OULUN AMMATTIKORKEAKOULU

Hieu Nguyen

A TEST MANAGEMENT TOOL FOR 5G RADIO NETWORK: REAL-TIME RADIO POSITION CONTROL WEB APP

A TEST MANAGEMENT TOOL FOR 5G RADIO NETWORK: REAL-TIME RADIO POSITION CONTROL WEB APP

Hieu Nguyen
Bachelor's Thesis
Autumn 2023
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Bachelor's Degree in Information Technology

Author(s): Hieu Nguyen
Title of the thesis: A Test Management Tool for 5G Radio Network: Real-time Radio Position Control Web App
Thesis examiner(s): Jouni Juntunen
Term and year of thesis completion: Spring 2024 Pages: 46

The rapid development of 5G technology increases the frequency with which 5G radio networks are developed, tested, and deployed to generate valuable products. To meet this fast-increasing demand, more over-the-air (OTA) chambers need to be built, as well as the technologies used in these chambers being upgraded. OTA chamber is a shielded room used for simulating radio testing scenarios. As a result, the test management tool needs more features to support running tests faster, more efficiently and easier remote management to those chambers. Specifically, a compact new user interface with features for browsing real-time radios status and controlling radio position in chambers is the main motivation for implementing this thesis.

The proposed approach uses the backend server based on FastAPI, a Python framework to connect to the chamber controller via NATS protocol. Furthermore, the backend server supports both Socket.IO and REST API, which allow the React.JS-based frontend to communicate with the backend for data, validation and logic.

After completion, the project met the essential objectives, such as improving the efficiency of testing radio positions and enabling easier remote management. It was then added as a module of the test management tool so that the end-users, also known as the testers, could utilize it.

Keywords: OTA chamber, test management tool, radio position control, NATS, FastAPI, Socket.IO, React.JS, real-time

CONTENTS

1 INTRODUCTION

Wireless networks are becoming increasingly crucial in everyday life. They enable Internet access and connection at any time and from any location. Radio networks, particularly for the emerging 5G technology, are a popular and vital wireless networking technology. But developing, testing, and deploying 5G radio networks is not an easy operation; especially testing radio networks as it requires specialized equipment such as the OTA chamber. OTA stands for over-the-air, which means that the signals or data are transferred through the air, or wirelessly. And OTA chamber is a shielded room used for simulating radio testing scenarios, this room stores radio components, such as the radio frequency modules, baseband processors, the antennas and so on. Furthermore, this testing requires sophisticated procedures, from preparation such as test line reservation, which is a method of booking radio resources for a chamber, to implementation such as radio position control, which is a procedure for changing the position of radio hardware in this chamber. Besides, an OTA testing chamber involves multiple stakeholders, such as developers, testers, and managers, who need to coordinate and monitor the radio testing process, particularly radio position manipulation, in real-time.

To address these issues, this project aims to develop a tool or web user interface (UI) application that enables users, also known as testers, to browse radio status and manage radio position when a radio is reserved and active in the chamber. These actions are updated in real time so that the users can have a clear and accurate view of the radio network testing status. It should be highlighted that the "radio" keyword in this thesis is also known as radio unit or module hardware or radio frequency module.

This work is a part of the DevOps team's test management tool, and it was built in Nokia's Otava in Oulu, Finland. Nokia - Oulu is known as the 'Home of Radio' where vital research and development is carried out (1) and there Otava, which stands for Over-The-Air Validation Area, is Nokia's flagship 5G testing centre, equipped with a variety of equipment such as antennas, radios, and chambers (2).

The web application is constructed up of two components: the backend and the frontend UI. Backend services include Web_api service and Socket_io service. Web_api service uses the Python-based FastAPI framework to provide a RESTful API to the frontend by creating new routes for each action. It also uses NATS protocol to communicate with the Manipulator, also known as the

NATS server and chamber controller within this thesis. Regarding socket.IO service, it serves as a connector for transmitting real-time data from the Manipulator to the Frontend. Finally, the frontend UI, which is built on the ReactJS library, connects to backend services, shows radio status in real-time using Socket.IO library and allows users to rotate radios over API backend by creating a new page in there.

The technologies in this project involve NATS, Socket.IO, FastAPI and ReactJS. NATS is an abbreviation for the neural autonomic transport system, which is a messaging system that enables diverse devices, such as client and server applications, to communicate with each other in real-time using a publish-subscribe pattern (3). Socket.IO is a library that allows real-time, bidirectional, and event-based communication between the client and server (4). FastAPI is a Python-based web framework for developing APIs that is fast and simple to use as its name (5) and offers interactive API docs, formerly known as Swagger UI, which enable users to test their API endpoints from the browser (6). ReactJS is a popular library for building applications' user interfaces (7; 8). Besides, the Virtual-DOM, which enables efficient rendering of the user interface, is one of the characteristics that makes React.JS suited for real-time applications; it selectively renders the components affected by changed real-time data, rather than refreshing the entire page. This leads to improved speed and user experience. (9.)

It should be noted that this project is part of a larger pre-built project, which means that the backend and frontend infrastructure have already been constructed. Specifically, the database, server, and logic for connecting to Manipulator via NATS protocol have been accomplished in the Web_api service. Thus, tasks in Web_api service are to create API endpoints for actions that support displaying radios status and controlling radio position using existing logic. Regarding Socket_io service, it has been initialized and connected to the Manipulator via the NATS protocol. Therefore, the Socket_io service's task is to develop a method for observing new messages from the Manipulator's radios status channels via NATS protocol and for transmitting these messages to the frontend via socket.IO, so that whenever the Socket_io service receives a new message of radios status from the Manipulator, it will then transmit it to frontend concurrently. Similarly, frontend has already set the socket client; and the main job of the frontend is to construct a comprehensive component to display radios status and control radio position in real-time by accessing API endpoints built in the Web_api service and listening channels from the Socket_io service. To take advantage of these pre-built components, it is necessary to first go through the technologies above to understand how they work. The architecture of the project will be discussed in further detail in the Implementation

section. It should be also notable that these tasks are being conducted in a development environment using mock data and mock servers. This way, it will save time verifying whether the entire application works properly with the data provided. Moreover, controlling no real hardware in development mode ensures that no harm is conceivable to certain hardware personnel and testers who are installing and testing in the chamber.

2 TECHNOLOGIES FOR DEVELOPING REAL-TIME WEB APPLICATION

The real-time web application consists of three major components: frontend, backend and protocols used for communication. It requires appropriate technology to handle logic, data transfer from server to client, and update the user interface without reloading the page. This chapter will go over the building pieces of real-time web application technology including NATS and Socket.IO for data transfer, Python-based FastAPI for backend, and React.JS for frontend.

2.1 NATS

NATS is an open-source messaging system that provides scalability and ease of use for distributed systems; it enables servers and clients to transmit and receive messages with low latency and high throughput in real-time (3; 10). NATS protocol typically operates at OSI (Open Systems Inter-connection) layer 7 (11) and on top of layer 4 TCP/IP protocol (12).

NATS protocol is ideal for real-time communication or event-driven systems because it is simple, quick, and scalable. NATS is basic because it employs an easy-to-use text-based format and has only a few commands and messages that cover the most typical scenarios. This protocol is quick since it utilizes a binary payload capable of carrying any data type, and it has low overhead and latency. Because it involves a publish-subscribe pattern that divides publishers and subscribers, the protocol is scalable, and it has features that enable flexible and efficient message routing and delivery. (12; 13; 14.)

2.1.1 NATS Client Protocol Structure

NATS messages have a simple format that allows for easy interpretation and processing. Each message typically consists of three main parts: control line, headers, and payload. The control line identifies the message and reveals the sort of operation or command being done; for example, PUB signifies a message being published to a specific subject. Regarding headers, it is an option to give additional information about the message, such as the encoding type, the subject of the reply, a unique message identifier (message ID) and a timestamp. Concerning payload, it is the message's real content, and it includes data in various formats such as JSON, text, and binary

data. Payload is also optional for most NATS messages, but necessary for others such as PUB, MSG, HPUB, and HMSG. (12; 15.)

A typical NATS message may resemble the following: "SUB news 6" (figure 1). In this instance, the first NATS client, which is created from the command line in the bottom right corner of the screen, sent a message to subscribe to the news subject with the subscription ID 6. To determine if a message was successful or not, NATS includes special Control lines known as status codes, which are +OK and -ERR commonly (12). Then, the NATS server sends status codes to the client, notifying the +OK result to it. After that, in this scenario, some messages were sent by the second NATS client, which was formed using the python file client.py, to publish the string with the content of "news number {i}" on the news subject. At the same time, the earliest NATS client receives a fresh message such as "news number 1" in real-time. Note that the initial NATS client uses the telnet protocol to connect to the NATS server before sending the message, and this message is missing an optional header.

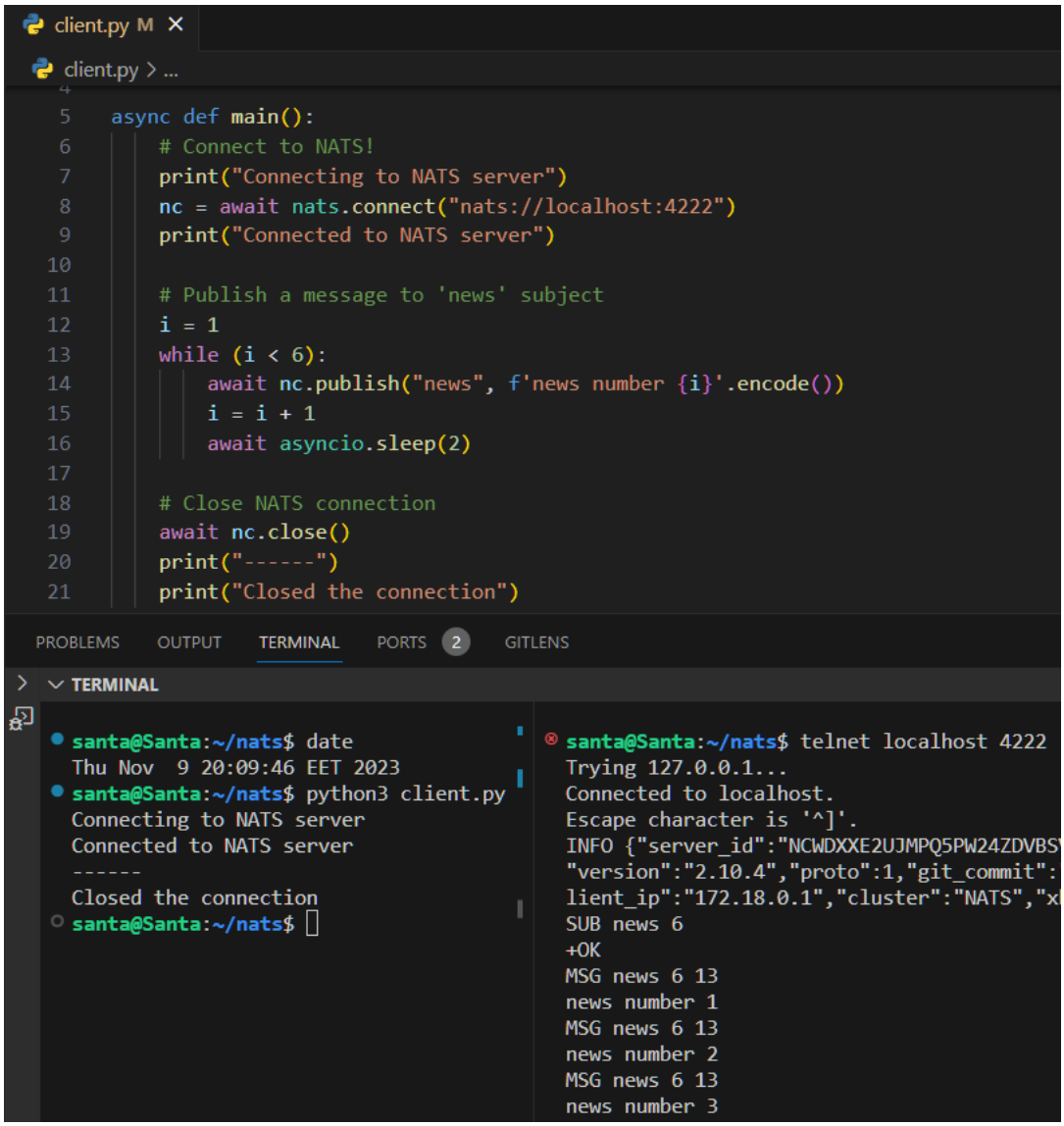


FIGURE 1. An example of NATS messages.

2.1.2 NATS's common features

NATS offers various features such as wildcards and queues. It also includes security capabilities like as authentication, authorization, and encryption using the TLS (Transport Layer Security) protocol to protect communication between servers and clients. (16.)

Wildcards, which are symbols that can match parts of a subject, are one of those features. There are two wildcard characters: * and >. The * wildcard suits any single token, whereas the > wildcard matches every token at the end of the subject. (17.)

Another feature is queues, which are groups of subscribers observing the same subject. When a publisher delivers a message to a topic with queue subscribers, the message is sent to just one subscriber in the queue group. Because the load may be divided across several subscribers, this enables scale and fault tolerance. (18.)

Nevertheless, NATS has a limitation: as a publisher, a NATS server can only deliver messages to NATS clients who are both online and connected to it at the same time. This means that if a NATS client disconnects and then reconnects, any messages sent while it is offline will be lost. JetStream is a feature of NATS that overcomes this problem. It is a streaming layer constructed on top of NATS that adds persistence, retention, and observability to NATS messages. It enables clients to create message streams that can be stored in memory or on disk and consumed by consumers. JetStream offers the temporal separation of publishers and customers. Consumers, in this instance NATS clients, can define numerous settings such as filters, acknowledgement policies, replay policies, and especially, deliver policies. For further information, deliver last, a capability in delivery policies that enables transmitting a consumer the last message of a stream or subject. This is handy for applications that require the ability to handle events in chronological order or synchronize data copies as this allows them to access the most recent state of a data stream without losing track of significant events. (19; 20; 21.)

As shown in Figure 2, for instance, when NATS client 1 is first launched, it establishes a stream called "news" in the code sample and publishes a few messages under the "news" subject (including the text "news number i" where i ranges from 1 to 8. Next, NATS client 2 is launched, and it subscribes to the "news" subject. Setting the delivery policy feature to "DeliverLastPerSubject" often referred to as "last_per_subject", will allow the consumer 2 receive the last message published to the stream with the content "news number 8".

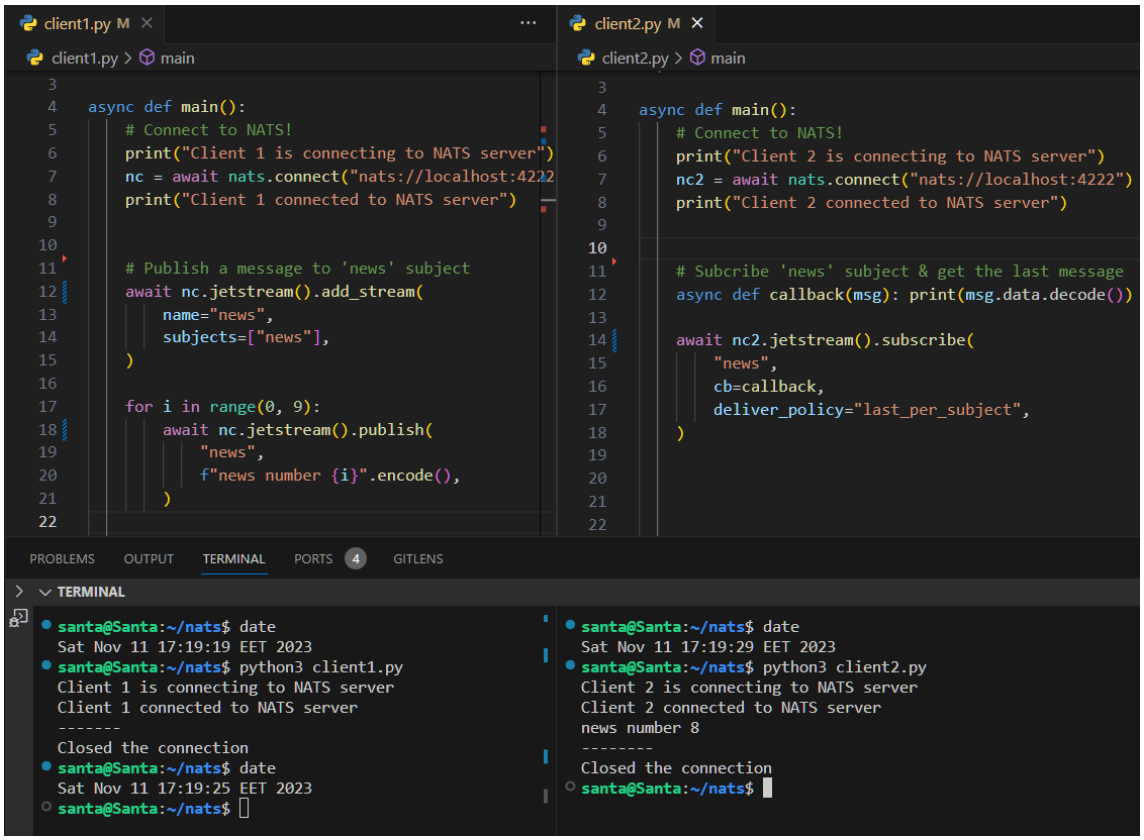


FIGURE 2. An example of how JetStream handles the last message.

Another notable feature of NATS is Monitoring NATS. It is a network monitoring tool that is utilized to monitor the status of the server (figure 3) (22).

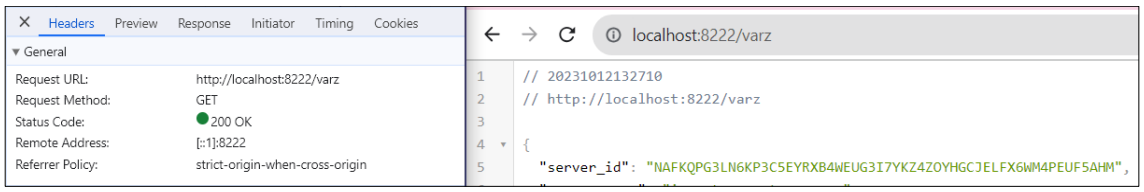


FIGURE 3. NATS server status.

Besides, the traffic between NATS servers and clients can be tracked via the tool above (22). Figure 4 displays an example of the traffic statistics.

FIGURE 4. The number of subscribers connecting to the NATS server.

2.2 Socket.IO

Socket.IO is a library that allows for real-time, bidirectional, and event-based communication between the client, sometimes known as the browser, and the server. Furthermore, Socket.IO has a reliability feature: if the communication is lost, the client will immediately attempt to reconnect. It is comprised of a server-side component and client-side components that can operate in a variety of languages and platforms, including Node.js, Python, and others. (23.)

Before entering the basic working mechanism of Socket.IO, the WebSocket and HTTP polling protocols should be reviewed, as Socket.IO employs both transports to create a connection between the client and the server, depending on the environment's capabilities. WebSocket is the preferred transport; it is a protocol that offers full-duplex, low-latency channel over a single TCP connection (4); it typically operates at layer 7 of the OSI model (24). WebSocket enables the clients and server to transmit and receive messages in real time with little overhead (23).

Nevertheless, if WebSocket is not supported by browsers or network proxies on servers, Socket.IO can fall back to HTTP polling, a method that emulates bidirectional communication by continually sending HTTP requests from the client to the server and getting responses containing data (25). HTTP polling is, of course, less efficient than WebSocket polling because it includes more network traffic and delay.

Due to using HTTP polling and WebSocket, Socket.IO protocol typically operates at OSI layer 7 and on top of layer 4 TCP/IP protocol. It should be noted that, in addition to HTTP polling and WebSocket, Socket.IO version 4 also employs WebTransport, a recent update to WebSocket that can transmit payload between the clients and server via HTTP/3; however, it will not be within the scope of the thesis because this project does not use HTTP/3. (25; 26.)

Socket.IO wraps the base transport's details and provides a simple and uniform API for both the client and the server. The API is constructed around the concept of events, which are messages with a name, sometimes known as a title, and optional data. The socket object allows the clients and server to interact by emitting and listening to these events. For instance, the client can send a 'news' event to the server with the data "I want to read discovery topic", and the server can listen to that event and reply appropriately; similarly, the server could send an event named "today" with some data to the client like as "Around the world", which the client can listen to and show in the browser (figure 5). In addition, the socket object emits various specified events that show the state of the connection, such as 'connect', 'disconnect', 'error', and so on. (27; 28.)

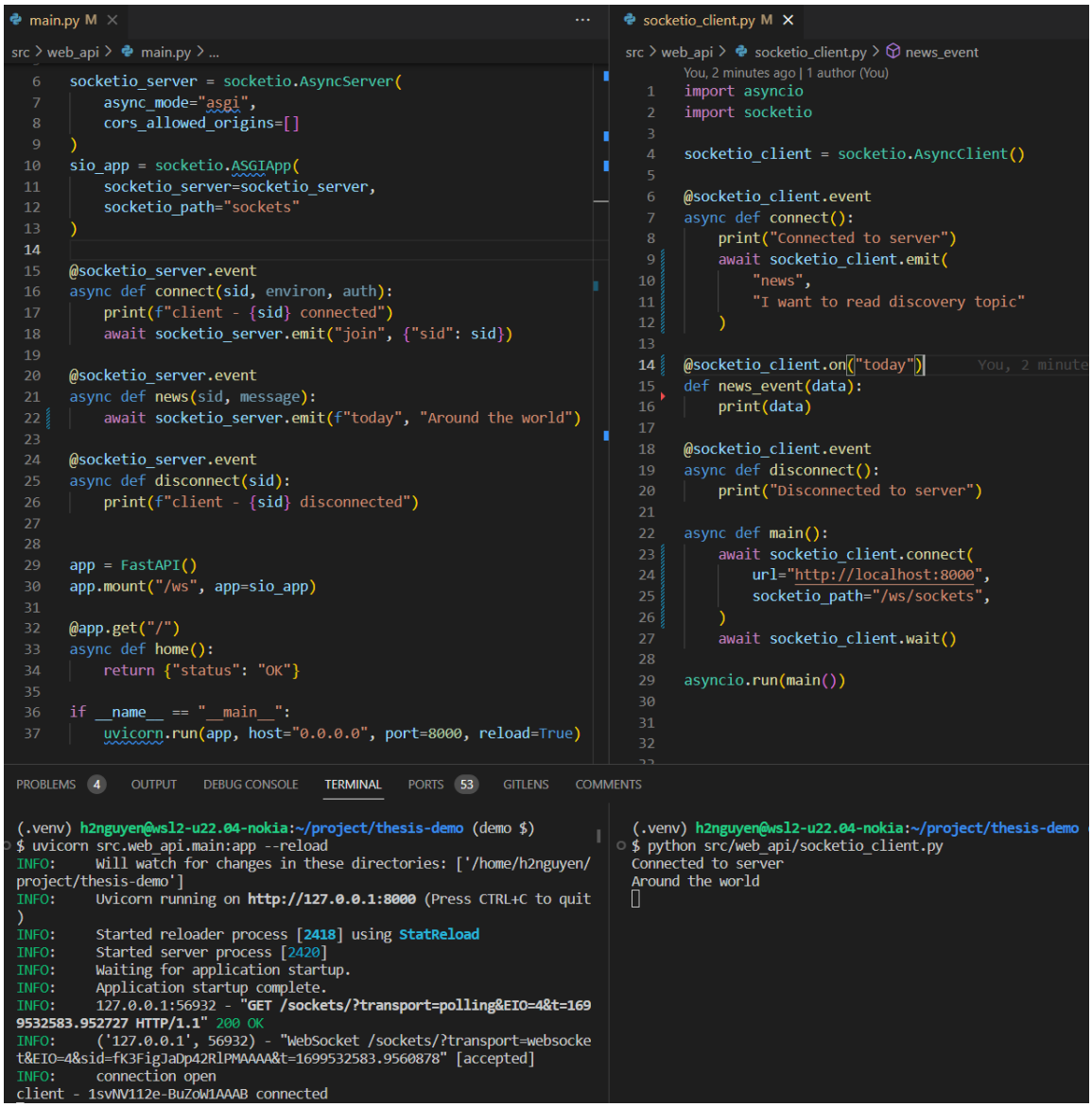


FIGURE 5. An example of interacting between server and client using Socket.IO.

Besides, the asyncio library in the code example on the right side of the above image is something to investigate. Users can write asynchronous code, or asynchronous programming, thanks to features like coroutines and async/await syntax that are built into this Python library. Asynchronous programming helps reduce the amount of time needed for processing by enabling an application to handle multiple tasks at once without blocking the main thread. This way, the program can continue processing while waiting for these operations to finish. Therefore, asyncio library is helpful for I/O-related tasks (input/output) as well as Socket.IO server and client, which can handle multiple connections and events without blocking the main thread. Regarding coroutines, they are functions that could pause and resume execution, coroutines are marked and called using async/await syntax. However, asynchronous programming can also be more difficult than typical linear one since it requires a different approach to generating code. (29; 30; 31.)

2.3 FastAPI - a Python framework

FastAPI is a contemporary and elegant web framework that uses common Python type hints to simplify APIs development using Python 3.8+. FastAPI was initially released in 2018 by Sebastián Ramírez. It is quick, simple, and robust, with numerous features that make API development more productive and interesting. FastAPI is utilized by many enterprises and organizations including Netflix, Uber, Microsoft, and Cisco. (5.)

2.3.1 The Architecture, Dependencies and Packages of FastAPI

Unlike some other web frameworks, FastAPI does not impose any architecture or design pattern for developing web apps. Rather, it is compatible with a variety of methods, including MVC (Model View Controller), MVP (Model View Presenter), and MVVM (Model View-View Model). The goal of these models is to uncouple the concerns of application logic, the user interface, and data access. (32). FastAPI offers the tools and capabilities needed to implement any of these patterns, including dependency injection, path operations, routers, templates, and so on. (33; 34.)

It should be mentioned that FastAPI has available practice structures for users to use (34); and it also works with any database and any type of library to communicate with the database, thanks to a variety of data manipulation tools (35), but this thesis does not focus on databases due to its scope.

FastAPI is built on Pydantic and Starlette, two Python libraries. Pydantic enables data validation as well as serialize/deserialize (36). Meanwhile, Starlette using the ASGI (Asynchronous Server Gateway Interface) convention offers web server capability and supports it to handle asynchronous requests and WebSocket (37). It is very quick and effective because it can handle several requests concurrently without blocking (38). Features offered by Starlette include testing, background tasks, middleware, routing, and more (37). Because FastAPI is developed on top of Starlette, it inherits all of Starlette's features and integrates multiple useful packages to provide capabilities like dependency injection, Celery for background tasks, testing, security, and OAuth2 authentication (39). From a standards-based standpoint, FastAPI declares the parameters and responses of the API endpoints using standard Python type hints, and they are automatically transformed into OpenAPI (formerly known as Swagger) and JSON Schema documentation. (5.)

2.3.2 FastAPI's common features

FastAPI supports the notion of lifespan events, which are code blocks that run before and after the application starts and stops. These events can be employed to configure and deactivate resources required by the entire application, such as database connections, machine learning models, and so on. FastAPI handles these events using the ASGI specification, which is established by using the asynccontextmanager decorator to start up and/or shutdown events (another option is using a deprecated on_event decorator). As an example, a database and its connection could be built upon application launch, and then the connection can be closed upon application shutdown. (40.)

Besides, FastAPI includes a consistent and simple method of managing requests and responses. Each request is a Python function known as the path operation function, which defines an endpoint for the API. This function is decorated with a specified HTTP method such as @app.get and a path like "/users/{user_id}" (figure 6). After that, it will validate the request data, inject dependencies, and run the function logic. For further control, this function can return any JSON object or a response object. FastAPI will then typically transform the return value to a JSON response and return it to the client. (41.)

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/users/{user_id}")
def get_user(user_id: str):
    return {"user_id": user_id, "user_name": "Henry"}      You, 1 second ago •
```

FIGURE 6. A get function example.

FastAPI elegantly handles errors and exceptions by utilizing the HTTPException class, which is a typical Python exception. Besides, the default exception handlers can be overridden or customized for better adaptability. For instance, an HTTPException may be thrown with a specified status code like 404 (for "not found") and a message like "User not found" (figure 7). It then checks to see whether the user_id exists in the list of users. If not, an HTTPException with the status code 394 and the message "User not found" is thrown. After catching the exception, FastAPI will return a JSON response with the same status code and message. If the user_id is found in the list of users, the user is returned as a JSON response with the status code 239 by default. (42.)

```python
from fastapi import FastAPI, HTTPException

app = FastAPI()

users = {"1": "Henry", "2": "Marry"}

@app.get("/users/{user_id}")
async def get_user(user_id: str):
    if user_id not in users:
        raise HTTPException(status_code=404, detail="User not found")
    return {f"user - {user_id}": users[user_id]}      You, 9 seconds ago •
```

FIGURE 7. An example of utilizing HTTPException to handle errors and exceptions.

In terms of the capacity to generate interactive documentation automatically using Swagger UI, it is a web interface for exploring, testing and documenting API endpoints. It displays the arguments, responses, and examples for each endpoint and allows us to test the API queries in the browser. Swagger UI is also automatically updated based on the code, ensuring that it always represents the lasted API version. (6; 43). The following figure shows an example of interactive API documentation, usually known as Swagger UI.

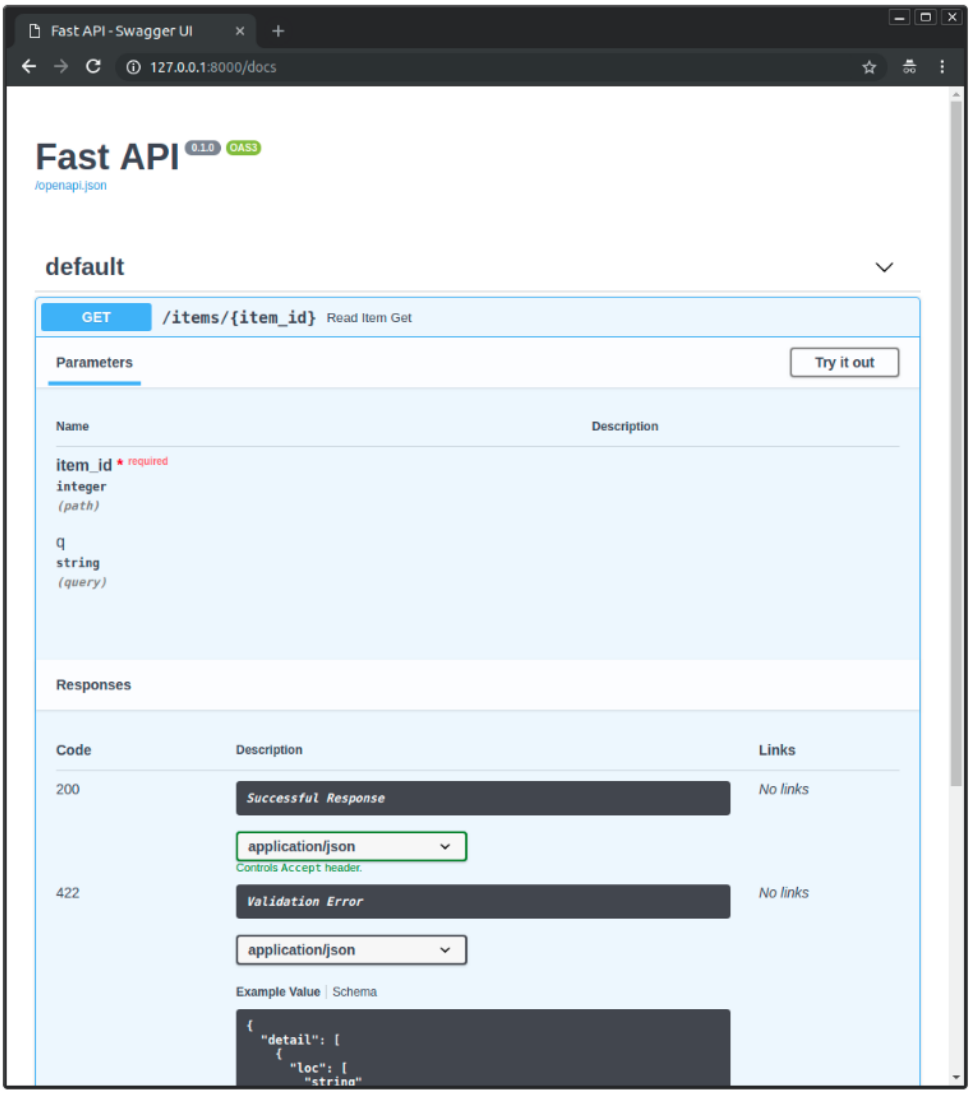


FIGURE 8. Example of interactive API documentation (6).

Another feature of FastAPI is data validation using Pydantic library (44). FastAPI utilizing both to check data from path, query, and body parameters as well as to create the JSON schema for the API documentation (45). It also gracefully handles failures and exceptions and returns valuable information to clients.

Pydantic is a library that uses type hints to verify, serialize, and deserialize data into Python objects and data structures. It can cope with sophisticated data types including integers, texts, dates, lists, dictionaries, and nested objects. Besides, it provides custom validators which can be applied to certain fields or the entire model. (44; 46; 47). As an example, a Pydantic model can be defined for a new_user_info and used as a type of hint for a body parameter; it is validated by the available constraints in Field (48), another way is employing the validator decorator (49) (figure 9).

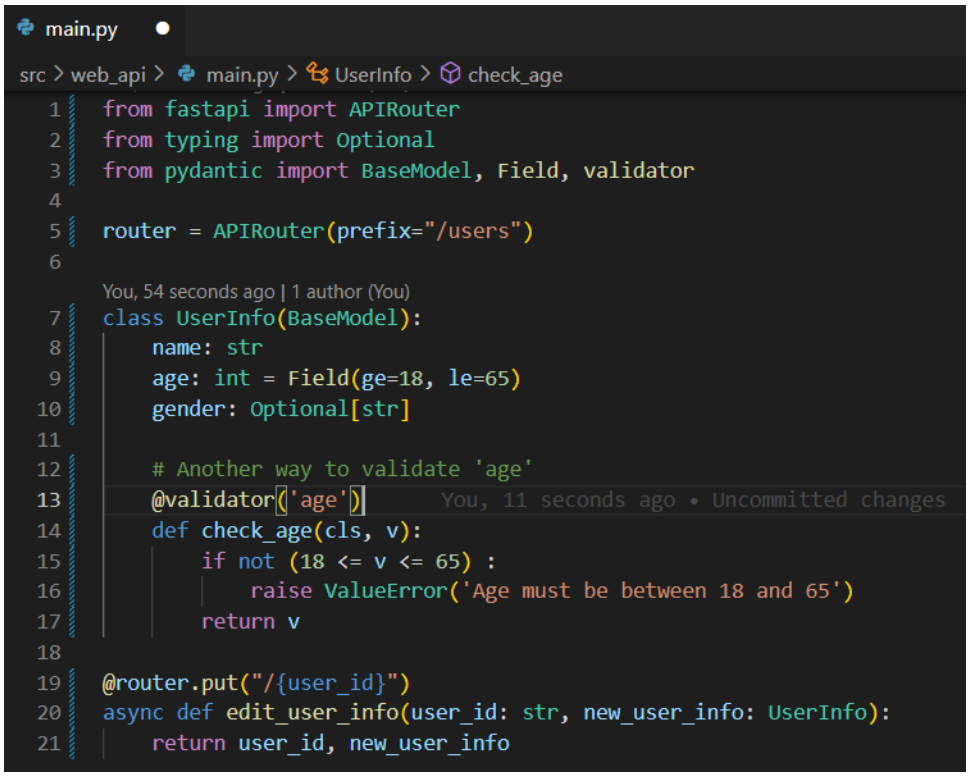


FIGURE 9. Example of using Pydantic.

2.3.3 FastAPI Overview and Challenges

In general, as previously mentioned, FastAPI is a quick, easy, and reliable framework for building APIs in Python. It offers a simple and expressive syntax that adheres to Python standards, and it makes use of Pydantic and Starlette to provide data validation and web server functionalities. (5.)

It does, however, have certain restrictions at the time being. For starters, Python 3.8 or higher is required, which may not be compatible with some legacy systems or environments. Second, it is still very young and may lack the support and reliability as older frameworks such as Django or Flask. Finally, it may not be appropriate for some complicated or specialized applications that demand greater flexibility or customisation than FastAPI provides. (50; 51.)

2.4 React.JS

As indicated in the introductory section, React.JS is one of the most common and frequently used front-end libraries in the web development. React.JS enables developers to generate reusable components capable of rendering dynamic data and interacting with users. State, context, and

custom hooks are also supported by React.JS, allowing state management and data transmission among components (52; 53).

ReactJS applies a syntax extension named JSX that enables developers to write HTML-like code in JavaScript. JSX simplifies the creation and manipulation of elements, as well as the passing of props to components. By using class or function components, ReactJS supports both object-oriented and functional component structuring. The figure below illustrates a basic parent component, which consists of a render function that returns the JSX code that appears on the screen, as well as how to pass 'Parent 1' props from parent to child components. (54; 55; 56.)

FIGURE 10. JSX component example.

Regarding state management, it is a method of organizing and supervising the state of components, which can alter over time due to data changes; and then determining how components appear and interact on UI, in other words, how to manipulate DOM (Document Object Model) without reloading the entire page. React.JS provides several state management approaches such as useState hook, useReducer hook, and the pair of createContext and useContext hook, as well as external libraries like Redux. (53; 57.)

Another feature is component lifecycle management using the useEffect hook. The component's lifecycle is a chain of stages that it goes through from construction to destruction, including mounting, updating, and unmounting (figure 11). When a component is mounted or generated, it is inserted into the DOM; when it is updated by using useEffect, it is re-rendered; and when it is unmounted, it is removed from the DOM. (58; 59; 60.)

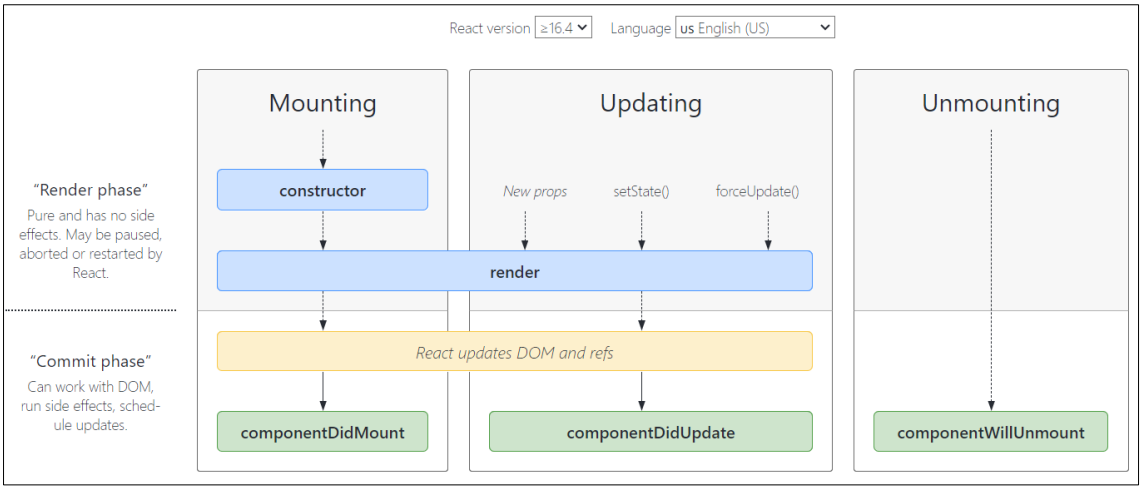


FIGURE 11. Component lifecycle diagram (60).

One of reasons for using useEffect is that browsers execute React.JS code and render the components on the main thread, which is also in charge of handling user interactions and updating the DOM. This means that any long-running or blocking operation on the main thread can degrade app performance, resulting in an unsatisfactory user experience. As a result, it is required a way to perform some operations outside of the main thread without affecting the UI in the mounting stage. These operations are known as side effects, and they are typically placed within the useEfect hook or the updating stage. Side effects are actions that affect or are dependent on something that is not within the scope of the function. (61; 62.)

The figure below demonstrates the way a React app handles real-time state changes by utilizing state management, useEffect with side effect, and lifecycle (figure 12), in the mounting stage, the browser displays "The visitor number:" text because the data value is null. Following that, in the updating stage, useEffect allows the fetchVisitorNumber function, a side effect in the component, to run. This side effect function takes 6 seconds to obtain a new number and then updates the component's new number state. The component is re-rendered because of the state change, and the browser now displays "The visitor number: 1" text. useEffect executes after every rendering by default unless the dependencies are specified. The dependencies are values of variables that side effects depend on, such as props and state. In this case, the dependency is an empty array, which means the side effect will only be executed once after the initial render. If the dependencies array is not empty, the side effect will run after the initial render and whenever any of the dependency's changes. And if no dependency array is passed, the side effect will be called after every render and re-render. useEffect can also return a cleanup function, which is executed before the component is unmounted to reduce memory leaks. (59.)

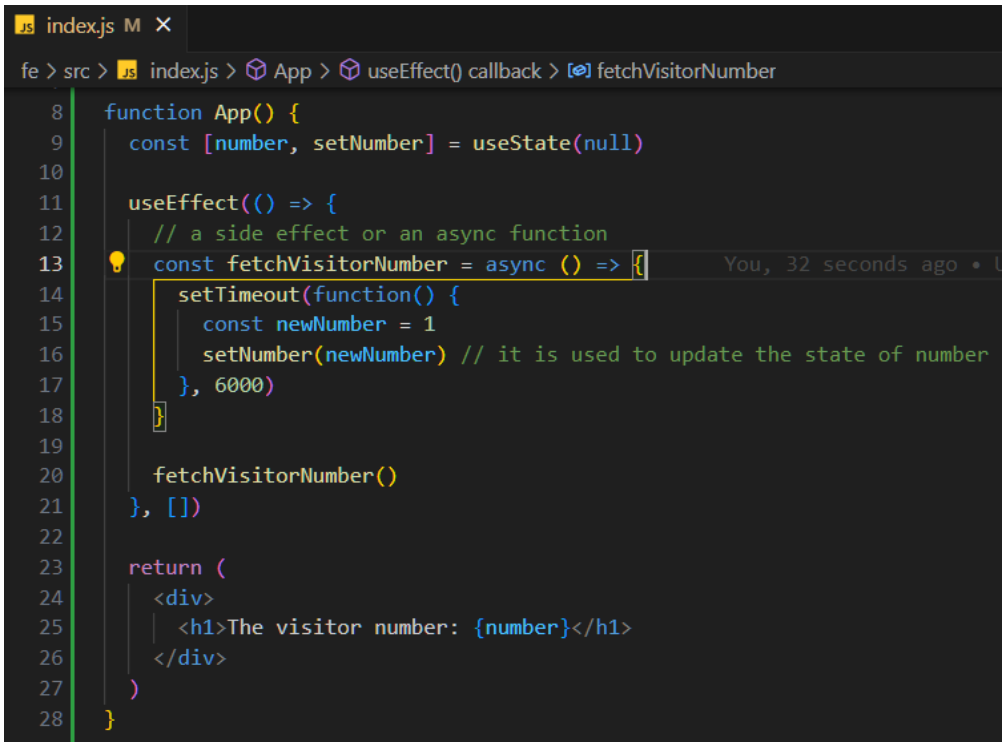


FIGURE 12. An example of handling real-time state changes.

The image below describes how the React app handles real-time state changes by utilizing state management, useEffect, and lifecycle (figure 13), the Counter component uses the useState to set the count state to 0 and display this initial state on screen for the first time during the mounting stage (this can be seen in the developer toolbox's logs as "render for the: 0th time"). Then in the updating stage, useEffect is used by the component to invoke the countUpInterval function, which updates the count state by adding 1 every second. This manner, apps only need to re-render the h2 tag associated with the count state efficiently and seamlessly. Furthermore, when the component is destroyed, useEffect returns a cleaning function that uses the clearInterval function to clear the countUpInterval.

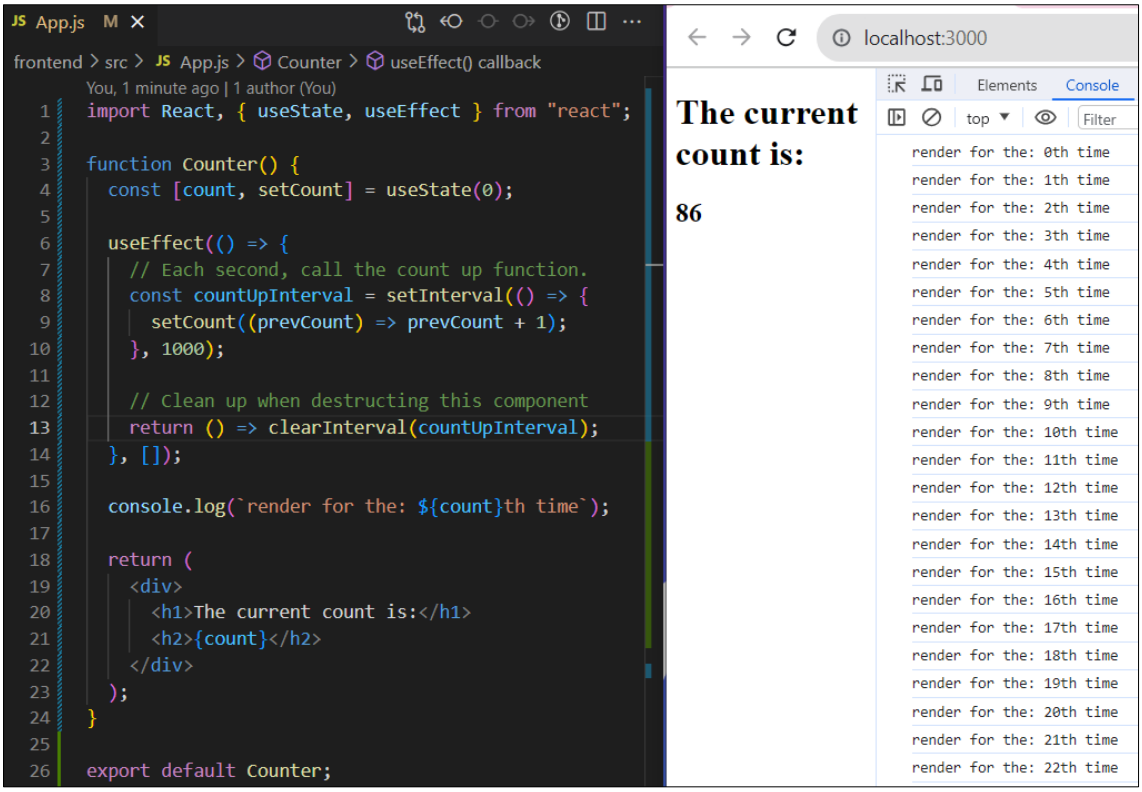


FIGURE 13. A state management and lifecycle example.

In addition, React.JS can be extended to support socket communication by installing a package named socket.io-client. This package provides a high-level API for components to generate, control socket connections to socket servers, as well as to observe and/or emit events' new messages (63). This library is an example of a third-party library that can be used to extend the functionality of React apps. Apps can make use of a variety of libraries available from sources such as npm, a JavaScript package manager (64). These libraries can provide features or solutions that React does not include, such as socket connections, which can be installed with "npm install socket.io-client" command in this case.

React.JS can efficiently display and update real-time data from the Socket.IO server by employing socket.io-client dependency, lifecycle and state management. As an instance, the React app (figure 14) uses socket.io-client library to generate a socket connection to Socket.IO server. The useState hook is utilized to update and display states such as count, isConnected on the screen. Meanwhile, useEffect is employed to listen to "start" and "stop socket connection" events as well as the "count" event via socket.on method; and if the socket client catches any new message from the socket server in, it then will update and show the new state on the screen in real-time. It should be noted

that, in practice, the socket connection and/or its listening events need to be cleaned up when the component's lifecycle ends.

FIGURE 14. Example of applying React's features in real-time application.

3 IMPLEMENTATION

The developed application contains two useful features which are radios status browsing and radio position manipulation. It allows the browser to display radio status in real time so that the users or tester can have a clear and accurate view of the radio network testing status. Figure 15 displays an example of both actions above. First, it shows AAFIA radio status view in Chamber number 1. There, real-time values of x and y positions, which are represented for the radio's coordinator including horizontal and vertical coordinates, are 14.7 and 30.0. Second, users can also alter the AAFIA radio's coordinator by entering the input fields or boxes, and then pressing the start button to implement the radio position change. In this figure, for example, x and y positions are required for changing position, whereas x and y speed are optional. In practice, the app also can be used to manage multiple chambers concurrently, including multiple radios, by reusing the compact Chamber component below and handling some additional socket io logic in the backend. However, the scope of this thesis will only cover the anatomy of controlling the radio component in real time. Besides, the development workflow will be detailed more explicitly in the architectural section, containing components and a sequence diagram, step by step.

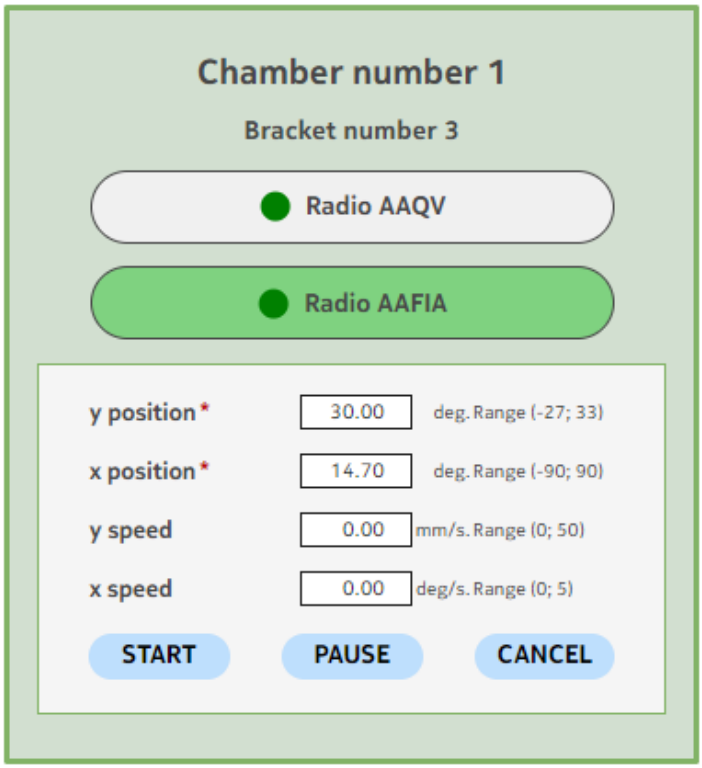


FIGURE 15. An example of the application's UI.

3.1 Architecture

The component diagram below (figure 16) illustrates the project's five components and their connections. First, the Manipulator Chamber serves as the controller for the real-world chamber, as well as the NATS client for receiving and sending messages to the NATS server. Second, NATS server is used to connect and transmit bidirectional messages to NATS clients including NATS pub (NATS client whose primary task is to publish messages in specified subjects in real-time) and NATS sub (NATS client whose primary task is to subscribe specified subjects to receive messages from these subjects in real-time). NATS server can also be used to store published messages. Third, Socket_io service offers a real-time communication protocol, this service works as a NATS sub to monitor messages from the NATS server, and then it acts as a Socket io server and is used to emit messages to Frontend. Fourth, Frontend is a user interface that allows testers to visually browse radio status and alter radio position. Finally, Web_api service provides REST API so that the Frontend can manage the radio position and obtain the reserved status of the Chamber, as detailed in section 3.2.1.

It should also be noted that, as mentioned in the introductory section, because this project is part of a larger project, these components have already been built into the system, especially NATS server and Manipulator Chamber were entirely built. As a result, it is not required to create a completely new component from scratch. Before employing these available components, it is necessary to research and comprehend the present structure to enable two primary features such as "radios status browsing" and "radio position manipulation". Then, in the Backend servers including Web_api service and Socket_io service, additional functions will be added, as well as new components in Frontend UI.

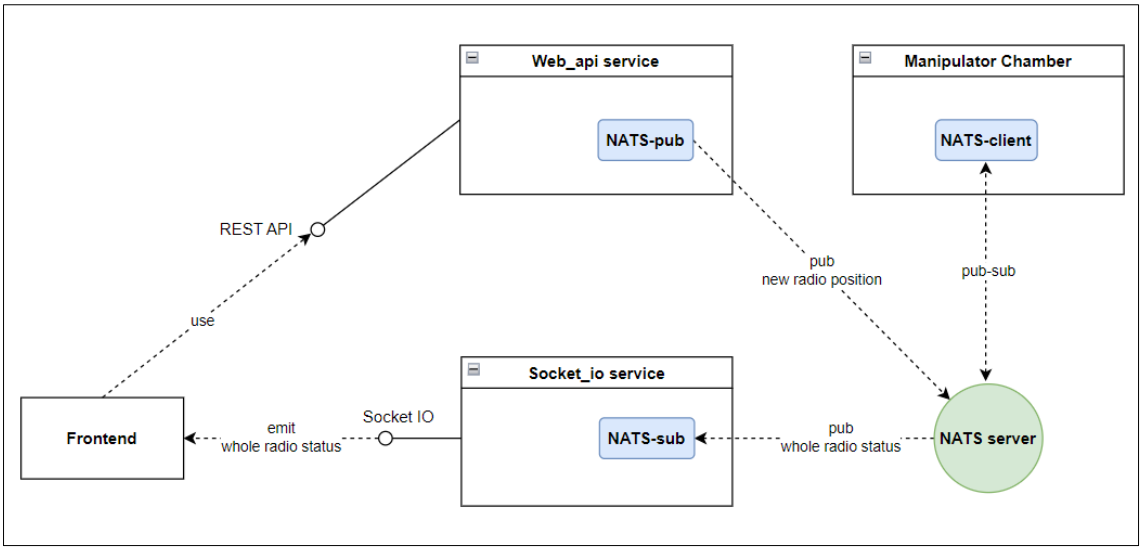


FIGURE 16. Component diagram.

Furthermore, the sequence diagram following (figure 17) demonstrates two main features in this project, including "radios status browsing" and "radio position control". Regarding the "radios status browsing" feature, when the user opens the browser, the first thing that must be done is to identify the chamber that the user has reserved by requesting the API endpoint from Web_api service. After obtaining information about a specific chamber, the browser connects to Socket_io service to gather radios status in real time belonging to this chamber by listening to radios status events. Concurrently, the Socket_io service is aware that it must subscribe to specified radios status subjects from the NATS server to obtain and then provide radios data in real-time to the browser. It should be noted that before users access the app, pre-built servers have already booted up and connected to one another.

Concerning "radio position control" action, the browser calls an API endpoint on the Web_api service to change the radio position. This invokes a function that validates the input data and publishes it to the NATS server, which then sends it to the Manipulator Chamber to control the radio position in the real world via the NATS protocol. Concurrently, the browser can update the radio position in real time since it is already listening to these radio events via a socket connection.

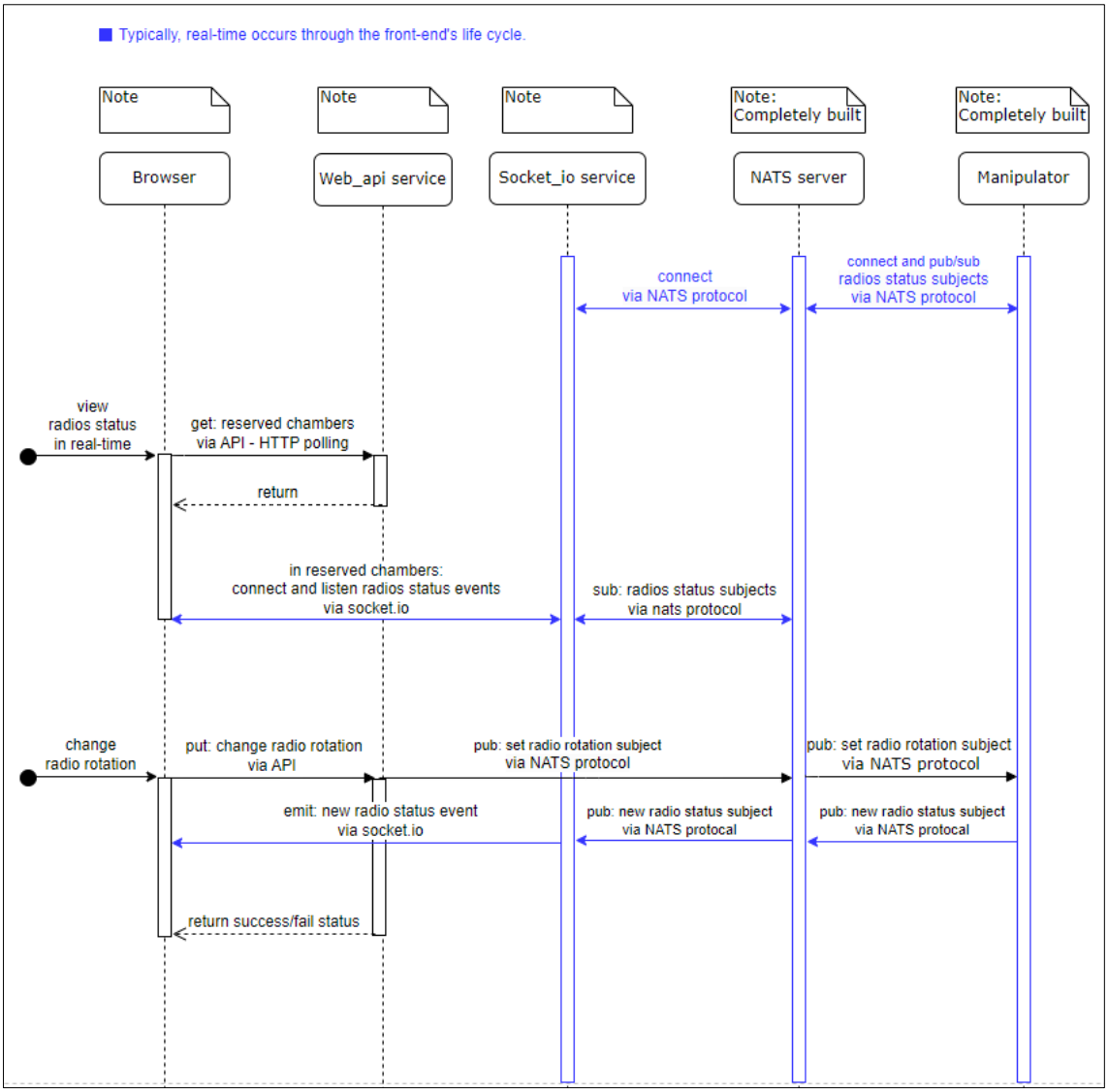


FIGURE 17. Sequence diagram.

3.2 Backend

As described in the Architecture section, the pre-built Backend servers include Web_api service and Socket_io service. They require the addition of functions to provide two key features "radios status browsing" and "radio position control". Regarding the pre-built Web_api using the FastAPI framework and NATS protocol, two new routes or API endpoints "chambers_reservation_status" and "manipulator_by_coordinate" need to be added. First, "chambers_reservation_status" route is generated to obtain a list of channels and their reservation status. Then, only radios in the reserved chamber will have their status updated in real-time by the Socket_io service. Second, "manipulator_by_coordinate" route for controlling radio position is created. Concerning the pre-built Socket_io

service, an additional method to subscribe radios status sub-jects from the NATS server using NATS protocol and then emit radios status events to the end clients using Socket io protocol is required.

3.2.1 Web_api service

Returning to the "view radios status" feature, the first step is to develop an API endpoint for providing chambers' reservation status to the frontend. Figure 18 shows how to create an API endpoint function named get_chambers_reservation_status in the pre-built FastAPI-based Web_api service using the GET method.

```
9   from fastapi import APIRouter
10
11  router = APIRouter(prefix="/radios_manipulator")
12
13  @router.get("/chambers_reservation_status")
14  async def get_chambers_reservation_status():
15      manipulator_chambers = ManipulatorChamber()
16      return manipulator_chambers["chambers"]["chambers_reservation_status"]
```

FIGURE 18. An example of providing an API endpoint via the GET method.

The figure below represents an interactive API documentation including the API resource mentioned above. This service endpoint displays the yield of get_chambers_reservation_status method which displays HTTP response status codes of 200 as well as a dictionary of chambers, with chamber number one currently reserved for radio testing.

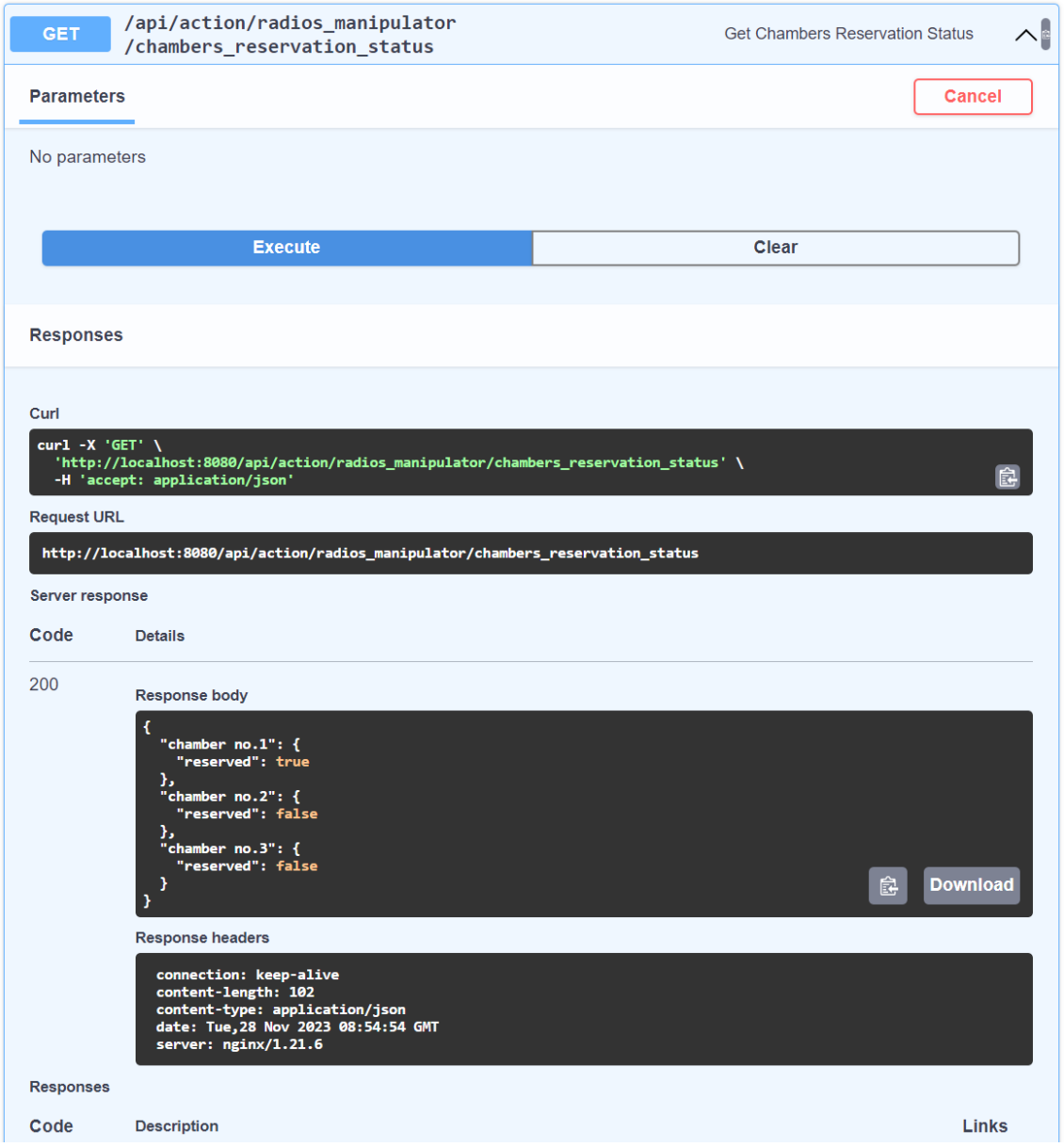


FIGURE 19. An example of using Swagger UI to manipulate the API - GET method.

To provide for the "radio position control" feature, an API PUT method is required. Figure 20 presents how to do it for set_manipulator_by_coordinate function. In terms of input data validation in this function, the chamber_name value must be in the list of chambersName, and the radio_id value must be a string. Furthermore, the payload value is ManipulatorPosition class type using Pydantic library to validate the client's body message, such as xAxis value must be a floating point number with a range of 0 to 99. This function also generates a NATS client to send a "radio position" subject with a new payload to the NATS server. The manner for creating a NATS client and publishing messages to a NATS server is already covered in the NATS theory section.

```python
from fastapi import APIRouter
from typing import Literal
from pydantic import BaseModel, Field


router = APIRouter(prefix="/radios_manipulator")


You, 1 second ago | 1 author (You)
class ManipulatorPosition(BaseModel):
    xAxis: float = Field(ge=0, le=99)
    yAxis: float = Field(ge=0, le=99)
    xSpeed: float
    ySpeed: float


@router.put("/manipulator_by_coordinate/{chamber_name}/{radio_id}")
async def set_manipulator_by_coordinate(
    chamber_name: Literal["chamber no.1", "chamber no.2", "chamber no.3"],
    radio_id: str,
    payload: ManipulatorPosition
):
    try:
        # Create Manipulator master object using NATS protocol to connect to NATS server
        async def create_manipulator_master():
            # Create a NATS client using NATS protocol inside create_nats_client method
            nats_client = await create_nats_client()
            return Manipulator(nats_client)

        manipulator_master = await create_manipulator_master()

        # Send "change radios rotation" subject to NATS server using NATS protocol
        return await manipulator_master.set_manipulator_by_coordinate(chamber_name, radio_id, payload)
    except Exception as e:
        err_text = str(e) if str(e) != "" else repr(e)
        logger.error(err_text)
        raise
```

FIGURE 20. API PUT method using validation feature and NATS client example.

Figure 21 displays a Swagger UI for the PUT method described above, with three main input fields: chamber_name, radio_id, and payload. Once these fields are valid, it will return HTTP response status codes of 200 and a dictionary of action {radio_rotation: "done"}. It should be recalled that while the set_manipulator_by_coordinate function is running, the browser receives real-time radio status updates via the Socket_io service.

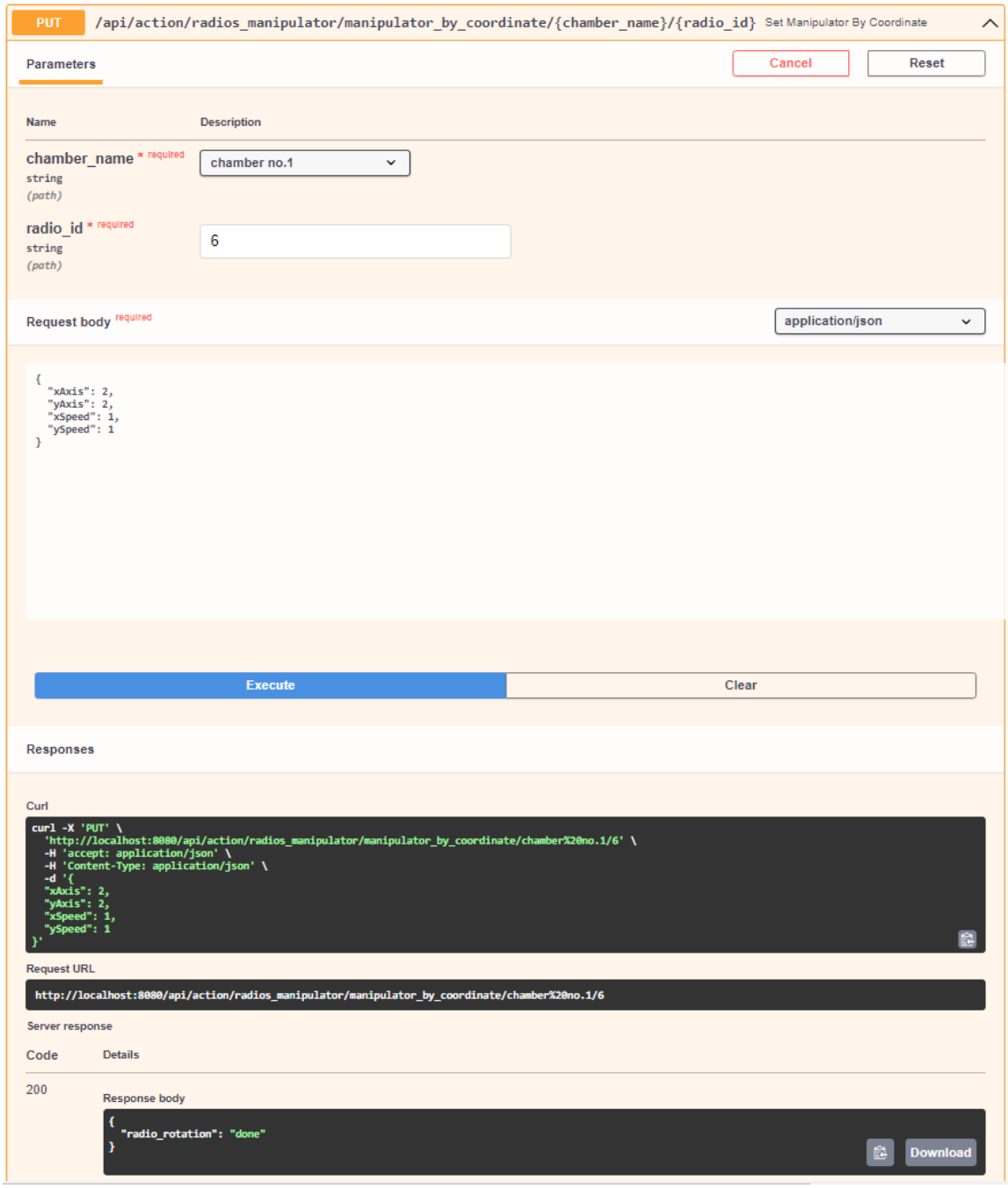
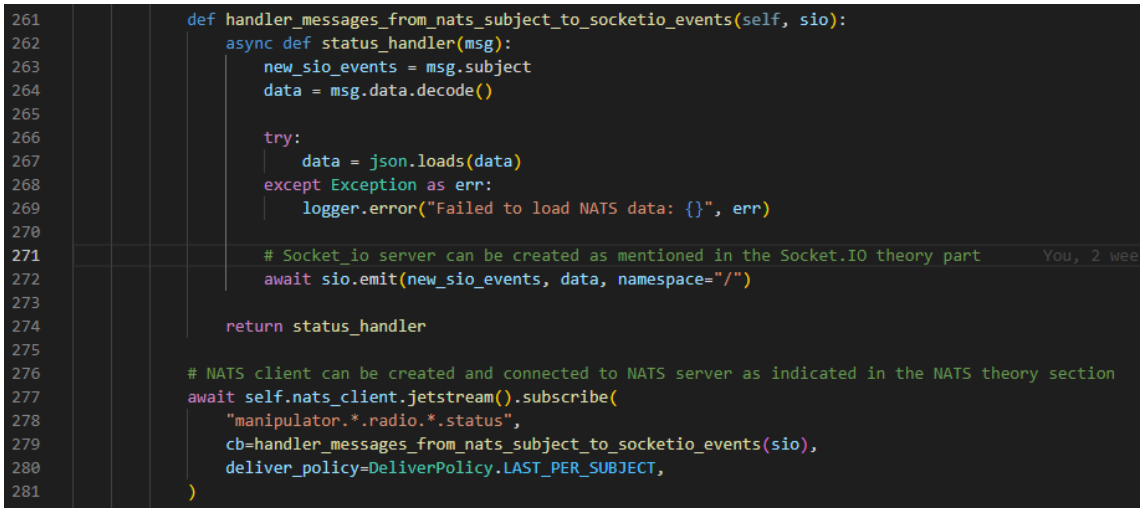


FIGURE 21. An example of using Swagger UI to interact with the API - PUT method.

3.2.2 Socket_io service

The built-in Socket_io service is used to create a new method for providing the “real-time radios status browsing” feature through NATS and Socket io protocol. This service works as a NATS sub to observe new radios status messages from the NATS server, then it acts as a Socket io server to emit these messages to Frontend.

Figure 22 describes how to use NATS - Jetstream and socket emit in Socket_io service. When the pre-built Socket_io service based on the socketio library runs, a NATS client is established and communicated to the NATS server, and it subscribes to "radios status" subjects from the NATS server in real-time using Jetstream to retrieve the lastest message. Whenever it receives a new message of "radios status" subjects, it immediately emits it in "radios status" events so that browsers listening to them can obtain it concurrently. The use of the wildcard * is noteworthy. By using "manipulator.*.radio.*.status" NATS client will subscribe to all status subjects of radios that need to be subscribed with just one line of code. It simplifies the code because developers don't have to subscribe to each radio, such as "manipulator.*.radio.1.status", "manipulator.*.radio.2.status", and so on, or create a loop to subscribe to these subjects.



```
261         def handler_messages_from_nats_subject_to_socketio_events(self, sio):
262             async def status_handler(msg):
263                 new_sio_events = msg.subject
264                 data = msg.data.decode()
265 
266                 try:
267                     data = json.loads(data)
268                 except Exception as err:
269                     logger.error("Failed to load NATS data: {}", err)
270 
271                 # Socket_io server can be created as mentioned in the Socket.IO theory part      You, 2 wee
272                 await sio.emit(new_sio_events, data, namespace="/")
273 
274             return status_handler
275 
276         # NATS client can be created and connected to NATS server as indicated in the NATS theory section
277         await self.nats_client.jetstream().subscribe(
278             "manipulator.*.radio.*.status",
279             cb=handler_messages_from_nats_subject_to_socketio_events(sio),
280             deliver_policy=DeliverPolicy.LAST_PER_SUBJECT,
281         )
```

FIGURE 22. An example of using NATS Jetstream and socket IO emit in Socket_io service.

3.3 Frontend UI features

The pre-made React.JS-based Frontend UI is used to develop some components, which is useful for supplying two key features: "real-time radio status browsing" and "radio position control". Particularly, in figure 15, the frontend UI consists of two main components. The first component is the ChamberView component, which indicates that a chamber can contain two radios named AAQV and AAFIA. Second, RadioForm component is the child of Radios component, which is discussed in the next paragraph. In this case, RadioForm is AAFIA radio view, which displays radio status in real-time with x and y coordinates (horizontal and vertical axes) in the real-world chamber. This RadioForm includes several input fields where the user may enter their desired data before pressing the Start button to implement the changing radio's position. The app can also be utilized for

other components that require a radio position control view and/or a chamber view, as well as for managing multiple radios in several chambers concurrently.

Regarding Radios component, which is the main child of the ChamberView component, displays radios in Chamber number 1 in figure 15. However, because its primary function is CSS, which stands for cascading style sheets, Radios component's UI can be discussed briefly in this section. The Radios component in the figure below, image left side, has two active radios named AAQV and AAFIA in button style. When the user selects "Radio AAFIA" the Radios component will render RadioForm of AAFIA to display real-time radio position and control its position, as seen in the right image. Simultaneously, the colour of the Chamber component and the Radios component is changed to green.

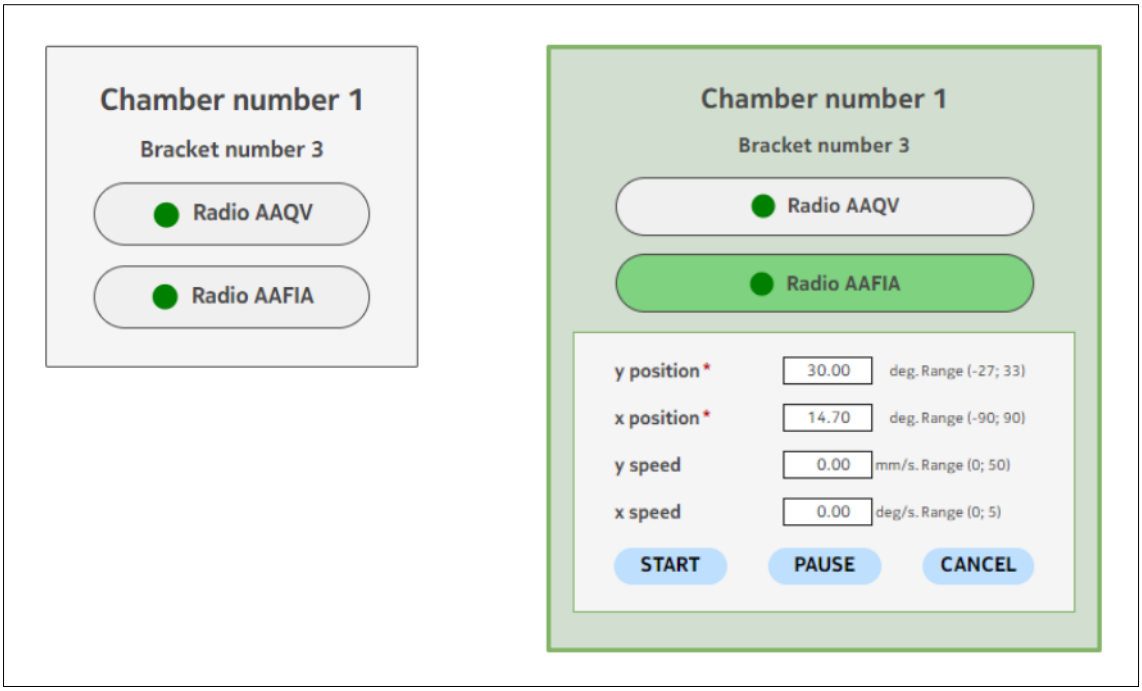


FIGURE 23. A styling instance of the Radios component.

3.3.1 Browsing real-time radios status

The first step in implementing the radios status browsing functionality is to construct a Chamber-View component and route for it in the pre-built React.JS-based Frontend UI. To support this job, the React Router DOM library is pre-installed in this Frontend project with the "npm install react-router-dom" command (65). The image beneath describes using the React Router DOM example.

```jsx
import React from 'react';
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';

export default function App() {
  return (
    <Router>
      <Routes>
        <Route exact path="/chamber" element={<Chamber />} />
```

FIGURE 24. An instance of utilizing React Router DOM library *for ChamberView component.*

To access chambers reservation data, a library called Axios was already installed in Frontend UI to call "/radios_manipulator/chambers_reservation_status" API from Web_api service. Axios is a widely used library for implementing HTTP communication in React development (66). The screenshot following demonstrates an example of utilizing Axios to send an HTTP request in ChamberView component.

```jsx
import axios from 'axios';

export default function Chamber() {
  const [chambersReservationStatus, setChambersReservationStatus] = useState(null);

  useEffect(() => {
    axios
      .get(`/api/action/radios_manipulator/sectors_reservation_status`)
      .then((res) => {
        setChambersReservationStatus(res.data);
      })
      .catch((err) => {
        console.error(err);
      });
  }, []);
```

FIGURE 25. An example of ChamberView component's sending a HTTP request - GET method.

The ChamberView component then uses the data above to filter out radios belonging to the chamber that have been reserved by the user. After that, RadioForm component inside the ChamberView component listens to these filtered radios status events using the socket.io-client library; it should be noted that the socket IO connection was already established in the main App component prior to this step. The image below describes an instance of listening to the radio status events using socket IO protocol. It is necessary to recall that after unmounting this component or quitting the ChamberView page, the socket.off method needs to be executed.

```
const [radios, setRadios] = useState({});

useEffect(() => {
  if (socket) {
    socket.on(`manipulator.${reservedChamberName}.radio.1.status`, (data) =>
      setRadios((radios) => ({
        ...radios,
        radios1: radios.map((radio) =>
          radio.radioId === 1 ? { ...radio, status: data } : radio
        ),
      }))
    );
  }

  return () => {
    if (socket) {
      socket.off(`manipulator.${reservedChamberName}.radio.1.status`);
    }
  };
}, [socket]);
```

FIGURE 26. An instance of listening to radios status events using socket.io-client library.

After combining useState, useEffect and socket.io-client library, as demonstrated in the image above, Frontend UI now could render radios status data in real-time. And RadioForm component uses a built-in React.JS input field, which allows the component to render various types of form inputs, to display these real-time data for each radio on-screen (figure 27) (67).

```
// The value of radio variable is retrieved from the radios list, which is obtained via socket io.
const [radioStatus, setRadioStatus] = useState({
  id: radio.id,
  name: radio.name,
  xPosition: radio.xPosition,
  yPosition: radio.yPosition,
  xSpeed: radio.xSpeed,
  ySpeed: radio.ySpeed,
});
const [error, setError] = useState(null);

return (
  <form>
    <div>
      <span className="label">x Position</span>
      <input
        name="x-position"
        value={radioStatus.xPosition}
        onChange={(e) => {
          setRadioStatus((radioStatus) => ({ ...radioStatus, xPosition: e.target.value }));
        }}
      />
```

FIGURE 27. Using built-in browser input element example.

3.3.2 Radio position control

To handle the radio position moving, the input data for the new radio position in RadioForm component is required to be sent from the browser to Web_api service. It is necessary to create an interface for users to enter data, and then the data is validated automatically. After that, when users submit valid data, browsers will send these data to the Web_api service.

The following image describes an example of a building interface for a validation form. A form data is created by utilizing a built-in React.JS form, input field and button (67). In onChange function, setRadioStatus function handles showing data entered by the user on the screen. Meanwhile, setError function checks the input data and displays the error to the screen if it is invalid; in this case, the browser displays an error in red colour because the input value of x Position is beyond the range (-90; 90).

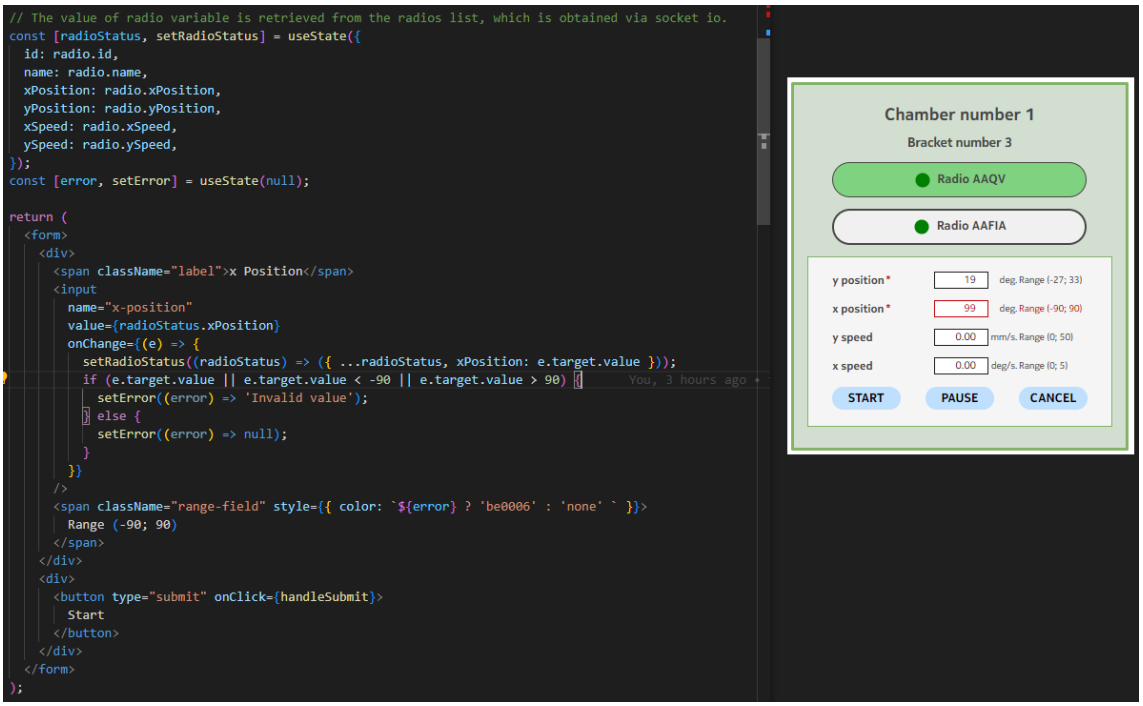


FIGURE 28. A sample of a validation form interface in Input form component.

When a user clicks the start button, the browser invokes the onClick function, in this case, known as handleSubmit function. This function first validates input data before calling the API PUT method to send input data to the Web_api service via the Axios library. (Figure 29).

```
// The value of radio variable is retrieved from the radios list, which is obtained via socket io.
const [radioStatus, setRadioStatus] = useState({
  id: radio.id,
  name: radio.name,
  xPosition: radio.xPosition,
  yPosition: radio.yPosition,
  xSpeed: radio.xSpeed,
  ySpeed: radio.ySpeed,
});
const [error, setError] = useState(null);

const handleSubmit = async (e) => {      You, 17 seconds ago • Uncommitted changes
  e.preventDefault();
  if (!error) {
    await axios
      .put('/api/action/manipulator_by_coordinate', radioStatus)
      .then((res) => console.log(res))
      .catch((err) => console.error(err));
  }
};

return (
  <form>
    <div>···
    </div>
    <div>
      <button type="submit" onClick={handleSubmit}>
        Start
      </button>
    </div>
  </form>
);
```

FIGURE 29. A submit handling function instance.

Everything is presently in working order. After designing some tiny components and styling for the app with some logic, the final figure illustrates an example of a basic finished real-time app and its lifecycle. The first image shows the current radio position, with the x position value set to 1. Picture number two displays the new radio position being entered by the user; it indicates that the user wishes to relocate the radio to a new location where the x position value is 8. After the user clicks the start button in picture number two to perform the radio moving action, image number three indicates the radio moving action is being conducted but has not yet been completed by the loading icon (in the start button). Furthermore, when the radio moving action is being performed, the real-time x position value is presented concurrently in image number three with value 6. Finally, after the radio moving action is completed; the screen exhibits the x position value of 8 in the fourth image, and the app enters a new life cycle.

FIGURE 30. An example of a basic completed real-time and its lifecycle.

4 DISCUSSION

The purpose of this project was to provide a web application that allows users to browse and control radio position in real-time in a chamber as part of the radio testing process. It can also be used to monitor multiple chambers simultaneously. This way, it enables end-users, also known as testers, to manage tests more effectively. For example, when numerous reserved chambers are used in a test case, users normally need to open multiple tabs on a browser or spend time visiting each control panel of real-world chambers to monitor and control radios' position. In this case, tracking and managing radio coordinates is a little more difficult. However, with the compact ChamberView component, users can open several chambers in a single browser tab if necessary.

At the end of the project, all important elements met the initial requirements, and the development processes were detailed in chronological sequence. The application was constantly optimized and developed during the design and implementation phase. The simple user interface is considered useful since it allows users to observe real-time radio status as well as control it on a single compact page. Directly validating radio status values in the browser also assists users in avoiding unwanted error responses that waste time when sending HTTP requests from the frontend to the backend.

Developing this project as a module of a larger project helped to hone skills in working with micro-services and understanding system architecture. It also aids in the improvement of reading and comprehension of existing code, as well as analytical and problem-solving abilities. Furthermore, the real-time topic contributes to the advancement of knowledge in the real-time and Internet of Things (IoT) fields. Particularly, many new technologies occurred in this project. Aside from the already familiar React.JS, the author had little experience with Socket.IO, FastAPI and none with NATS protocol, which was really a significant challenge. To address this issue, it is necessary to first understand how these new technologies work from the ground up, followed by the Docker platform; after which they can be applied to read and comprehend the structure in pre-built components. In summary, they open an opportunity to self-study and set the groundwork for occupations to come.

Further development is dependent on future user feedback, which will be transmitted from testers to the team leader and then transformed into new tickets or tasks in Jira. For example, the frontend UI can be upgraded and reused; also, the Redux Toolkit can be utilized for simpler management of multiple state management. Besides, whereas the real-time web application has completed the

development and build stages, as well as bypassed testing stages, it will be preferable if more Python-based unit tests and Cypress-based end-to-end tests are added to increase test coverage.

REFERENCES

1: Nokia Corporation 2021. Nokia to build new campus in Oulu. Search date 19.11.2023. https://www.nokia.com/about-us/news/releases/2021/12/03/nokia-to-build-new-campus-in-oulu/

2: Nhung, Do 2018. Tech wonders for Air Guitar Enthusiasts. Search date 19.11.2023. https://www.businessoulu.com/en/for-media/news/tech-wonders-for-air-guitar-enthusiasts.html

3: NATS 2021. What is NATS - NATS Docs. Search date 19.11.2023. https://docs.nats.io/nats-concepts/what-is-nats

4: Socket.IO 2023. Introduction - What Socket.IO is. Search date 22.11.2023. https://socket.io/docs/v3/

5: Sebastián Ramírez 2023. FastAPI Docs. Search date 22.11.2023. https://github.com/tiangolo/fastapi

6: Sebastián Ramírez 2023. FastAPI Docs - Example - Interactive API docs. Search date 22.11.2023. https://github.com/tiangolo/fastapi#interactive-api-docs

7: Stack Exchange 2023. 2023 Developer Survey - Web frameworks and technologies. Search date 22.11.2023. https://survey.stackoverflow.co/2023/#section-admired-and-desired-web-frameworks-and-technologies

8: Stack Exchange 2023. Stack Overflow Trends. Search date 22.11.2023. https://insights.stackoverflow.com/trends?tags=reactjs%2Cvue.js%2Cangular%2Csvelte%2Cangularjs%2Cvuejs3

9: Suraj, Surve 2021. Why You Should Use React.js For Web Development. Search date 22.11.2023. https://www.freecodecamp.org/news/why-use-react-for-web-development/

10: NATS 2023. Welcome to the official NATS documentation. Search date 22.11.2023. https://docs.nats.io/

11: NATS 2023. NATS - Protocol Demo. Search date 22.11.2023. https://docs.nats.io/reference/reference-protocols/nats-protocol-demo

12: NATS 2023. NATS - Client Protocol. Search date 22.11.2023. https://docs.nats.io/reference/reference-protocols/nats-protocol

13: Pramod, V G 2023. NATS: A Lightweight and Scalable Messaging System for Modern Applications. Search date 22.11.2023. https://www.linkedin.com/pulse/nats-lightweight-scalable-messaging-system-modern-applications-v-g/

14: Shay, Bratslavsky 2023. Comparing NATS and gRPC: Understanding the Differences. Search date 22.11.2023. https://memphis.dev/blog/comparing-nats-and-grpc-understanding-the-differences/
15: NATS 2021. NATS - Payload. Search date 22.11.2023. https://nats-io.github.io/nats.ts/enums/payload.html
16: NATS 2023. NATS - Security. Search date 22.11.2023. https://docs.nats.io/nats-concepts/security
17: NATS 2022. NATS - Subject-Based Messaging. Search date 22.11.2023. https://docs.nats.io/nats-concepts/subjects
18: NATS 2022. NATS - Queue Groups. Search date 22.11.2023. https://docs.nats.io/nats-concepts/core-nats/queue
19: NATS 2022. NATS - JetStream. Search date 22.11.2023. https://docs.nats.io/using-nats/developer/develop_jetstream
20: NATS 2023. NATS - JetStream - Consumers. Search date 22.11.2023. https://docs.nats.io/nats-concepts/jetstream/consumers
21: NATS 2023. NATS - JetStream - DeliverPolicy. Search date 22.11.2023. https://docs.nats.io/nats-concepts/jetstream/consumers#deliverpolicy
22: NATS 2023. NATS - Monitoring. Search date 22.11.2023. https://docs.nats.io/running-a-nats-service/nats_admin/monitoring
23: Socket.IO 2023. Socket.IO. Search date 22.11.2023. https://socket.io/
24: Wikimedia Foundation 2023. WebSocket. Search date 22.11.2023. https://en.wikipedia.org/wiki/WebSocket
25: Socket.IO 2023. Socket.IO Introduction Version: 4.x. Search date 22.11.2023. https://socket.io/docs/v4
26: Mozilla Corporation 2023. WebTransport API. Search date 22.11.2023. https://developer.mozilla.org/en-US/docs/Web/API/WebTransport_API
27: Socket.IO 2023. Socket.IO - Server API. Search date 22.11.2023. https://socket.io/docs/v4/server-api/
28: Socket.IO 2023. Socket.IO - Emitting events. Search date 22.11.2023. https://socket.io/docs/v3/emitting-events/
29: Guy, Bar-Gil 2022. Asynchronous Programming in Python – Understanding the Essentials. Search date 22.11.2023. https://www.mend.io/free-developer-tools/blog/asynchronous-programming-in-python-understanding-the-essentials/

30: Brad Solomon 2019. Async IO in Python: A Complete Walkthrough. Search date 22.11.2023. https://realpython.com/async-io-python/#the-asyncio-package-and-asyncawait

31: Python Software Foundation 2023. Asyncio - Asynchronous I/O. Search date 22.11.2023. https://docs.python.org/3/library/asyncio.html

32: Rishu, Mishra 2022. Difference Between MVC, MVP and MVVM Architecture Pattern in Android. Search date 22.11.2023. https://www.geeksforgeeks.org/difference-between-mvc-mvp-and-mvvm-architecture-pattern-in-android/

33: FastAPI Practices 2023. FastAPI - Best Architecture. Search date 22.11.2023. https://github.com/fastapi-practices/fastapi_best_architecture

34: Sebastián Ramírez 2020. FastAPI - Bigger Applications - Multiple Files. Search date 22.11.2023. https://fastapi.tiangolo.com/tutorial/bigger-applications/

35: Sebastián Ramírez 2020. FastAPI - SQL (Relational) Databases. Search date 22.11.2023. https://fastapi.tiangolo.com/tutorial/sql-databases/

36: Pydantic Services 2023. Pydantic. Search date 22.11.2023. https://docs.pydantic.dev/latest/

37: Encode OSS Ltd 2018. Starlette Introduction. Search date 22.11.2023. https://www.starlette.io/

38: Siddhant, Goel 2022. Why you should choose Starlette for your next SaaS. Search date 22.11.2023. https://dev.to/siddhantgoel/why-you-should-choose-starlette-for-your-next-saas-3eln

39: Sebastián Ramírez 2020. FastAPI - Background Tasks. Search date 22.11.2023. https://fastapi.tiangolo.com/tutorial/background-tasks/

40: Sebastián Ramírez 2020. FastAPI - Lifespan Events. Search date 22.11.2023. https://fastapi.tiangolo.com/advanced/events/

41: Sebastián Ramírez 2020. FastAPI - Using the Request Directly. Search date 22.11.2023. https://fastapi.tiangolo.com/advanced/using-request-directly/

42: Sebastián Ramírez 2020. FastAPI - Handling Errors. Search date 22.11.2023. https://fastapi.tiangolo.com/tutorial/handling-errors/#override-request-validation-exceptions

43: SmartBear Software 2015. Swagger UI. Search date 22.11.2023. https://swagger.io/tools/swagger-ui/

44: Sebastián Ramírez 2019. FastAPI - Validation. Search date 22.11.2023. https://fastapi.tiangolo.com/features/#validation

45: Sebastián Ramírez 2020. FastAPI - Declare Request Example Data. Search date 22.11.2023. https://fastapi.tiangolo.com/tutorial/schema-extra-example/

46: Pydantic Services 2023. Pydantic - Why use Pydantic? Search date 22.11.2023. https://docs.pydantic.dev/latest/why/#performance
47: Pydantic Services 2023. Pydantic - Validators. Search date 22.11.2023. https://docs.pydantic.dev/latest/concepts/validators/
48: Pydantic Services 2023. Pydantic - Numeric Constraints. Search date 22.11.2023. https://docs.pydantic.dev/latest/concepts/fields/#numeric-constraints
49: Pydantic Services 2023. Pydantic - Validators. Search date 22.11.2023. https://docs.pydantic.dev/1.10/usage/validators/
50: Romulo, Gatto 2023. FastAPI vs Flask: The Ultimate Showdown of Python Web Frameworks!. Search date 22.11.2023. https://medium.com/@romulo.gatto/fastapi-vs-flask-the-ultimate-showdown-of-python-web-frameworks-1f2700ac8852
51: Planeks 2023. Flask vs FastAPI for API Development. Search date 22.11.2023. https://www.planeks.net/flask-vs-fastapi-for-api-development/
52: Peter, Ekene Eze 2023. Building reusable UI components with React Hooks. Search date 22.11.2023. https://blog.logrocket.com/building-reusable-ui-components-with-react-hooks/
53: Meta Platforms 2023. Built-in React Hooks. Search date 22.11.2023. https://react.dev/reference/react/hooks
54: Refsnes Data 2019. React JSX. Search date 22.11.2023. https://www.w3schools.com/REACT/react_jsx.asp
55: Meta Platforms 2023. React - Components and Props. Search date 22.11.2023. https://legacy.reactjs.org/docs/components-and-props.html
56: Meta Platforms 2023. React - Passing Props to a Component. Search date 22.11.2023. https://react.dev/learn/passing-props-to-a-component
57: Ebenezer Don 2023. React Hooks vs. Redux: Do Hooks and Context replace Redux?. Search date 22.11.2023. https://blog.logrocket.com/react-hooks-vs-redux-hooks-context-replace-redux/
58: Meta Platforms 2023. React Component. Search date 22.11.2023. https://legacy.reactjs.org/docs/react-component.html
59: Meta Platforms 2023. React - useEffect. Search date 22.11.2023. https://react.dev/reference/react/useEffect
60: Meta Platforms 2018. React lifecycle methods diagram. Search date 22.11.2023. https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/
61: Mozilla Corporation 2020. Main thread. Search date 22.11.2023. https://developer.mozilla.org/en-US/docs/Glossary/Main_thread

62: Reed, Barger 2022. The React useEffect Hook for Absolute Beginners. Search date 22.11.2023. https://www.freecodecamp.org/news/react-useeffect-absolute-beginners/

63: Socket.IO 2023. Socket.IO - How to use with React. Search date 22.11.2023. https://socket.io/how-to/use-with-react

64: Refsnes Data 2018. What is npm? Search date 22.11.2023. https://www.w3schools.com/whatis/whatis_npm.asp

65: Remix Software 2022. React Router tutorial. Search date 11.12.2023. https://reactrouter.com/en/main/start/tutorial

66: John Jakob "Jake" Sarjeant 2023. Axios - Promise based HTTP client for the browser and node.js. Search date 11.12.2023. https://github.com/axios/axios

67: Meta Platforms 2023. React - API reference - component - input. Search date 11.12.2023. https://react.dev/reference/react-dom/components/input