
factor: 0
```

Expected Output: Empty HistoryCapture object (no data found)

Description: Test the function with invalid device and stream IDs to ensure it handles cases where no data is found correctly and returns an empty HistoryCapture object.

Actual Output: Empty HistoryCapture object with no captures.

Evaluation: This test case has passed.

Test Case 4: Error Handling

Input:

```
deviceId: 1,
streamId: 1,
startDate: NaN, // inval
endDate: NaN, // invalid
powerFilter: false,
downSampleFilter: false,
power: 0,
factor: 0
```

Expected Output: Error message logged; function returns undefined

Description: Test the function with invalid date inputs to ensure it handles errors gracefully, logs the appropriate error message, and returns undefined.

Actual Output: Error message indicating failure to fetch data from database.

Evaluation: This test case has passed. Test Cases

GetDeviceandStream.ts:

fetchDistinctValues():

Test Case 1: Valid Column Name and Time Range

Input:

- columnName: "deviceId"
- startTime: 1672531200000 (January 1, 2023)
- endTime: 1672617600000 (January 2, 2023)

Expected Output: Array of unique device IDs from the specified time range.

Actual Output: Array of distinct device IDs, e.g., [101, 102, 103].

Description: This test case verifies that the function can correctly fetch distinct values from a valid column within the specified time range.

Evaluation: This test case has passed as the function returned the expected array of unique device IDs.

Test Case 2: Column Name with No Data in Time Range

Input:

- columnName: "streamId"
- startTime: 1672531200000 (January 1, 2023)
- endTime: 1672617600000 (January 2, 2023)

Expected Output: Empty array, as no stream IDs are present within the specified time range.

Actual Output: [] (Empty array).

Description: This test case checks how the function handles cases where the column has no entries within the given time range.

Evaluation: This test case has passed as the function correctly returned an empty array when no data was available.

Test Case 3: Invalid Column Name

Input:

- columnName: "invalidColumn"
- startTime: 1672531200000 (January 1, 2023)
- endTime: 1672617600000 (January 2, 2023)

Expected Output: Error indicating an unexpected query result format.

Actual Output: Error message: "Unexpected query result format".

Description: This test case ensures that the function handles invalid column names by throwing an appropriate error.

Evaluation: This test case has passed as the function correctly threw an error for an invalid column name.

GetDevicesandStreams():

Test Case 1: Valid Time Range with Existing Devices and Streams

Input:

- startTime: 1672531200000 (January 1, 2023)
- endTime: 1672617600000 (January 2, 2023)

Expected Output:

- devices: [101, 102, 103]
- streams: [201, 202, 203]

Actual Output:

- devices: [101, 102, 103]
- streams: [201, 202, 203]

Description: This test case verifies that the function can correctly fetch and return distinct device IDs and stream IDs for a valid time range where data is present.

Evaluation: This test case has passed as the function correctly returned the expected arrays of device IDs and stream IDs.

Test Case 2: Time Range with No Data

Input:

- startTime: 1672531200000 (January 1, 2023)
- endTime: 1672617600000 (January 2, 2023)

Expected Output:

- devices: []
- streams: []

Actual Output:

- devices: []
- streams: []

Description: This test case checks how the function handles a time range where no device IDs or stream IDs are available.

Evaluation: This test case has passed as the function correctly returned empty arrays for both devices and streams.

Test Case 4: Error Handling

Input:

- startTime: 1672531200000 (January 1, 2023)

- endTime: 1672617600000 (January 2, 2023)

Expected Output: Error message logged, and the function handles it gracefully.

Actual Output: Error message logged: "Error fetching data from database: <error details>", and function handles the error without crashing.

Description: This test case ensures that the function can handle errors gracefully, such as when there is an issue with the database query.

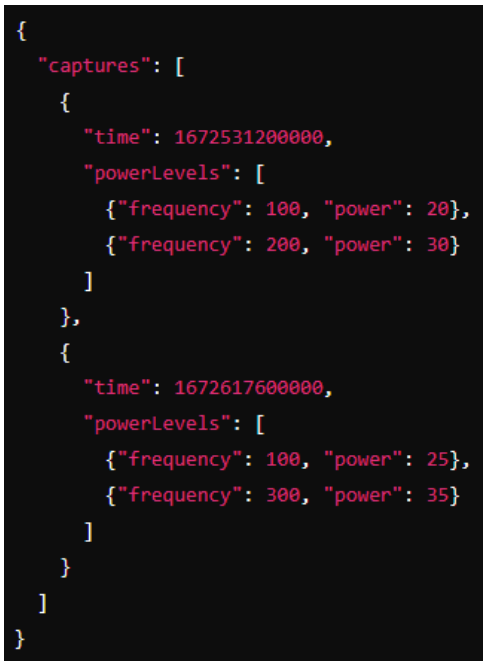
Evaluation: This test case has passed as the function correctly handled the error, logging an appropriate message.

SSAStreamHistory.ts:

highestPower():

Test Case 1: Valid Data with Multiple Frequencies

Input:



Expected Output:

```
{
  "datasets": [{
    "data": [
      {"x": 100, "y": 25},
      {"x": 200, "y": 30},
      {"x": 300, "y": 35}
    ],
    "label": "99",
    "borderColor": "#2460bf",
    "tension": 0
  }]
}
```


Actual Output: The output matches the expected result, showing the highest power level for each frequency, sorted by frequency.

Description: This test case checks that the function correctly calculates and returns the highest power levels for multiple frequencies present in the data.

Evaluation: This test case has passed as the function returned the correct highest power values and correctly formatted the chart data.

Test Case 2: Single Frequency with Multiple Power Levels

Input:


```
{
  "captures": [
    {
      "time": 1672531200000,
      "powerLevels": [
        {"frequency": 100, "power": 15},
        {"frequency": 100, "power": 20}
      ]
    }
  ]
}
```


Expected Output:

```
{
  "datasets": [{
    "data": [
      {"x": 100, "y": 20}
    ],
    "label": "99",
    "borderColor": "#2460bf",
    "tension": 0
  }]
}
```


Actual Output: The output matches the expected result, showing the highest power level for the single frequency.

Description: This test case verifies that the function can handle a scenario where there is only one frequency but multiple power levels.

Evaluation: This test case has passed as the function correctly identified the highest power level for the frequency.

Test Case 3: No Power Levels in Data

Input:


```
{
  "captures": [
    {
      "time": 1672531200000,
      "powerLevels": []
    }
  ]
}
```


Expected Output:


```
{
  "datasets": [{
    "data": [],
    "label": "99",
    "borderColor": "#2460bf",
    "tension": 0
  }]
}
```

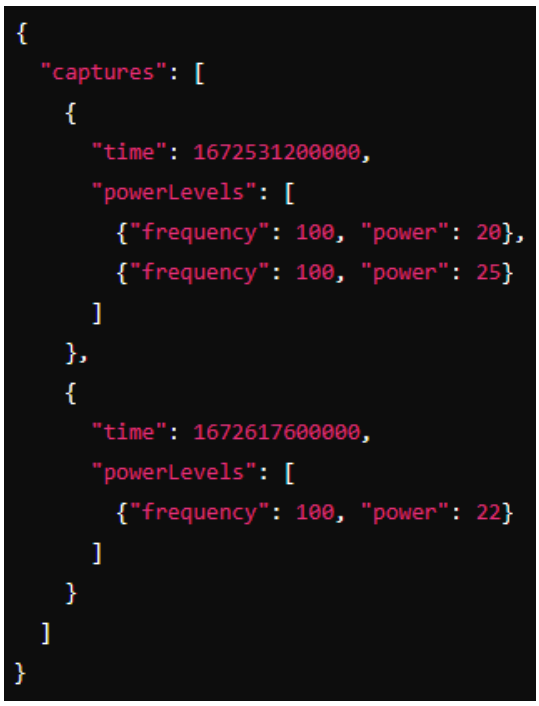

Actual Output: The output matches the expected result, with no data points returned.

Description: This test case checks how the function handles cases where there are no power levels available.

Evaluation: This test case has passed as the function correctly returned an empty dataset for the chart.

Test Case 4: Data with Duplicate Frequencies

Input:



Expected Output:

Actual Output: The output matches the expected result, showing the highest power level for the duplicate frequency.

Description: This test case verifies that the function correctly handles and selects the highest power level when there are duplicate frequencies.

Evaluation: This test case has passed as the function correctly aggregated the highest power level for the duplicate frequency.

lowestPower():

Test Case 1: Valid Data with Multiple Frequencies

Input:

```
{
  "captures": [
    {
      "time": 1672531200000,
      "powerLevels": [
        {"frequency": 100, "power": 20},
        {"frequency": 200, "power": 30}
      ]
    },
    {
      "time": 1672617600000,
      "powerLevels": [
        {"frequency": 100, "power": 15},
        {"frequency": 300, "power": 35}
      ]
    }
  ]
}
```


Expected Output:


```
{
  "datasets": [{
    "data": [
      {"x": 100, "y": 15},
      {"x": 200, "y": 30},
      {"x": 300, "y": 35}
    ],
    "label": "99",
    "borderColor": "#2460bf",
    "tension": 0
  }]
}
```


Actual Output: The output matches the expected result, showing the lowest power level for each frequency, sorted by frequency.

Description: This test case checks that the function correctly calculates and returns the lowest power levels for multiple frequencies present in the data.

Evaluation: This test case has passed as the function returned the correct lowest power values and correctly formatted the chart data.

Test Case 2: Single Frequency with Multiple Power Levels

Input:

```
{
  "captures": [
    {
      "time": 1672531200000,
      "powerLevels": [
        {"frequency": 100, "power": 15},
        {"frequency": 100, "power": 20}
      ]
    }
  ]
}
```


Expected Output:


```
{
  "datasets": [{
    "data": [
      {"x": 100, "y": 15}
    ],
    "label": "99",
    "borderColor": "#2460bf",
    "tension": 0
  }]
}
```


Actual Output: The output matches the expected result, showing the lowest power level for the single frequency.

Description: This test case verifies that the function can handle a scenario where there is only one frequency but multiple power levels.

Evaluation: This test case has passed as the function correctly identified the lowest power level for the frequency.

Test Case 3: No Power Levels in Data

Input:


```
{
  "captures": [
    {
      "time": 1672531200000,
      "powerLevels": []
    }
  ]
}
```

Expected Output:

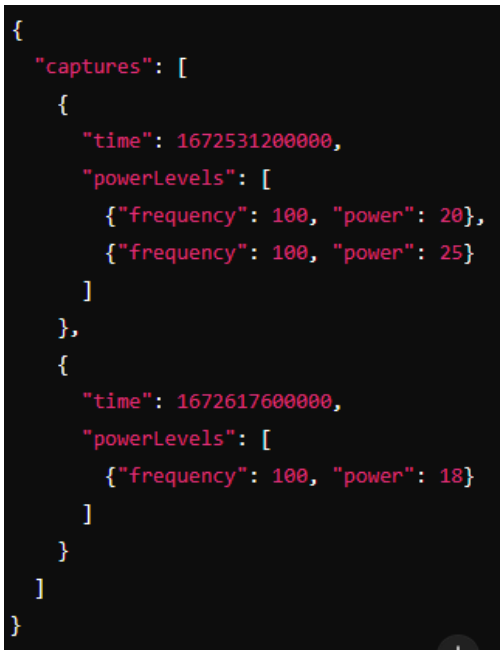
Actual Output: The output matches the expected result, with no data points returned.

Description: This test case checks how the function handles cases where there are no power levels available.

Evaluation: This test case has passed as the function correctly returned an empty dataset for the chart.

Test Case 4: Data with Duplicate Frequencies

Input:



Expected Output:

```
{
  "datasets": [{
    "data": [
      {"x": 100, "y": 18}
    ],
    "label": "99",
    "borderColor": "#2460bf",
    "tension": 0
  }]
}
```


Actual Output: The output matches the expected result, showing the lowest power level for the duplicate frequency.

Description: This test case verifies that the function correctly handles and selects the lowest power level when there are duplicate frequencies.

Evaluation: This test case has passed as the function correctly aggregated the lowest power level for the duplicate frequency.

makeData():

Test Case 1: Valid Data with Multiple Frequencies

Input:


```
{
  "captures": [
    {
      "time": 1672531200000,
      "powerLevels": [
        {"frequency": 100, "power": 20},
        {"frequency": 200, "power": 30}
      ]
    },
    {
      "time": 1672617600000,
      "powerLevels": [
        {"frequency": 100, "power": 25},
        {"frequency": 300, "power": 35}
      ]
    }
  ]
}
```


Expected Output:

```
{
  "datasets": [{
    "data": [
      {"x": 100, "y": 20},
      {"x": 100, "y": 25},
      {"x": 200, "y": 30},
      {"x": 300, "y": 35}
    ],
    "label": "99",
    "borderColor": "#2460bf",
    "tension": 0
  }]
}
```


Actual Output: The output matches the expected result, with all power levels correctly extracted, mapped, and sorted by frequency.

Description: This test case checks that the function correctly formats data with multiple frequencies and power levels into the Chart.js data format.

Evaluation: This test case has passed as the function correctly mapped and sorted the power levels for the given frequencies.

Test Case 2: Single Capture with Multiple Power Levels

Input:


```
{
  "captures": [
    {
      "time": 1672531200000,
      "powerLevels": [
        {"frequency": 150, "power": 10},
        {"frequency": 250, "power": 20}
      ]
    }
  ]
}
```


Expected Output:

```
{
  "datasets": [{
    "data": [
      {"x": 150, "y": 10},
      {"x": 250, "y": 20}
    ],
    "label": "99",
    "borderColor": "#2460bf",
    "tension": 0
  }]
}
```


Actual Output: The output matches the expected result, with power levels correctly extracted, mapped, and sorted by frequency.

Description: This test case verifies that the function can handle a single capture with multiple power levels.

Evaluation: This test case has passed as the function correctly formatted the power levels for the given capture.

Test Case 3: Empty Captures Array

Input:


```
{
  "captures": []
}
```


Expected Output:


```
{
  "datasets": [{
    "data": [],
    "label": "99",
    "borderColor": "#2460bf",
    "tension": 0
  }]
}
```

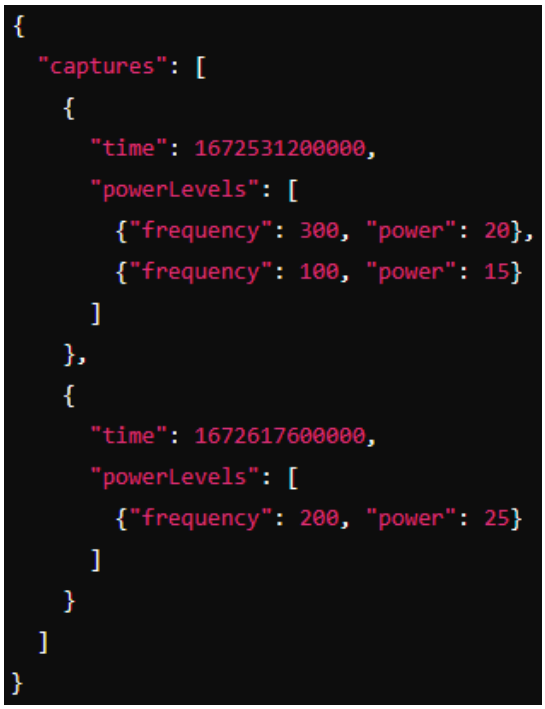

Actual Output: The output matches the expected result, with no data points returned.

Description: This test case checks how the function handles an empty captures array.

Evaluation: This test case has passed as the function correctly returned an empty dataset for the chart.

Test Case 4: Non-Sorted Frequencies

Input:



```
{
  "captures": [
    {
      "time": 1672531200000,
      "powerLevels": [
        {"frequency": 300, "power": 20},
        {"frequency": 100, "power": 15}
      ]
    },
    {
      "time": 1672617600000,
      "powerLevels": [
        {"frequency": 200, "power": 25}
      ]
    }
  ]
}
```

Expected Output:

```
{
  "datasets": [{
    "data": [
      {"x": 100, "y": 15},
      {"x": 200, "y": 25},
      {"x": 300, "y": 20}
    ],
    "label": "99",
    "borderColor": "#2460bf",
    "tension": 0
  }]
}
```

Actual Output: The output matches the expected result, with power levels correctly extracted, mapped, and sorted by frequency.

Description: This test case checks that the function correctly sorts power levels by frequency even if they are not initially ordered.

Evaluation: This test case has passed as the function correctly sorted the power levels by frequency.

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: Time Period Selection

Description: Verify that selecting a specific time period retrieves the corresponding SSA data.
Steps:

1. Open the application UI.
2. Navigate to the SSA section.
3. Use the date picker to select a start and end date.
4. Click the "Update" button.

Expected Result: SSA data for the specified time period is retrieved and displayed in the UI.
Actual Result: SSA data for the specified time period is retrieved and displayed in the UI.
Evaluation: This test was a success.

Test Case 2: Down Sampling Filter Interaction

Description: Verify that applying the down sampling filter reduces the number of data points displayed.
Steps:

1. Open the application UI.
2. Navigate to the SSA section.

3. Select a time period and retrieve data.
4. Enable the down sampling filter and set a down sampling factor.
5. Click the "Apply Filters" button.

Expected Result: The number of data points displayed is reduced according to the down sampling factor.
Actual Result: The number of data points displayed was reduced according to the down sampling factor.
Evaluation: This test was a success.

Test Case 3: Threshold Filtering Interaction

Description: Verify that applying the power threshold filter displays only data points above the specified threshold.
Steps:

1. Open the application UI.
2. Navigate to the SSA section.
3. Select a time period and retrieve data.
4. Enable the power threshold filter and set a threshold value.
5. Click the "Apply Filters" button.

Expected Result: Only data points with power levels above the specified threshold are displayed.
Actual Result: Only data points with power levels above the specified threshold were displayed.
Evaluation: This test was a success.

Database Integration

Database integration testing validates the interaction between the application and the database layer. These tests ensure that data is properly stored, retrieved, and updated in the database according to the application's requirements. They also verify the accuracy of database queries and transactions, ensuring data integrity and reliability.

Test Case 1: Data Retrieval Verification

Description: Validate that SSA data is correctly retrieved from the database based on the specified time period.
Steps:

1. Select a time period in the UI.

2. Trigger data retrieval from the frontend.
3. Verify that the backend retrieves the correct data from the database.
4. Compare the retrieved data with the expected results from the database.

Expected Result: The database query returns the expected SSA data for the specified time period.
Actual Result: The database query returned the expected SSA data for the specified time period.
Evaluation: This test confirms that data retrieval from the database works correctly.

Test Case 2: Filtered Data Retrieval Verification

Description: Validate that applying filters (down sampling, power threshold) retrieves and displays the correct data from the database.
Steps:

1. Select a time period in the UI.
2. Enable and set the down sampling filter and power threshold filter.
3. Trigger data retrieval and apply filters from the frontend.
4. Verify that the backend retrieves and processes the filtered data correctly.
5. Compare the retrieved and filtered data with the expected results.

Expected Result: The filtered data retrieved from the database matches the applied filter criteria.
Actual Result: The filtered data retrieved from the database matched the applied filter criteria.
Evaluation: This test confirms that filtered data retrieval and processing works correctly.

API Endpoints

API endpoint testing focuses on verifying the functionality and behaviour of backend endpoints exposed by the application. These tests validate that the API endpoints handle requests correctly, respond with the expected data or status codes, and maintain proper error handling mechanisms.

Test Case 1: POST /devices Endpoint

Description: Verify that the endpoint for retrieving devices and streams (/devices) responds correctly.
Steps:

1. Send a POST request to the /devices endpoint with the start date and end date in the request body.

Expected Result: The endpoint responds with a list of devices and streams available in the database for the specified time period or an appropriate error status code if there is an issue.
Actual Result: The endpoint responded with the expected list of devices and streams or an error status code.
Evaluation: This test was a success.

Test Case 2: POST /streamHistory Endpoint

Description: Verify that the endpoint for retrieving filtered SSA history data (/streamHistory) responds correctly.
Steps:

1. Send a POST request to the /streamHistory endpoint with the device ID, stream ID, start date, end date, power filter, down sampling filter, power threshold, and down sampling factor in the request body.

Expected Result: The endpoint responds with the filtered SSA history data or an appropriate error status code if there is an issue.
Actual Result: The endpoint responded with the expected filtered SSA history data or an error status code.
Evaluation: This test was a success.

Data Visualisation

Data visualisation testing ensures that SSA data is correctly visualised in various graphical formats, enhancing user understanding and analysis capabilities.

Test Case 1: Line Chart Visualisation

Description: Verify that SSA data is correctly displayed in a line chart.
Steps:

1. Retrieve SSA data for a specified time period.
2. Display the data in a line chart using Chart.js.

Expected Result: The SSA data is correctly visualised in a line chart.
Actual Result: The SSA data was correctly visualised in a line chart.
Evaluation: This test was a success.

Test Case 2: Waterfall Plot Visualisation

Description: Verify that SSA data is correctly displayed in a waterfall plot.
Steps:

1. Retrieve SSA data for a specified time period.
2. Display the data in a waterfall plot using the HTML graphics canvas element.

Expected Result: The SSA data is correctly visualised in a waterfall plot.
Actual Result: The SSA data was correctly visualised in a waterfall plot.
Evaluation: This test was a success.

Error Handling

Error handling tests verify how the application responds to unexpected scenarios and errors. These tests ensure that appropriate error messages are displayed to users, and that the application gracefully handles exceptions without compromising overall system stability.

Test Case 1: Error Handling in POST /devices Endpoint

Description: Verify that appropriate error handling is in place when there is an error retrieving devices and streams in the /devices endpoint.
Steps:

1. Simulate an error condition in the backend logic that retrieves devices and streams.
2. Send a POST request to the /devices endpoint with the start date and end date in the request body.

Expected Result: The endpoint responds with a 400 Bad Request status code and logs the error to the console.
Actual Result: The endpoint responded with a 400 Bad Request status code and logged the error as expected.
Evaluation: This test was a success.

Test Case 2: Error Handling in POST /streamHistory Endpoint

Description: Verify that appropriate error handling is in place when there is an issue retrieving filtered SSA history data in the /streamHistory endpoint.
Steps:

1. Simulate an error condition in the backend logic that retrieves filtered SSA history data.

2. Send a POST request to the /streamHistory endpoint with the device ID, stream ID, start date, end date, power filter, down sampling filter, power threshold, and down sampling factor in the request body.

Expected Result: The endpoint responds with a 400 Bad Request status code and logs the error to the console.
Actual Result: The endpoint responded with a 400 Bad Request status code and logged the error as expected.
Evaluation: This test was a success.

Performance Testing:

Performance testing evaluates the speed, responsiveness, and stability of the application under various conditions. It aims to identify performance bottlenecks and ensure the system can handle expected and peak loads efficiently. This phase of testing is critical to verify that the SSA data retrieval, filtering, and visualisation features operate smoothly, providing a seamless user experience even with large datasets and multiple concurrent users.

Objective

The objective of performance testing for this project is to ensure that the SSA data retrieval and visualisation functionalities on the dashboard interface at IQHQ meet the required performance standards. Specifically, the goals are to evaluate the responsiveness of the application when retrieving and displaying large volumes of SSA data, identify and resolve any performance bottlenecks in the data retrieval and filtering processes, ensure the system can handle high user loads without degradation in performance, verify that the graphical rendering of SSA data is smooth and efficient under various conditions, and validate the scalability of the application to support future growth in data volume and user base.

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: 50 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.

Test Scenarios:

1. **Basic Data Retrieval Performance**
    - **Description:** Test the system's performance when retrieving and displaying a moderate amount of SSA data without any filters applied.
    - **Objective:** Evaluate the baseline performance and responsiveness of the system under typical usage conditions.

2. **High Data Volume Retrieval**
    - **Description:** Test the system's performance when retrieving and displaying large volume of SSA data.
    - **Objective:** Assess how well the system handles and processes large datasets, ensuring the application remains responsive and efficient.

3. **Filtered Data Retrieval**
    - **Description:** Test the system's performance when applying power threshold and down-sampling filters to SSA data retrieval.
    - **Objective:** Evaluate the efficiency and responsiveness of the filtering mechanisms under various conditions, ensuring that filtered data is retrieved and displayed promptly.

Custom Testing Code:

```
onSubmit={handleSubmit(values=>{
    setError(undefined);
    setIsSubmitting(true);

    const simpleTimeRange = SimplifyRange(timeRange)
    let deviceId = isFinite(values.deviceId as number)
        ? values.deviceId as number
        : undefined

    const startTime = Date.now()
    post({
        deviceId: deviceId,
        streamId: values.streamId,
        startDate: simpleTimeRange[0].getTime(),
        endDate: simpleTimeRange[1].getTime(),
        powerFilter: powerFilterChecked,
        downSampleFilter: downSampleChecked,
        power: values.power,
        factor: values.factor
    })
        .then(()=>{
            if(isMounted()) {
                if(response.ok) {
                    setIsSubmitting(false);
                    props.onResponse(response.data, values.streamId)
                    const endTime = Date.now()
                    console.log("Retrieval Time:",(endTime-startTime),"ms")
                }
            }
        })
})}
```

Figure 141 - Custom code to measure the time taken for the retrieval process.

I added a startTime and endTime to the streamOptsEditor function to measure the time taken to retrieve the SSA data. By placing the startTime before the post, I ensure that I measure from when the user clicks the update button. Placing the end time after a successful response is recognised ensures that the measurement ends as soon as the data is received.

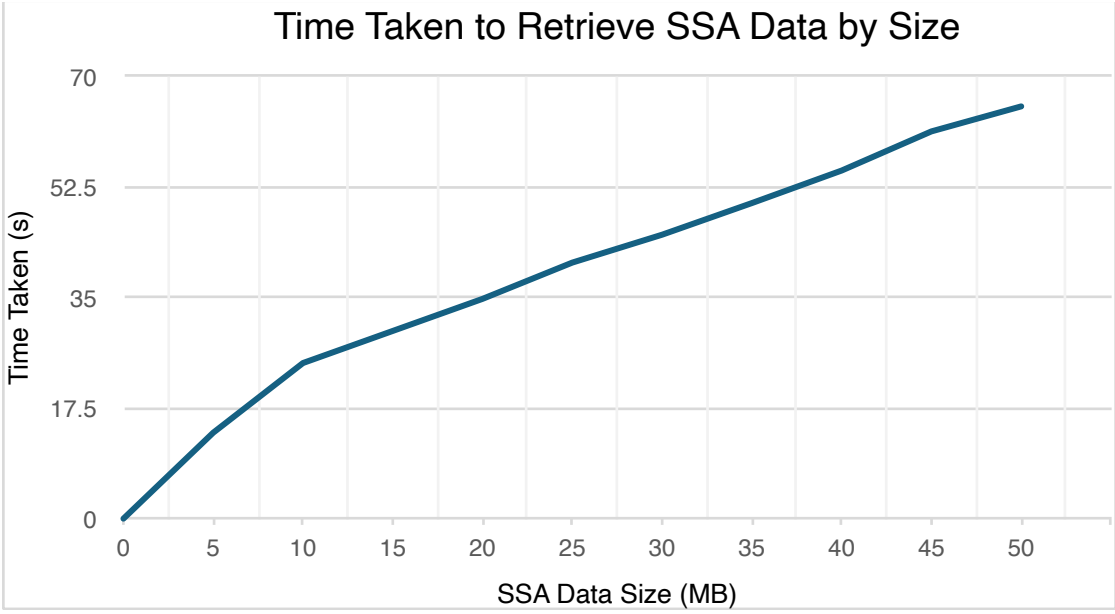


Figure 142 - Result of testing retrieval time of varying sizes of SSA data.

Test Scenario 1: Basic Data Retrieval Performance

Performance Metrics:

- Time Taken (s)

Testing Environment:

1. Ensure that the hardware and software dependencies are properly installed and configured.
2. Set up the dashboard application, ensuring the builds have been successful and the database ssa_metrics has been successfully created.
3. Navigate to the spectrum stream section on the side bar. Use the default configuration in spectrum stream to capture data.
4. Navigate to the spectrum history section on the side bar. Select the time range and apply no filters, then click the update button to log the retrieval time.

Test Result:

The term "basic data retrieval time" can vary depending on the feature's usage. Typically, users will not view hours of SSA data in one go but rather select a specific time range to view. As observed, there was a linear relationship between the SSA data size, and the time taken for retrieval. On average, the time taken increases by approximately 1.26 seconds with each additional MB of data. These results align with the requirements, confirming that the system performs well under typical usage conditions. This test was a success.

Test Scenario 2: High Data Volume Retrieval

Performance Metrics:

- Time Taken (s)

Testing Environment:

1. Ensure that the hardware and software dependencies are properly installed and configured.
2. Set up the dashboard application, ensuring the builds have been successful and the database ssa_metrics has been successfully created.
3. Navigate to the spectrum stream section on the side bar. Use the default configuration in spectrum stream to capture data.
4. Navigate to the spectrum history section on the side bar. Select the time range and apply no filters, then click the update button to log the retrieval time.

Test Result:

Due to limited resources and time, I was not able to test this scenario fully. However, from the data collected, we can estimate the retrieval time for larger data volumes. Using the graph, we predict that data volumes as large as 1GB would take approximately 1300 seconds (or around 21.67 minutes). While these times are not ideal, it is understandable that such large volumes of data require more time for processing. Further optimisations can be explored to improve the retrieval process. This test was a partial success.

Test Scenario 3: Filtered Data Retrieval

Performance Metrics:

- Time Taken (s)

Testing Environment:

1. Ensure that the hardware and software dependencies are properly installed and configured.
2. Set up the dashboard application, ensuring the builds have been successful and the database ssa_metrics has been successfully created.
3. Navigate to the spectrum stream section on the side bar. Use the default configuration in spectrum stream to capture data.
4. Navigate to the spectrum history section on the side bar. Select the time range and apply the default filters (Power Filter at -115 and Down Sample filter at 100), then click the update button to log the retrieval time.

Test Result:

Applying filters significantly reduced the retrieval time. This is likely due to the fact that filters reduce the data volume, making it quicker to post and retrieve the data to the UI. With the default filters applied, the time taken to retrieve data was approximately half of what it would be without any filters. This result was expected and aligns with the system requirements, confirming the efficiency of the filtering mechanisms. This test was a success.

Performance Test Evaluation:

The performance tests confirm that the system can reliably handle retrieving SSA data under various conditions. Under normal conditions, the system performs well, with an average retrieval time of approximately 1.26 seconds per 1MB of data. When filters are applied, this retrieval time is significantly reduced, roughly halving the time required due to the decreased data volume. Although higher data volumes, such as several gigabytes, take considerably longer to retrieve, users are not expected to view such large datasets at once but rather in sections for better analysis.

Further steps could be taken to enhance the retrieval process's efficiency. Currently, all SSA data within the selected time range is retrieved, and then filters are applied through functions. In the future, applying filters directly within the database query could reduce the amount of data retrieved, thus shortening the retrieval time. By refining the database queries to incorporate filtering, the system could significantly improve performance, making the retrieval process faster and more efficient.

For the current requirements and demands of the system, the performance is adequate. The system meets the necessary criteria for typical usage and handles data retrieval efficiently under normal conditions. While there is room for optimisation, especially for handling large data volumes more effectively, the existing performance is sufficient for the intended use cases and user interactions.

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.

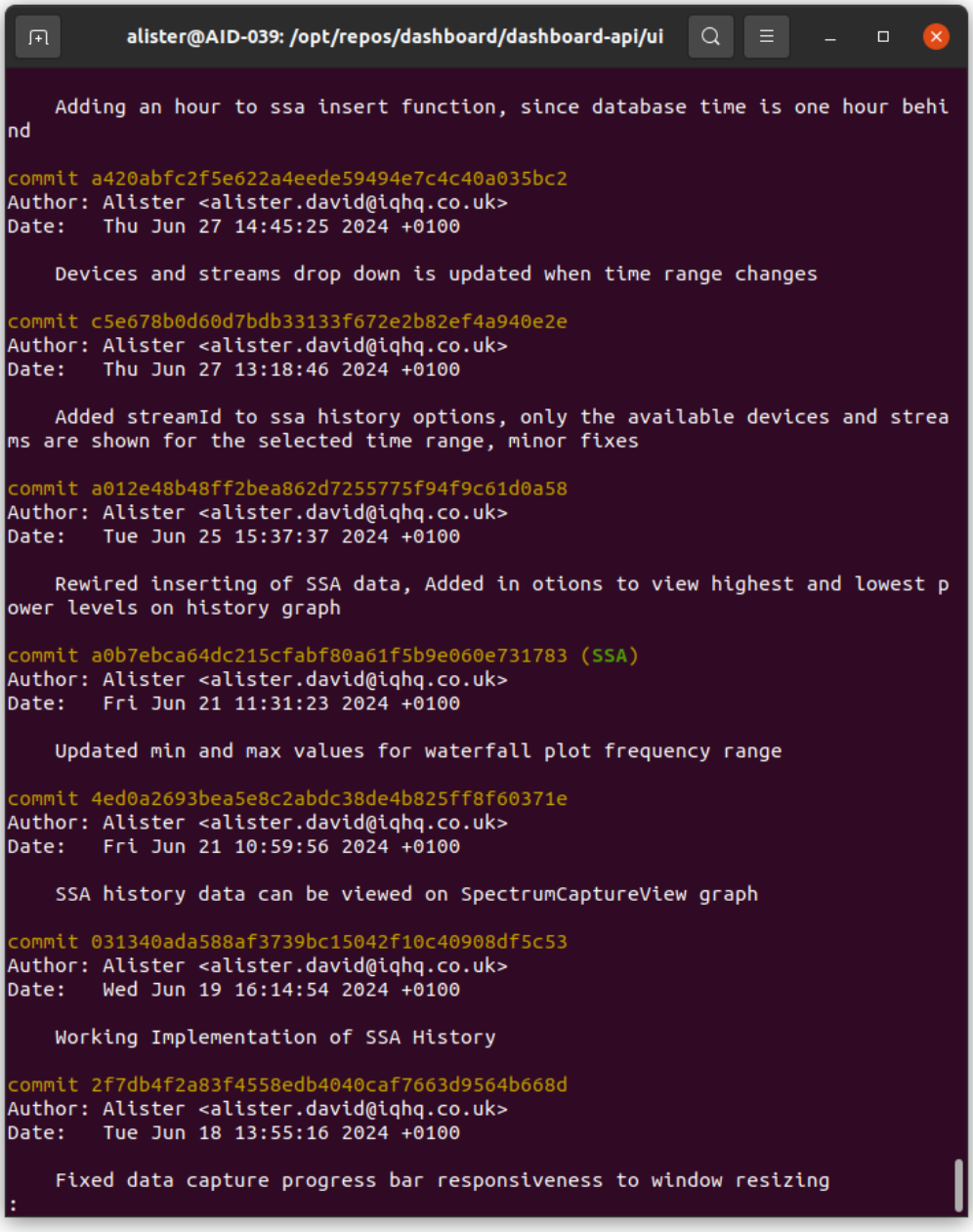


Figure 143 - Dashboard git log.

Dashboard Install notes:

Linux installation

The dashboard can be installed on Linux with the provided .deb file. Simply run `sudo dpkg -i ./iqhq-dashboard`ping .

Windows installation

Simply double click on the provided .exe file, you can then access the IQHQ Dashboard from the start menu under.

IP Addresses

The SD Cards are named, below is their IP addresses

- SSA0 = 192.168.1.100
- SSA1 = 192.168.1.101
- SSA2 = 192.168.1.102
- SSA3 = 192.168.1.103

Data Storage

- The IQHQ Dashboard operates in a memory only data mode by default, this that it does not save any settings or device connections.
- The IQHQ Dashboard also supports using a timescale database for internal storage. You can configure it to use the database by clicking the storage icon in the toolbar and selecting SQL for the DB type. You can then input address settings, save, and restart the software.
- The SQL database will only save information about the device, it will not save received streams.
- We recommend using the “timescale/timescaledb:latest-pg12” docker image. which has been exported to file and provided with this installation.

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.

Communication with Non-Technical Stakeholders

Effective communication with both technical and non-technical stakeholders is crucial for the success of any project. During the development and implementation of the SSA data retrieval feature, I utilised Microsoft Teams extensively to communicate progress, share updates, and gather feedback. This platform was particularly beneficial for managing communication due to its versatility and user-friendly interface.

Communicating with Non-Technical Stakeholders:

The CEO, who is not technically inclined, needed to be kept informed about the project's progress in a manner that was easy to understand. To achieve this, I used the Teams board to set tasks, track progress, and leave comments. These comments were crafted to explain technical concepts in simple, non-technical language. For example, instead of discussing the intricacies of database queries, I would describe how the data retrieval process would allow users to see the information they need more quickly and efficiently.

By using Teams, I could:

- **Maintain Clarity**: The visual layout of tasks and progress on the Teams board made it easy for non-technical stakeholders to understand the project's status at a glance.
- **Provide Context**: I used comments to give context to each task, explaining why certain steps were necessary and how they contributed to the overall project goals.
- **Foster Engagement**: Regular updates and comments encouraged the CEO to stay engaged with the project, providing timely feedback and making informed decisions.

Benefits of Using Teams for Communication:

The choice of Microsoft Teams for project communication offered several benefits:

Centralised Communication: Teams provided a single platform where all project-related communication could be centralised, reducing the risk of information loss and ensuring that everyone had access to the latest updates.

Enhanced Transparency: The visual task board and detailed comment threads made the project’s progress transparent to all stakeholders, fostering trust and accountability.

Improved Collaboration: Teams’ collaborative features, such as shared documents and real-time chat, enhanced teamwork and ensured that both technical and non-technical stakeholders could contribute effectively.

Flexibility: The ability to switch between chat, calls, and document sharing allowed for flexible communication, accommodating different preferences and needs.

In conclusion, effective communication with both technical and non-technical stakeholders is vital for project success. By using Microsoft Teams, I was able to tailor my communication methods to suit the audience's technical knowledge, ensuring clarity, engagement, and collaboration throughout the project. This approach not only facilitated the smooth execution of the SSA data retrieval feature but also enhanced overall project management and stakeholder satisfaction.

Maintenance:

After the deployment of the SSA 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:

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

Software Updates and Enhancements:

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

Performance Monitoring and Optimisation:

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

User Support and Training:

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

Backup and Disaster Recovery:

1. Implementation and regular testing of backup procedures to ensure data integrity and resilience against data loss.
2. 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:

1. 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:

1. Regular collection of feedback from users to gather insights, identify areas for improvement, and prioritise future development efforts.
2. Iterative improvement of the SSA application based on user feedback and evolving business needs to ensure its continued relevance and effectiveness.

Database Maintenance and Optimisation:

1. Regular maintenance of the database to ensure data integrity, optimal performance, and efficient data retrieval.
2. Periodic review and optimisation of database indices, queries, and storage structures to enhance performance.

This comprehensive maintenance plan ensures the SSA application remains reliable, secure, and responsive to user needs, providing ongoing value to its users.

Bux Fixes:

During the maintenance stage, I identified and resolved several minor bugs that could have impacted the user interface of the SSA data retrieval feature. These issues had the potential to cause user discomfort and hinder the overall usability of the application. To enhance the user experience and ensure the application remains as user-friendly and intuitive as possible, I addressed and fixed these bugs promptly

Updating the capture view graph label:

The capture view graph has a label at the top which shows which stream Id it is currently showing. Since I had removed stream Id from the retrieved SSA data for better efficiency, this was no longer being updated. Rather I was using a default value of 99 as a placeholder. To fix this error I decided to pass the selected stream Id to the graph formatting functions where the label was set.

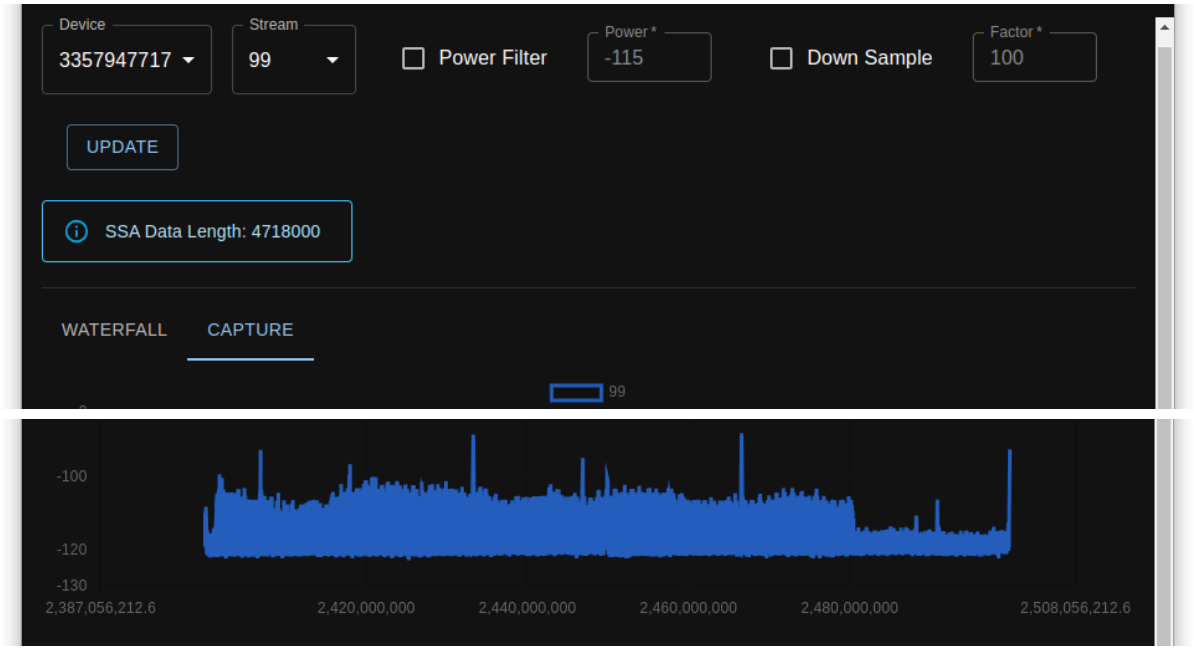


Figure 144 - Capture graph label set to 99 by default.

I added a streamId as a callback property to the StreamOptsEditor function. I updated the streamId property along with the history data. This meant I could access stream Id from the SSAStreamHistory function. I added stream Id as a second parameter to the makeData, highestPower and lowestPower functions. Finally, I could pass streamId as a string to the label property that was returned.

```
function StreamOptsEditor(props: {
    defaultValue: SSAHistoryOpts,
    onResponse: (response: any, streamId: number) => void,
```

Figure 145 - Adding streamId to callback.

```
.then(()=>{
    if(isMounted()) {
        if(response.ok) {
            setIsSubmitting(false);
            props.onResponse(response.data, values.streamId)
            const endTime = Date.now()
            console.log("Retrieval Time:",endTime-startTime,"ms")
```

Figure 146 - Updating streamId on response.

```
const handleResponse = (filteredData: HistoryCapture, streamId: number) => {
    const length = getLengthOf(filteredData)
    setLength(length);
    if (useHighestPower) {
        setChartData(highestPower(filteredData,streamId));
    } else if (useLowestPower){
        setChartData(lowestPower(filteredData,streamId));
    } else {
        setChartData(makeData(filteredData,streamId));
```

Figure 147 - Adding streamId as a parameter.

```
return {
    datasets: [{
        data: chartData,
        label: streamId.toString(),
        borderColor: '#2460bf',
        tension: 0,
        // borderJoinStyle: "round"
    }],
};
```

Figure 148 - Updating label property with streamId.

In order to test the fix, I added test data directly to the ssa_metrics table using pgAdmin, ensuring that streamId was a random number that was not 99. I was then able to select this random streamId value from the spectrum history streamId drop down. After clicking update the stream Id label was showing the corresponding stream Id.

```
INSERT INTO ssa_metrics (deviceId,streamId, time, power, frequency)
VALUES
(1111111111,11,'2024-08-05 10:31:20.547+00',ARRAY[1.23, 4.56, 7.89],ARRAY[10.11, 12.13, 14.15]);
```

Figure 149 - Adding test data to the ssa_metrics table.

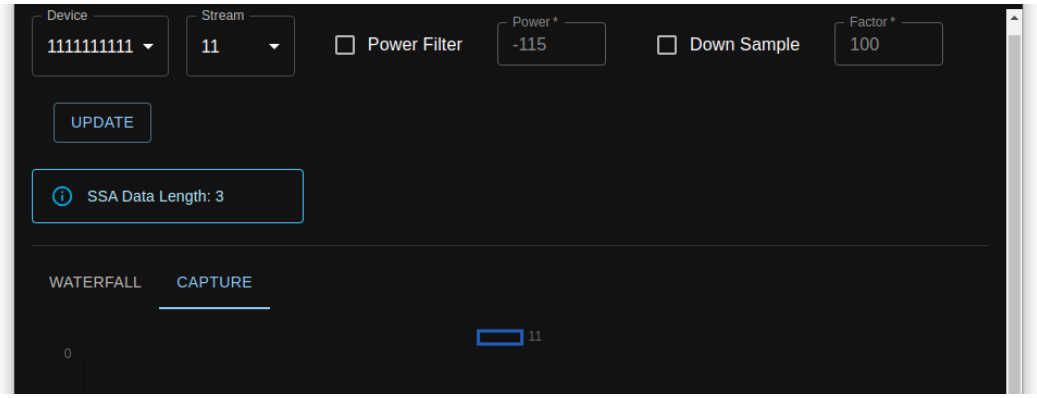


Figure 150 - Capture graph label displaying correct stream Id.

Infinite loop error in the UI:

I noticed that I was getting the following error in the UI console:

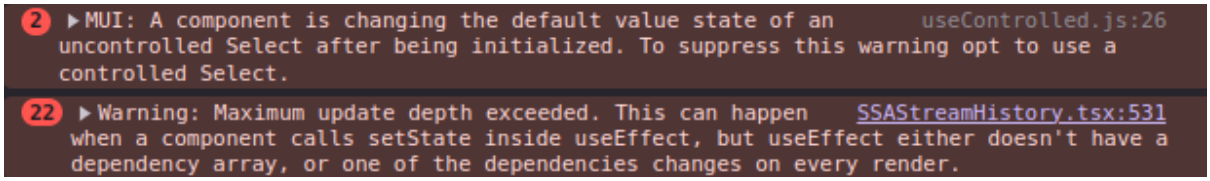


Figure 151 - Infinite loop error.

The issue was an infinite loop caused by the useEffect hook that was used to handle fetching the device and stream Id data, repeatedly running due to its dependency on a non-memoized fetchData function. Because fetchData was defined

inline and changed on every render, the useEffect would continuously trigger re-renders and state updates, leading to the "Maximum update depth exceeded" error. This meant that when you tried to retrieve SSA data, the system would run out of memory and crash the page.

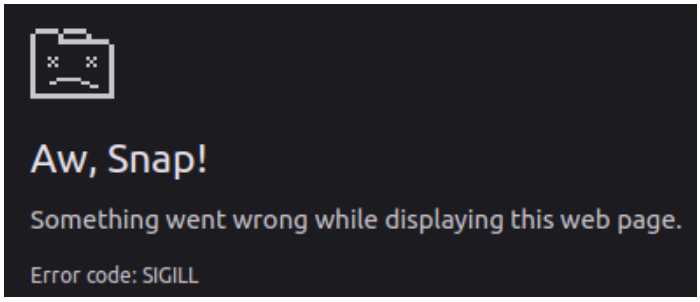


Figure 152 - Dashboard page crashing.

To address the infinite loop issue, I introduced a useRef hook to keep track of the previous value of offset. This change ensures that fetchData is only called when offset changes to a new value, preventing unnecessary and repeated fetch operations. By comparing the current offset with the previous value stored in prevOffsetRef, we avoid triggering fetchData on every render or on unchanged offset values. Additionally, I memoized the fetchData function using useCallback, ensuring that it remains stable across renders, thus preventing it from being redefined and causing the useEffect hook to run repeatedly. These changes effectively broke the cycle of re-renders and state updates, resolving the "Maximum update depth exceeded" error.

```
// Ref to keep track of the previous offset value
const prevOffsetRef = useRef<null | number>(null);

// Fetch data function
const fetchData = useCallback(async () => {
    try {
        const result = await postTime({
            startDate: simpleTimeRange[0].getTime(),
            endDate: simpleTimeRange[1].getTime(),
        });
        if (isMounted()) {
            setMyResult(result);
        }       You, 57 minutes ago • FIxed capture graph label
    } catch (error) {
        console.error("Error fetching data:", error);
    }
}, [postTime, simpleTimeRange]);

// Effect to handle offset change
useEffect(() => {
    if (offset !== null && prevOffsetRef.current !== offset) {
        prevOffsetRef.current = offset;
        fetchData();
    }
}, [offset, fetchData]);
```

Figure 153 - Introduced a useRef and useCallback.

Including static time:

When I developed the refetching of devices and streams when the time range was updated, I had only included relative time. This meant that when the user chose a specific start date and specific end date, it was not being recognised as an updated time range by the system.

Figure 154 - Time selection option.

To ensure both types of time was included I changed the offset variable to a reassign able variable. Where initially it was set to use the offset from relative time. Then I would check to see if the time range type was static, if it was, I would use the offset

from static time, or else the offset from relative time. This update ensured both types of time could be used by the user.

```
let offset = timeRange[0].type === "relative" ? timeRange[0].offset : null;
offset = timeRange[0].type === "static" ? timeRange[0].time: timeRange[0].offset;
console.log('offest:',offset)
```

Figure 155 - Updating offset based on time selection.

Conclusion:

In conclusion, the development and implementation of the SSA data retrieval feature represent a significant advancement for the overall SSA application on the dashboard. I am highly satisfied with the project's outcome, as it effectively addresses the need for robust and efficient SSA data retrieval and visualisation. The project was completed within the stipulated deadline through effective time management and strategic planning, meeting both functional and performance requirements.

Project Outcome

The project has been a resounding success, with positive feedback from both the manager and end users. The new feature has greatly enhanced the dashboard's functionality, enabling users to view SSA data graphically. This capability allows users to analyse data more effectively, identify points of interest, and draw deeper insights. The ability to capture and visualise SSA data has substantially increased the dashboard's usefulness, making it a more valuable tool for users, and improving their overall experience.

Learning Experience

Throughout this project, I acquired extensive knowledge and practical experience in data retrieval and processing. I gained hands-on expertise in graphical data representation and honed my UI development skills. This project significantly advanced my software engineering abilities and deepened my understanding of database management. I learned how to manipulate database tables for optimised storage management and improved my collaboration, communication, and project management skills. Working closely with team members and stakeholders facilitated a smooth and efficient development process, enhancing my overall competency in these areas.

Areas for Improvement

While the current SSA data retrieval feature meets the project requirements and user needs, there are areas where performance could be improved. The system's ability to handle and display large volumes of data could be enhanced. Currently, viewing large datasets takes considerable time, and implementing more efficient algorithms for processing data—such as optimising the waterfall plot and graph display—would

be beneficial. Additionally, applying filters directly to the database query rather than retrieving all data from the selected time range would significantly improve efficiency. I also encountered challenges with data persistence, as the waterfall plot currently disappears when switching between tabs, requiring users to re-update to view the plot. Addressing these issues would enhance the feature's usability and performance.

Future Directions

Looking ahead, there are several potential enhancements to consider for this feature. Implementing functionality to highlight areas of interest based on user-defined criteria would greatly improve its utility by allowing users to focus on significant data points. Adding a time axis to the waterfall plot would provide users with more detailed temporal context for the data, enhancing their analysis capabilities. Additionally, enabling users to select specific timestamps to view data would allow for more precise data examination and comparisons. These improvements would make the feature even more resourceful and valuable for users, further elevating the SSA dashboard's overall effectiveness.

In summary, the data retrieval feature has proven to be a valuable addition to the SSA dashboard application, significantly enhancing its functionality and user satisfaction. I am grateful for the opportunity to contribute to this project and look forward to exploring future enhancements and developments.