The implementation of BifrostConnect Front-end scope, re-design and development with the relevant back-end support develop.

Fei Gu
Erhvervs Akademi Sydvest
Computer Science 21

January 7, 2024

Supervisor: Henrik Boulund Meng Hansen
Company: BifrostConnect
Engineering Director: Jasper Wass

Contents

1 Introduction

1.1 Background

This project marks the culmination of the EASV Computer Science program. Its primary objective is to showcase the breadth and depth of knowledge and skills that the student has amassed over two and a half years of study, and to apply these competencies in a practical setting.

In the scope of this project, the student will collaborate with a company to tackle a specific problem they are encountering. The student will leverage the knowledge and skills gleaned from their education to devise a solution.

This implies that the project will be directly relevant to the company's business domain, and will employ both software engineering and software development methodologies to identify, analyze, and ultimately resolve the problem.

1.2 The company

BifrostConnect, a startup founded in 2018, is a dedicated team of 10 to 12 professionals based in Copenhagen, Denmark. Over the past five years, the company has developed a unique solution for remote access system.

This solution is specifically designed for systems that operate under strict security conditions and are not permitted to have internet connectivity. Instead of relying on third-party services, BifrostConnect uses a relay device to facilitate remote access.

BifrostConnect's main offering is a hardware device coupled with a network application. This combination enables secure, remote access to client systems.

In addition to this, the company provides related services and comprehensive solutions to cater to a wide range of client needs.

1.3 The project

This project entails a thorough redesign and redevelopment of the existing front-end software, leveraging the foundation provided by the company's original software.

Additionally, it involves enhancing the back-end to provide the necessary support for the new front-end.

This project will follow the software design process. Therefore, it will also include the Software Engineering process, the Software Design method, the Software Development method, and DevOps on all process as well.

This project is covered by NDA due to security and confidentiality reasons. Thus, the project will initially not be publicly deployed, however, I will describe the deployment in my report as well.

2 Problem Statement

2.1 Statement

In the scope of this project, the integration of the Remote Access Interface and Tunnel Interface into the Device Manager Application is a primary objective. The goal is to create a single-page application that fully encapsulates the functionality of both the Remote Access and Tunnel Connection capabilities.

2.2 Situation

The existing BifrostConnect front-end application is divided into two distinct user interfaces: the Device Manager (DM) and the Remote Access Interface (RAI). This division requires users to manage their devices and users in one application, while operating their devices to use the Bifrost product in another web application. Moreover, the RAI is further split into two different user interfaces, known as the Classic RAI and Tunnel RAI. These interfaces direct users to different web applications depending on the type of access solution they wish to create.

This situation arose because the development of the Bifrost product was primarily focused on the device, with the application part not receiving as much attention. As a result, the application part was developed by different developers, each with their own design logic. This led to a lack of thorough planning for the application functionality during the development process. Consequently, the design logic of the product is not scalable, which has resulted in a diminished user experience and an exponential increase in code complexity, making it increasingly difficult to maintain and improve. Furthermore, the product design style lacks scalability due to a lack of modularity and code consistency, and the operation logic is unclear.

2.3 Potential Solution

To address those issue, there is a need to redesign and remake the front-end with a focus on implementing a modular unified front-end management system that encompasses all the necessary functionality.

The goal is to provide users with seamless control, monitoring, and access to their devices from anywhere, enhancing their overall user experience and productivity while keeping code complexity as low as possible.

For the implementation to be successful, we also need to dedicate a back-end service, including an API and Database. The back-end server will contain the necessary service for the front-end to reach the purpose. Such as the RESTful API service, authentication service, MQTT service, Data Channel service and Database.

3 Requirement Analysis

The requirement analysis in this project is based on the problem statement. After analyzing the problem statement, we will explore the perspectives of various stakeholders to understand their specific needs. This will help us gain a comprehensive understanding of the project requirements.

To ensure a focused approach, we will define the project's limitations and constraints. This will help us set realistic expectations and avoid scope creep.

Finally, we will establish clear and measurable goals for this project. These goals will serve as benchmarks to evaluate the success of the project and guide our development process.

3.1 Stakeholders

The stakeholders are the individuals or groups who are relevant to the issues of this project.

There are three main stakeholders in this project:

- User
- Developer
- Project Owner

User The user is the person who will use the system.

In this project, the user refers to the customer who will utilize the web application to manage their devices, users, and create the RAI and Tunnel.

The user's requirements for the web application include ease of operation, intuitive recognition of features, and clear usability.

Project Owner The project owner is the individual or organization responsible for managing the project.

In this case, the project owner is the company that owns the product, which is the BifrostConnect.

The project owner has several requirements for the project:

- Timely completion of the project.
- Compatibility with the hardware developed by the company.
- Easy maintenance, updates, migration, and handover to other developers.

Developer The developer is responsible for developing, modifying, scaling, and fixing the system in any situation.

To facilitate these tasks, there are several requirements for the developer to reach out, the system should have clear and concise comments, a well-organized structure that is easy to modify, reusable components for easy scalability, and effective bug identification for efficient issue resolution.

3.2 Business Domain

During the requirement analysis, we can base on the company's business domain to define the domain model.

The domain model is a conceptual model of the domain that incorporates both behavioral and data concepts. The domain model is used to describe the various entities, their attributes, and their relationships.

The domain model following:

Organization The organization is the customer who will use the Bifrost unit (the product) and solution to solve their remote access issues.

It has all the users and devices and can group them together to allow the users to use the devices to connect to remote equipment.

The organization has the following attributes:

- name: The name of the organization.
- users: A collection of users.
- devices: A collection of devices.
- groups: A list of groups.

User The user can be a technician or an operator who uses the device to connect to remote equipment from the local equipment. They are the starting and ending point of all processes.

The user also can be the manager who manage the users and devices in the organization. Or can be the normal user who belongs to a group to use the device in same group to connect to remote equipment.

The user has the following attributes:

- username: The identifier of the user.
- role: The role of the user, can be normal user or admin.

Device The device is the hardware that using for connects to remote equipment. Which is the part of the product from BifrostConnect. The device can be set in a group to allow the user to use it to connect to remote equipment.

The device also have two different types, one is Attend Unit, another is Unattended Unit.

The device has the following attributes:

- name: The identifier of the device.
- type: The type of the device, can be Attend Unit or Unattended Unit.

Group The group is a collection of users and devices created by the organization. Only users and devices belonging to the same group can interact with each other.

Which means the user can only use the device in the same group to connect to remote equipment.

The group has the following attributes:

- Name: The identifier of the group.
- Users: A list of users.
- Devices: A list of devices.

3.3 Scope

The scope of this project is to develop a web application that manages devices and users, and creates the RAI and Tunnel.

But since the device manager already exists, the main focus of this project will be on implementing the RAI and Tunnel into the device manager.

Therefore, there is no need to develop the device manager or add new features to the RAI and Tunnel interface at the beginning of the project.

As the Device Manager is built using the React framework, we also use the same framework to develop the RAI and Tunnel interface.

Therefore, to develop an adequate architecture for the entire system also be a big task for this project.

Furthermore, due to the redundant and coupled nature of the entire system, we can consider redesign the system components and establish their relationships.

3.4 Goals

The goal of this project is to develop a RAI and Tunnel interface in the Device Manager that can create, modify, delete, and view access connections. Additionally, it should have the capability to remotely control the device and target equipment.

4 Software Design

4.1 Software Development Methods

Project methodology refers to a systematic set of methods. And principles adopted during project management and implementation. It includes specific steps and approaches for planning, organizing, implementing, monitoring, and closing different phases of the project.

Project methodology helps project teams effectively manage. And control projects, ensuring timely, high-quality, and cost-effective completion, and achieving the desired goals and benefits.

Common project methodologies include Agile development, Waterfall model, and Iterative development.

Choosing the appropriate project methodology is crucial for project success.

4.1.1 Agile Software Development

The Agile software development is a set of software development methods to aim to provide a sustainable, iterative and incremental software development process.

The Agile promotes adaptive planning, evolutionary development, early delivery, continuous improvement, encourages rapid and flexible response to change.

Since this project is a single person project, the Agile software development method is not suitable for this project. However, the Agile software development method is still a good reference for the project management.

Therefore, in this project will try to use part of the scrum method to manage the project daily tasks.

The fully period of this project from 10/15/2023 to 1/15/2024, which will cost three months. The project will be divided into six sprints, each sprint will be contained two weeks. The first sprint will be started from 10/15/2023 to 10/29/2023, and the last sprint will be started from 12/31/2023 to 1/15/2024.

The first sprint will be used to do the requirement analysis, the domain model analysis, the Technology research.

The second sprint will be used to do the system design, which included the database design, the back-end design, the front-end design, and the DevOps design.

The third sprint will be used to do the overall development, to make the entire system can be run without any useful features.

The fourth sprint will be used to do the feature development, to implement the basic functionalities of the system.

The fifth sprint will be used to do the connection development, which use for creating a session from the local device to the remote equipment.

The last sprint will be used to do the testing, the deployment, and the documentation.

4.1.2 Feature Driven Development

Feature Driven Development (FDD) is an iterative and incremental software development process. It is a model-driven short-iteration process that consists of five basic activities:

1. Develop an overall model
2. Build a feature list
3. Plan by feature
4. Design by feature
5. Build by feature

The FDD based on the feature underlying principles. At the beginning, it cost less time to develop a temporary system to contain a basic structure and can be run for no feature.

Then, the features will be design and develop into the system one by one. Each feature will be developed in a short iteration, which will be contained the design, the development, and the testing.

Develop an overall model The first step of the FDD is to develop an overall model. The overall model is a domain model that proposed to let the application can have a basic structure and can be run for no feature. After the overall model is developed, the feature list can be created for adding the features to the application.

In this project, because the main focus of this project is to migrate the interfaces into the exist application, the overall model will be only force on the UI, state management, and constructure of interface in the Device Manager.

Feature list The feature list is a list of the features that the application will have. It will follow the use-case that be identified in the user story. And connect the all front-end, back-end and database together.

Therefore, each feature will produce the sequence start from the front-end to the back-end and then to the database and the device, and return the result to the front-end to display to the user.

4.2 Technology selection

Before the development process be start, there are multiple technology actually fit with this project. But because of the existing application already have some decisions about the usage of technology.

Therefore, the technology selection will be based on the existing application and extended some of useful newer technology to be choice.

4.2.1 Front-end

JavaScript The JavaScript is a fundamental programming language of web development. Since the original front-end application was using this, there are no reason to change it.

React The React is a JavaScript library for building user interfaces. It is maintained by Facebook and a community of individual developers and companies. React can be used as a base in the development of single-page or mobile applications.

React-Router The React-Router is a routing library for React. It abstracts away the details of server and client and allows developers to focus on building apps.

Redux The Redux is a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments (client, server, and native),and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger.

RTK The RTK is a powerful, opinionated Redux tool set for writing better reducers, organizing and accessing state in components, and managing side effects.

RTKQ RTKQ is a powerful data fetching and caching tool. It is designed to simplify common cases for loading data in a web application, eliminating the need to hand-write data fetching & caching logic yourself.

Tailwind CSS The Tailwind CSS is a utility-first CSS framework for rapidly building custom user interfaces. It is a highly customizable, low-level CSS framework that gives you all the building blocks you need to build bespoke designs without any annoying opinionated styles you have to fight to override.

Semantic UI is a development framework that helps create beautiful, responsive layouts using human-friendly HTML. Semantic UI treats words and classes as exchangeable concepts. Classes use syntax from natural languages like noun/modifier relationships, word order, and plurality to link concepts intuitively.

Jest The Jest is a JavaScript testing framework maintained by Facebook, Inc. designed and built by Christoph Nakazawa with a focus on simplicity and support for large web applications. It works with projects using Babel, TypeScript, Node.js, React, Angular, Vue.js and Svelte. Jest does not require a browser to run tests, and it runs tests in parallel - this makes Jest fast. Jest is well-documented, requires little configuration and can be extended to match your requirements.

4.2.2 Back-end

CSharp CSharp is a general-purpose, multi-paradigm programming language.

dotNET Core dotNET is a free and open-source, managed computer software framework for Windows, Linux, and macOS operating systems.

Entity Framework EF is an open-source ORM framework for .NET applications supported by Microsoft.

xUnit xUnit is a unit testing framework for the .NET.

Moq Moq is a mocking library for .NET for mocking interfaces and classes.

4.2.3 Database

SQLite SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine.

4.2.4 Data communication

HTTP HTTP is an application-layer protocol for transmitting hypermedia documents, such as HTML. It was designed for communication between web browsers and web servers, but it can also be used for other purposes. HTTP follows a classical client-server model, with a client opening a connection to make a request, then waiting until it receives a response. HTTP is a stateless protocol, meaning that the server does not keep any data (state) between two requests. Though often based on a TCP/IP layer, it can be used on any reliable transport layer; that is, a protocol that doesn't lose messages silently, such as UDP.

FlatBuffers FlatBuffers is an efficient cross-platform serialization library for C++, C#, C, Go, Java, JavaScript, Lobster, Lua, TypeScript, PHP, Python, and Rust. It was originally created at Google for game development and other performance-critical applications.

MQTT MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium.

WebRTC WebRTC is a free, open-source project that provides web browsers and mobile applications with real-time communication (RTC) via simple application programming interfaces (APIs). It allows audio and video communication to work inside web pages by allowing direct peer-to-peer communication, eliminating the need to install plugins or download native apps.

4.2.5 DevOps

Git Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

GitHub GitHub is a provider of Internet hosting for software development and version control using Git. It offers the distributed version control and source code management (SCM) functionality of Git, plus its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.

GitHub Actions GitHub Actions is an API for cause and effect on GitHub: orchestrate any workflow, based on any event, while GitHub manages the execution, provides rich feedback, and secures every step along the way. With GitHub Actions, workflows and steps are just code in a repository, so you can create, share, reuse, and fork your software development practices.

Docker Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. All containers are run by a single operating system kernel and are thus more lightweight than virtual machines. Containers are created from images that specify their precise contents. Images are often created by combining and modifying standard images downloaded from public repositories.

4.3 Architecture design

The overall architecture design outlines the comprehensive structure of the system. The primary objective of this design is to ensure the system's functionality, reliability, scalability, and maintainability.

In this project, the architecture design is based on the Client/Server (C/S) model, with an additional module incorporated for facilitating remote communication.

The client-side of the application is a web browser, serving as the primary interface for user interaction. User operations include data requests, information display, data modification, device configuration, and the creation and management of classic remote access and tunnels.

The server-side of the application is responsible for processing client requests, executing database operations (such as Create, Read, Update, and Delete operations) based on the request content, and returning the results to the client.

The additional module in the back-end for remote communication is designed to implement the functionality of remote communication, ensuring its reliability, scalability, and maintainability.

In this context, the MQTT protocol is employed, with the open-source Mosquitto serving as the MQTT server and the open-source Paho as the MQTT client.

The WebRTC, an open-source technology, is utilized for remote communication.

The system utilizes a Relational Database Management System (RDBMS), specifically SQLite, for data management.

In the C/S architecture, the RESTful API serves as the interface definition, JSON is used as the data transmission format, and HTTP is used as the data transmission protocol. Between the front-end with the MQTT broker and WebRTC server, the FlatBuffers protocol will be implemented into this data transmissions.

And there are an end point which is the device will also create a connection together with the Front-end and Back-end.

4.3.1 Front-end

In the architecture design of the front-end, the primary goal is to migrate the existing remote access interface and the tunnel interface into the web application. And the secondary goal is refactoring the device manager as the well-structured web application.

The front-end utilizes the React framework based on view design and the Redux framework based on state and data flow.

The structure of the React framework divides the web page into multiple components based on modularity. Each component has its own state and properties.

Components are categorized into container components, presentational components, and functional components.

Container components manage data and state, presentational components display data and state, and functional components implement specific functionalities

Remote Access Interface The remote access interface needs to be redesigned as a single page application, and its purpose is to provide the interface to create the remote access connection, display the connection status, the device status, and the data stream from the device.

For the UI of the remote access interface, the component will be injected into the RAI page from the page component, and the component will be the Bifrost page component which is the reusable page component directory. The prototype described following:

- The top of the component is the status bar, which contains the status of the device.
  - The left of the status bar is the device name, serial number, and device framework version.
  - The middle of the status bar is the device status, it will display the device status, such like charging status, battery status, and the device connection status.
  - The right of the status bar is the exit session button, it will exit the current session.
- The main content is the body of the component, which display the data stream from the device.
- The bottom of the component control bar, it includes two part.
  - The left of the control bar is the tag switcher, it will switch the data stream between the KVM data stream, and the serial data stream.
  - The right of the control bar is the control buttons, it includes the setting button, will display the setting menu, the full screen button, will display the data stream in full screen in the browser, and key map button, will display the key map menu.

The UI state will be isolated from the session state which will be interactive with the session state. For UI the state will not store in the Redux store, it will store in each of the component.

For the session be created for this interface, it will be store in the Redux store as the session state. The session state will be store in the session reducer, and the session reducer will be combined with the root reducer.

The session state will be store as the object, and the object will be store in the session array. Use this pattern we can easily to create multiple session for the remote access interface, and easily to manage, switch, and delete the session.

Each session will have its own state, the state include:

- id, which is the unique ID for the session. Because each of the device can only active one session at the same time, so the session ID will be the same as the device serial number.
- name, which is the name of the session, we can use the device name as the session name.
- status, this is the flag of the session active or not.
- WebRTC connection, which is the WebRTC connection object of the session.
- MQTT connection, which is the MQTT connection object of the session.

For creating the session, there are some hooks will be use in the component, for example:

- useCreateSession, this hook will be used to create the session, and it will return the session object.
- useCloseSession, this hook will be used to close the session, and it will return the session object.
- keepAliveSession, this hook will be used to keep the session alive, and it will return the session object.
- useMQTT, this hook will be used to subscribe the MQTT topic, receive the MQTT message, and publish the MQTT message.
- useWebRTC, this hook will be used to create the WebRTC connection, and so on.
- useFlatBuffer, this hook will be used to process the FlatBuffer message.

Tunnel Interface Same as the remote access interface, the tunnel interface also need to be redesigned as a single page application, and its purpose is to provide the interface to create the tunnel connection, display the connection status, the device status, and the data stream from the device.

For the UI of the tunnel interface, it will be following the same pattern as the remote access interface, and the UI design will be little different.

The reason for the different UI design is the tunnel interface will be use two devices to create the tunnel, connecting each of the end point though the tunnel.

The UI will be scribes following:

- The top of the component is the connecting status bar, which contains each of the device status. Between the two device status will be the tunnel status.
- The main content is the body of the component, except to display the data stream from the device, it also includes a sidebar for display the IP mapping list.

The rest of the state design and the hooks design will be the same as the remote access interface depends on the reuse of the component.

4.3.2 Back-end

The backend uses the dotNet framework. The development language using the C# language.

In this project, the backend uses the Onion Architecture. The Onion Architecture is a typically layered architecture, where each layer depends on the inner layer and provides interfaces to the outer layer. The outer layer provides services to the outermost layer and other modules in the same layer based on the interfaces of the inner layer.

From inner to outer, the layers are: Domain, Application, Infrastructure, Presentation. The Domain layer is the core layer and the innermost layer, used to define domain models, which are the business models. It includes domain models and domain service interfaces. Domain models are used to define the business models, which are the entities in the entity-relationship model and their attributes. Domain service interfaces are used to define the business services, which are the relationships between entities in the entity-relationship model.

The Application layer is the application layer, used to define application services, which are the business logic. It includes domain service implementations and application service interfaces. Domain service implementations implement the methods of the inner layer's domain service interfaces and implement the business logic of the domain models. Application service interfaces are used to define application services, which are the business logic. It includes but is not limited to database interfaces, testing interfaces, HTTP API interfaces, MQTT interfaces, etc.

The Infrastructure layer is the infrastructure layer, used to define infrastructure. It includes database implementations, testing implementations, HTTP API implementations, MQTT implementations, etc. Database implementations implement the database interfaces and provide CRUD services for the database. Testing implementations implement the testing interfaces and provide services for unit testing and integration testing. HTTP API implementations implement the HTTP API interfaces and provide CRUD operations for HTTP APIs. MQTT implementations implement the MQTT interfaces and provide CRUD operations for MQTT.

The Presentation layer is the presentation layer, used to define presentation logic, such as interfaces and pages. Since this is a backend project, data presentation and control are handled by the frontend, so this layer is not needed.

4.3.3 Data communication and storage

Design of data communication:

Selection of communication protocols There are three types of data transmitted from the front-end to the back-end:

- Requests for basic CRUD operations, which have low requirements for timeliness but require accuracy, integrity, order, and security. For this type of data transmission, the HTTP protocol and RESTful API design are used to effectively meet the above requirements.
- Creation of data channels and transmission of streaming media data, which require high timeliness and security. For this type of data transmission, the WebRTC protocol and MQTT protocol are used, along with the fast-decoding FlatBuffer protocol, to effectively meet the above requirements.
- Transmission of device status information and operation information. Which require high integrity, order, and security. For this type of data transmission, the MQTT protocol is used along with the FlatBuffer protocol.

Communication architecture and flow of data communication The data communication architecture of this project is based on a front-end and back-end separation architecture, with React framework used for the front-end and dotNet framework used for the back-end.

When the front-end needs to send data to the back-end, it sends an HTTP request to the back-end. Upon receiving the HTTP request, the back-end selects different data processing methods based on the data type of the request. For basic CRUD requests, the back-end performs CRUD operations on the data based on the RESTful API design. For the creation of data channels and transmission of streaming media data, the back-end creates data channels based on the WebRTC protocol and helps establish data channels between the front-end and devices. Once the data channel is established, the front-end and devices use the FlatBuffer data format for streaming media data transmission. For the transmission of device status information and operation information, the front-end directly sends MQTT requests to the MQTT broker. The devices listen to the relevant MQTT requests in their firmware and return the corresponding data.

Data formats for data communication There are three data formats for data communication in this project:

- HTTP protocol: Data is transmitted using the JSON format.
- WebRTC protocol: Data is transmitted using the FlatBuffer format.
- MQTT protocol: Data is transmitted using the FlatBuffer format.

Design of data storage:

- Selection of database for data storage: For data storage in this project, a lightweight database SQLite is used. SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. This choice is made because the purpose of the entire project is to achieve data communication between the front-end and devices. There is no high requirement for database operations such as CRUD, the data volume is small, and the transactional requirement for database data is not high. Therefore, SQLite database is chosen.

- Design of data structures for the front-end and back-end of the project: In this project, the front-end uses the React framework, so the design of data structures for the front-end adopts a state-based approach. Each component or data set contains a state object, and the properties of this state object represent the various states of the component. The use of state objects facilitates state management. By using the object-property form, it is easy to distinguish between similar states of different components. Since cross-component states are managed by Redux, this design of state objects enables easier state updates and transfers. The back-end uses the dotNet framework, so the design of data structures for the back-end adopts a class-based approach. Object-oriented programming principles are used to encapsulate data, making data transmission more secure, ordered, and complete.

4.4 DevOps CI/CD process design

4.4.1 Continuous Integration

The continuous integration is the practice of merging the code from each of develop process into a source control repository to integrate and test the code.

The proposal for continuous integration is to quickly check the code is working for different part of the project together which means integration.

For this project, each of the feature will be developed in separate branch and should be merged into the master branch and test as a whole project.

Therefore, when the feature is developed, the developer should push the code to the remote repository, and create a pull request to merge the code into the master branch.

Pull Request When the developer pushes the code to the remote repository, the pull request will be created automatically.

The pull request will trigger the continuous integration process, which will build the project and run the test cases. If the test cases are passed, the pull request will be approved and the code will be merged into the master branch.

The continuous integration process will be implemented by the GitHub Actions.

Start from the checkout the code from the remote repository, the GitHub Actions will build the project.

For the front-end project, automatically the GitHub Actions will set up the Node.js environment, install the dependencies, and build the project using the NPM command.

After build the project, the GitHub Actions will

Continuous Testing The continuous testing is the practice of running automated test cases by using the NPM command in the GitHub Actions.

That will lead the NPM to run the unit test cases and then generate the test coverage report.

After the unit test, the GitHub Actions will run the end-to-end test cases by using the E2E testing framework.

4.4.2 Continuous Delivery

Once the software is built, it is delivery to the test environment. The test environment is a server that is used to E2E test and performance test the software.

In this project, the test environment is a server that is running on the docker container. Therefore, the GitHub Actions will build the docker image and push the image to the private docker registry.

After the docker image is pushed to the private docker registry, the docker image will be pulled by the test environment server. Then the docker image will be run on the test environment server with the full project including the front-end, back-end, database, MQTT broker/client, and the WebRTC ICE server/ TUNE server.

E2E Testing The E2E testing is the practice of testing the software from the user's perspective. The E2E testing will test the software as a whole project, which means the front-end, back-end, database, MQTT broker/client, and the WebRTC ICE server/ TUNE server.

The E2E testing start from the front-end, the E2E testing framework will open the browser in the test environment server, and then the E2E testing framework will simulate the user's action to require the feature's function. And then check the result of the action is correct or not.

For example, the E2E testing framework will open the browser and then login to the system. After login to the system, the E2E testing framework will check the login result as the expected result.

Performance Testing The performance testing is the practice of testing the software's performance. It's including the response time, throughput, and the resource utilization.

4.4.3 Continuous Deployment

The continuing Deployment is the next step of the continuous delivery. It is the practice of automatically deploying the software from test environment to the production environment.

Production Environment The production environment is a server that is used to run the software. Basically, the production environment is the same as the test environment, but without the test tools in the production environment.

Continuing Operation The production environment is the relay server to face to the real user. Therefore, the production environment also includes the performance monitoring tools to monitor the performance of the software. Response the performance issue and bugs as soon as possible.

5 Software Development

In this step, the project will be got involved in the develop process. At the beginning, the project will follow the design phase to make an overall application which includes the front-end, back-end and database. Also develop the GitHub actions as the CI/CD pipeline and prepare the deployment process.

5.1 Overall development

The overall development require the minimum of the project can be operating without any feature as well.

Therefore, it is only necessary to develop the basic three part of the entire system.

- Front-end
- Back-end
- Database

But based on the three part, the project also need the CI/CD pipeline setup, and the unit test for the basic operation.

5.1.1 Front-end

The front overall development will create the basic front-end for the project. For developing the front-end, we need to create the following components:

- pages
- components
- hooks
- store
- utils
- styles

Pages In the React, the page is the container of the application using the JSX syntax. The page is the structure of the application.

See the figure 1 for the device page example.

```
        const devicePage= () => {
            return (
                <>
                    <headerComponent />
                    <deviceComponent />
                    <footerComponent />
                </>
            )
        }

```

Figure 1: The device page of the application

Components The components are the reusable elements in the application. Which included two types of components:

- page components: the reusable page components included multiple element components. Such as the user component, the device component, the group component, etc.
- element components: the reusable basic elements. Such as the button, the input, the select, etc.
- specific components: the components reusable with the specific usage. Such as the session component, the data channel component, etc.

See the figure 2 for the device component example.
See the figure 3 for the table component example.

Hooks The hooks are the reusable functions in the application which used the React customize hooks. The hooks should be the pure functions. And should operate its own scope and state. The hook can be invoked in any of the components. See the figure 4 for the useGetDeviceList hook example.

Store The store is the global state of the application. In this project we use the Redux to manage the global state. Instead, using the original Redux, we use the Redux Toolkit to simplify the Redux development.

Therefore, the store's `index.js` file is the Redux Toolkit store which used for combine the reducers and the middlewares. The store's `slice.js` file is the reducer of the store which used for manage the state of the store.

See the figure 6 for the store index example.

The Slice is the reducer of the store. It is a collection that contains the state and the reducers for the one kind of the states.

For example, the deviceSlice is the slice for the device state. It contains multiple states such like the deviceList, the deviceInfo, the deviceStatus, etc.

```
        const deviceComponent = () => {
            const {
                data: deviceData,
                loading: deviceLoading,
                error: deviceError,
                getDeviceList
                } = useGetDevicelist();

            useEffect(() => {
                getDeviceList();
            }, []);

            return (
                <>
                    {deviceError &&
                        <div>Failed to load device list</div>}
                    {deviceLoading &&
                        <div>Loading...</div>}
                    {!deviceLoading && !deviceError &&
                        <TableComponent data={deviceData} />}
                </>
            )
        }

```

Figure 2: The device component of the application

```
        const Table = ({ data }) => {
            const columns = useMemo(() => COLUMNS, []);
            const tableData = useMemo(() => data, [data]);
            const {
                getTableProps,
                getTableBodyProps,
                headerGroups,
                rows,
                prepareRow,
                } = useTable({
                    columns,
                    data: tableData,
                });

            return (
                <table {...getTableProps()}>
                    <thead>
                        {headerGroups.map(headerGroup => (
                            <tr {...headerGroup.getHeaderGroupProps()}>
                                {headerGroup.headers.map(column => (
                                    <th {...column.getHeaderProps()}>
                                        {column.render('Header')}
                                    </th>
                                ))}
                            </tr>
                        ))}
                    </thead>
                    <tbody {...getTableBodyProps()}>
                        {rows.map(row => {
                            prepareRow(row);
                            return (
                                <tr {...row.getRowProps()}>
                                    {row.cells.map(cell => {
                                        return (
                                            <td {...cell.getCellProps()
}>
                                                {cell.render('Cell')}
                                            </td>
                                        )
                                    })}
                                </tr>
                            )
                        })}
                    </tbody>
                </table>
            )
        }

```

Figure 3: The Table component

```
        export const useGetDeviceList = () => {
            const [deviceList, setDeviceList] = useState([]);
            const [loading, setLoading] = useState(true);
            const [error, setError] = useState(null);

            const getDeviceList = async () => {
                setLoading(true);

                return fetch(
                    env + "/api/devices",
                    {
                        method: "GET",
                        headers: {
                            "Content-Type": "application/json",
                            Authorization: `Bearer ${localStorage.
    getItem(
                                "token"
                            )}`,
                        },
                    }
                )
                    .then((res) => res.json())
                    .then((data) => {
                        setDeviceList(data);
                        setLoading(false);
                    })
                    .catch((error) => {
                        setError(error);
                        setLoading(false);
                    });
                )
            };

            useEffect(() => {
                deviceList.length === 0 && getDeviceList();
            }, []);

            return { deviceList, loading, error };
        }

```

Figure 4: The useGetDeviceList hook

And it contains multiple reducers to modify the relevant states when there is an action to cause the state change.

See the figure 5 for the deviceSlice example.

Utils The utils are the pure functions in the application which can be reused in any of the components.

Styles The styles are the style sheets of the application.

```
import { createSlice } from "@reduxjs/toolkit";
import { getDeviceList } from "../../api/device";

const initialState = {
    deviceList: [],
    deviceListLoading: false,
    deviceListError: null,
};

const deviceSlice = createSlice({
    name: "deviceSlice",
    initialState,
    reducers: {
        getDeviceListStart(state) {
            state.deviceListLoading = true;
            state.deviceListError = null;
        },
        getDeviceListSuccess(state, action) {
            state.deviceListLoading = false;
            state.deviceList = action.payload;
        },
        getDeviceListFailure(state, action) {
            state.deviceListLoading = false;
            state.deviceListError = action.payload;
        },
    },
});

export const {
    getDeviceListStart,
    getDeviceListSuccess,
    getDeviceListFailure,
} = deviceSlice.actions;

export const fetchDeviceList = () => async (dispatch) => {
    try {
        dispatch(getDeviceListStart());
        const deviceList = await getDeviceList();
        dispatch(getDeviceListSuccess(deviceList));
    } catch (error) {
        dispatch(getDeviceListFailure(error));
    }
};

export default deviceSlice.reducer;

```

Figure 5: The deviceSlice

```
1   import { configureStore } from "@reduxjs/toolkit";
2
3   const store = configureStore({
4       reducer: {
5           deviceSlice: deviceReducer,
6       },
7       middleware: (getDefaultMiddleware) => getDefaultMiddleware
    ({
8               serializableCheck: false,
9       })
10  });
11  export default store;
12
```

Figure 6: The store index

5.1.2 Back-end

The back-end use dotNet framework to develop, the language is C#.

During the development of the back-end, the overall of development is created the Clean Architecture for the back-end.

For creating the architecture, we need the following layers:

- Domain Layer:
- Application Layer: The application layer is the layer that contains the application logic, which is implemented the domain service interface and offer the service to the presentation layer, infrastructure layer and the test layer.
- Infrastructure Layer: The infrastructure layer is the layer that contains the implementation of the application service interface, there are not just one of the infrastructure but many, for example,
  - Database infrastructure: The database infrastructure is the layer which contains the implementation of the database service interface, it can be the DTO, the Repository , the Query, etc.
  - MQTT infrastructure: The MQTT infrastructure will contain the back-end MQTT client which will subscribe the MQTT topic and publish the MQTT topic though the MQTT broker to communicate with the device.
  - WebRTC's infrastructure: The WebRTC infrastructure is working for offer the WebRTC service which specific as the ICE config and TURN server.
- Test Layer: The test layer is the layer that contains the test code for all layers.

Domain Layer The domain layer is the heart of the software, it contains the business logic and the entities of the software.

The domain layer include two parts:

The entities of the software, for example, the user entity, the device entity, etc. See the figure 7.

```
1         public class Device{
2             public string Id { get; set; }
3             public string? SerialNumber { get; set; }
4             public string? ShortName { get; set; }
5             public Organization? OrganizationID  { get; set; }
6             public Group? GroupID { get; set; }
7         }
8
```

Figure 7: The Device entity

The domain service of the software, this part included the interface which identify the service which the application layer should be implemented to reach the operation of the domain.

It uses for limited the application layer to access the domain layer.

See the figure 8.

```
1         public interface IDeviceService
2         {
3             List<Device> GetDeviceList();
4             Device GetDeviceById(int id);
5             Device CreateDevice(Device device);
6             Device UpdateDevice(Device device);
7             Device DeleteDevice(int id);
8         }
9
```

Figure 8: Domain Layer Interface

Application Layer The application layer is the layer that implement the domain service interface and offer the service to the presentation layer, infrastructure layer and the test layer. See the figure 9.

Infrastructure Layer The fracture layer working for implement the external service interface. For the overall of the fracture layer, it can be the webAPI, the database repository, the MQTT, the WebRTC, etc.

The webAPI implement the webAPI service interface from the application layer. And the webAPI will offer the RESTful API to the presentation layer

```
1     public class DeviceService: IDeviceService{
2
3         private IDeviceRepository _deviceRepository;
4
5         public DeviceService (IDeviceRepository deviceRepository)
6         {
7             _deviceRepository = deviceRepository ?? throw new
      InvalidDataException("deviceRepository can not be null");
8         }
9
10        public List<Device> GetAllDevice()
11        {
12            return _deviceRepository.FindAll();
13        }
14
15        public Device CreateDevice(Device device)
16        {
17            return _deviceRepository.CreateDevice(device);
18        }
19
20        public Device ReadDevice(Device device)
21        {
22            return _deviceRepository.ReadDevice(device);
23        }
24
25        public Device UpdateDevicee(Device device)
26        {
27            return _deviceRepository.UpdateDevice(device);
28        }
29
30        public Device DeleteDevice(Device device)
31        {
32            return _deviceRepository.DeleteDevice(device);
33        }
34    }
35
```

language=CSharp

Figure 9: Application Layer Service

which is the webAPI controller. Also, the DTO, the Program.cs file as the entry point of the back-end service and config file.

The database repository implements the database service interface from the application layer. Out of the database repository, it also can be the database DTO, DbContext, Entities etc.

The MQTT implement the MQTT service interface from the application layer, which contain the MQTT client, the MQTT broker, the MQTT topic utility, etc.

Test Layer The test layer is the layer that offer the unit test for all the layers. Which means that the test layer reference all the layers but not necessary to implement the interface of the layers.

The unit test is testing the data type correct, the interface of the method have the correct parameters and the return value type is correct, also the functions and methods interact as pure function, which means that the function or method should not have the side effect.

Dependency Injection The dependency injection is the design pattern which use for inject the dependency of the class. This is the most important design pattern for the Clean Architecture.

5.1.3 DevOps

The core of DevOps process is always the CI/CD process, this process will control and manage the whole software development process.

In this project, we use the GitHub Actions as the CI/CD process. Therefore, we can easily config the main.yml file inside the `.github/workflow` folder.

5.2 Feature development

Since we create the overall structure of the application, we also have to implement the features that we want to have in the application. This section will describe the features that we have implemented in the application.

5.2.1 Use Case 1

Before we start the feature development, we can identify the use case to explicitly define the requirements for the feature.

The use case is defined as follows:

Use Case 1	**The user can get device list**
Actors	User
Preconditions	1. The user is logged in 2. The user is on the device page 3. The user has at least a device online
Basic Flow	The user will have the vision of the device list
Alternative Flows	null
Postconditions	The device list display on the device page

Table 1: Use Case 1

Following the use case, we can start the feature development.

5.2.2 Feature 1

Front-end To be able to display the data from the back-end, we need to make a request to the back-end. The request is made by using the `fetch` API.

In the `deviceComponet` we need use the customer hook `useGetAllDevices` to get the device list from the back-end.

After the request is made, we need to handle the response from the back-end. The response is a JSON object, we need to parse the JSON object to the DOM object.

And then, we need to display the device list on the device page.

Back-end To be able to response the request from the front-end, we need to implement the webAPI controller to handle the request.

And then, the webAPI controller will call the service to get the device list from the domain service.

The domain service will invoke the repository to get the device list from the database.

After the device list is got from the database, the domain service will return the device list to the webAPI controller.

The webAPI controller will return the device list to the front-end.

6 Conclusion

6.1 Result

Since the project is still in progress, the result is not available yet. So far, basic structure of this project has been built. But the most features are not implemented yet.

6.2 Discussion

As a single developer for this project, I am confident what I have done so far. And I can say I understand the most of the knowledge I have used in this project, which also means I can explain all the part of the project. But this project also relevant some of the complex knowledge which I have to continue to study and practice.

6.3 Future Work

The future work is to implement the rest of the features. Including the most important part which is the 'create session' feature.