Huy Anh Pham

Biti’s Hunter E-Commerce Admin Dashboard

Bachelor’s thesis

Information Technology

Bachelor of Engineering

2025

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author	Huy Anh Pham
Thesis title	Biti's Hunter
Year	2025
Pages	71 pages, 4 pages of appendices
Supervisor	Ville Kauppi

ABSTRACT

The purpose of this thesis was to build an admin dashboard for a fashion brand to effectively manage store operations.

The website was developed with both front-end and back-end using Typescript, MUI, Tailwind CSS as well as Redux and Redux Toolkit for state management, while NodeJS and ExpressJS were used as creating controllers, routes and seeding data from mock data file to the database PgAdmin 4- a database management tool for PostgreSQL.

The deployment architecture for this web application was built on AWS to deliver a secure, scalable, and efficient application infrastructure. The core setup included a VPC (Virtual Private Cloud) with both public and private subnets and Internet Gateway for external connectivity. Alongside with those, EC2, a computing platform with 750 instances for latest processors, storage, networking and RDS (Relational Database Service), and a database management tool was used to offer flexibility for customizing databases according to the user's needs.

The result of this thesis is an application which can be utilized by enterprises for managing their revenue and expenses, providing a comprehensive solution to manage the cash flow and operational efficiency.

Keywords: web application, full stack, database, amazon, dashboard

CONTENTS

1 INTRODUCTION

In today's digital age with constant advances and the development of the internet, businesses have realized the importance of reaching customers and interacting with them online.

Bitis's Hunter is a well-known fashion brand in Vietnam, appealing to the country's teenagers by promoting individuality, contemporary style, and collaborating with well-known influencers within the country. (Abbey, 2025). However, many customers still have only little information about the brand, especially within the European market. This study focuses on highlighting the technical aspects of building a robust and user-friendly platform to manage online shop.

The primary objective of the thesis is to conceptualize, design and develop the Biti's Admin Dashboard. The aim is to create a manageable web app that can control the store's properties such as income, products, and user feedback. Moreover, this web app also improves user experience with features to explore the products, user's detail, income and feedback from customers.

In general terms, this thesis introduces technologies that can be used to develop a complex web app.

The structure of this thesis is divided into 5 parts:

- introduction: the general dashboard webapp,
- theoretical background: the foundational concepts and technologies relevant to the study,
- implementation: description of the design and development of the webapp,
- result: description of the dashboard, and the challenges and difficulties that were encountered,
- conclusion: the outcome of the thesis.

2 THEORETICAL BACKGROUND

This section introduces fundamental concepts about TypeScript, NextJS, ExpressJS along with all other technologies that were employed in this study to give the reader a better understanding of the conceptual basis of the thesis.

2.1 TypeScript Fundamentals

TypeScript is a free and open-source programming language created and managed by Microsoft. It follows an object-oriented approach and enhances JavaScript by adding static typing and other advanced features. As a matter of fact, TypeScript cannot be interpreted directly in browser so a TypeScript Compiler, with the extension “tsc”, is required as TypeScript code will be compiled into plain JavaScript Code as in Figure 1. (Introduction to TypeScript, 2025.)

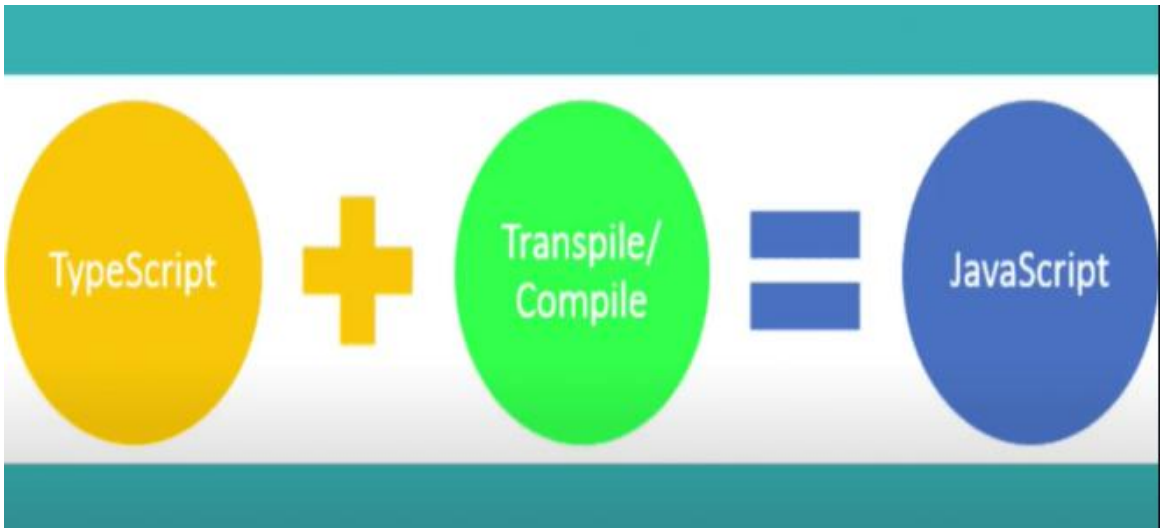


Figure 1: TypeScript compiler method (Introduction to TypeScript, 2025)

In the early 2012, TypeScript was popularly used within the developer community owing to since TypeScript is a superset of JavaScript which means all JavaScript code remains valid within TypeScript. It can also be understood that developers can seamlessly transition their JavaScript projects to TypeScript while benefiting from its additional features, for example, static typing and enhanced tooling. In other words, TypeScript builds upon JavaScript by adding additional features,

such as static typing, interfaces and advanced tooling, while still allowing developers to use existing JavaScript code without modification. (Derek, 2023.)

One of the key features of this Object-Oriented Programming (OOP) language is its type-checking. Although it requires more effort to code, it significantly improves code quality by preventing potential bugs and making the code more readable. Moreover, it can enable IDEs to offer a resourceful environment for spotting common errors in coding.

2.2 TAILWIND CSS

TailwindCSS is a utility framework that streamlines web development by offering a collection of pre-built classes. These classes help users to create custom designs without writing customs CSS classes, ensuring consistency, scalability, and efficiency during the development process. (Introduction to Tailwind CSS, 2024.)

Some of the key benefits of Tailwind CSS are smaller and more manageable built-in responsiveness, allowing developers to create mobile-friendly effortlessly layouts

2.3 NEXTJS Framework

Next.js is a framework based on React library, focusing on building full-stack web applications. It is suitable for creating interfaces, as well as back-end and other additional features and optimizations. This technology is built on top of React, however, it offers more powerful tools such as server-side rendering, static site generation and full-stack development. (Ahmed, 2023.)

In general, NextJS simplifies the development process by handling vital tools which are included in React such as bundling and compiling, enabling a smoother workflow, allowing developers to spend more time on constructing their application rather than setting configuration. (Ahmed, 2023.)

Consideration is its rendering techniques. It allows the user to build single page applications rendered either on the client or server side, which is not easy to do with only plain React. By default, all pages are server - side rendered. However, developers can easily choose the type of each page based on their needs. (Ahmed, 2023.)

When it comes to the back-end, users normally must set up two different repositories: one for the front-end and one for the back-end. On the other hand, Next.js provides an API layer powered by express making it possible to build an express app inside one repository instead of separating front - end and back - end to two different repositories. (Ahmed, 2023.)

Overall, Next.js offers a combination of efficiency, speed and flexibility, all promoted by community support, making a preferred alternative when it comes to choosing frameworks.

2.4 REDUX

Redux is, in simple terms, a JavaScript library with main features of predicting and maintaining global state management. It helps developers to write applications more consistently and in different environments such as clients, server or native. Two essentials properties of this library are React Redux and Redux Toolkit. (Rany, 2021.)

React Redux is the official React User Interface that binds layers for Redux, assists React components in reading data from Redux store and dispatches this data from the store to update its state. (Rany, 2021.)

On the other hand, Redux Toolkit handles data fetching and state management in building scalable and maintainable applications. RTK query is a powerful solution to simplify tasks and offers a set of utilities to make the application’s process more efficient. (Rany, 2021.)

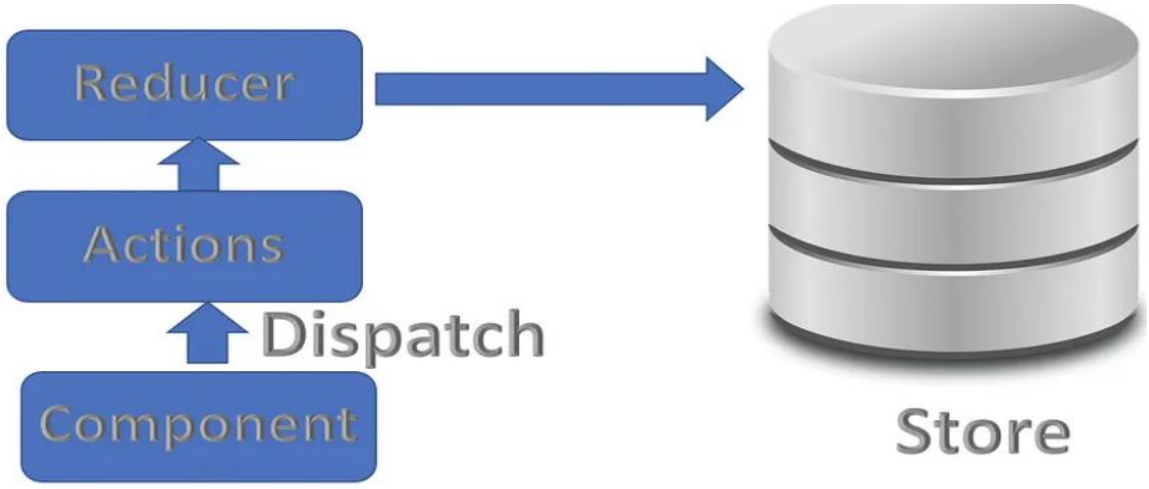


Figure 2: How RTK Query stores and conducts its data (ElHousieny, 2021)

Figure 2 shows the data flow starting from the “Component”, which then dispatches the “actions” when an event occurs such as “onClick” event. The “Reducer” then modifies the “Store”, which holds the state of the application. (Rany, 2021.)

In general, Redux helps developers to store the state of the variables within their applications by creating a process and following its procedures to interact with the store to prevent components from being updated or read randomly from the store.

2.5 NODEJS

NodeJS is an open-source, cross-platform runtime built on the V8 JavaScript engine. It operates on a single-threaded model, avoiding the need to create a new thread for each request. Moreover, NodeJS provides asynchronous I/O primitives in its library, enabling non-blocking operations while still supporting synchronous behavior when needed. (Viacheslav, 2024.)

When developers decide to access their database or file system, instead of blocking the thread or wasting CPU resources, NodeJS resumes the operations when the responses come back. (Viacheslav, 2024.)

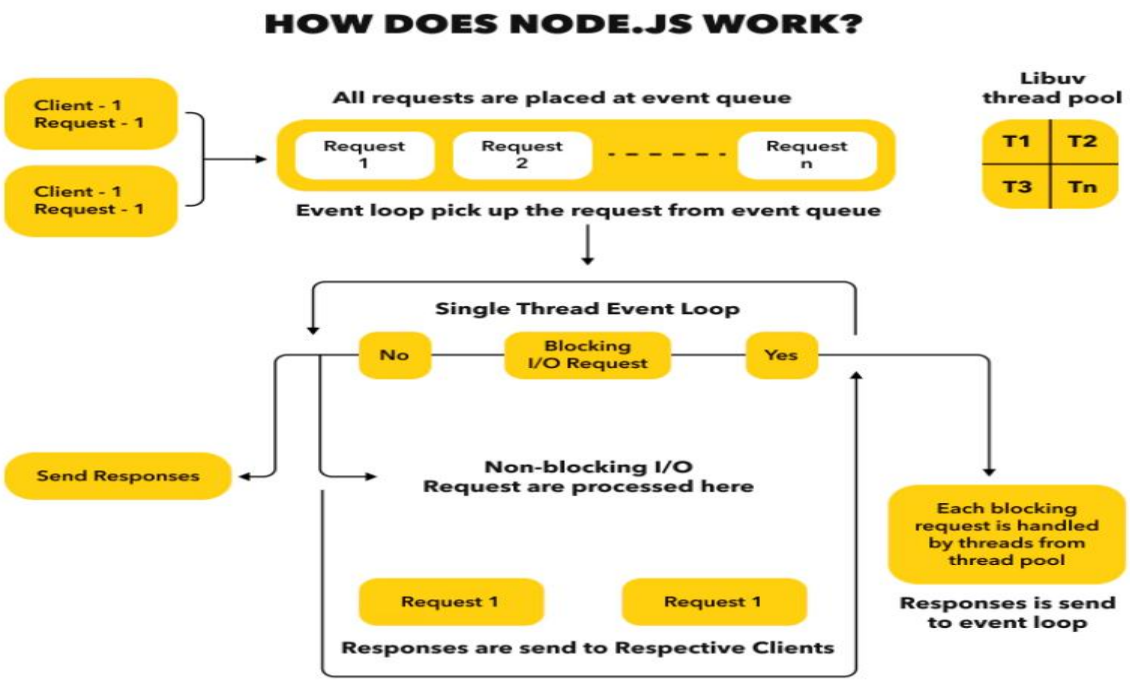


Figure 3: An example of NodeJS use scenario (Viacheslav, 2024)

With this utility, NodeJS can process thousands of concurrent connections with only a single server, without having to manage thread concurrency, which could contain a significant number of bugs. (Viacheslav, 2024.)

Overall, NodeJS stands out as a robust and versatile technology, continuing to empower developers in creating high-performance applications across diverse domains.

2.6 EXPRESS JS

Express JS was built on top of NodeJS, providing a list of features specific to building web and mobile applications. It helps to minimize the complexity of development of server-side applications, adding structure and organization to NodeJS applications. For example, Express JS offers simple functions which can be used to define routes rather than writing a much more complex code to process different URLs and requests. (Express.js Tutorial, 2024.)

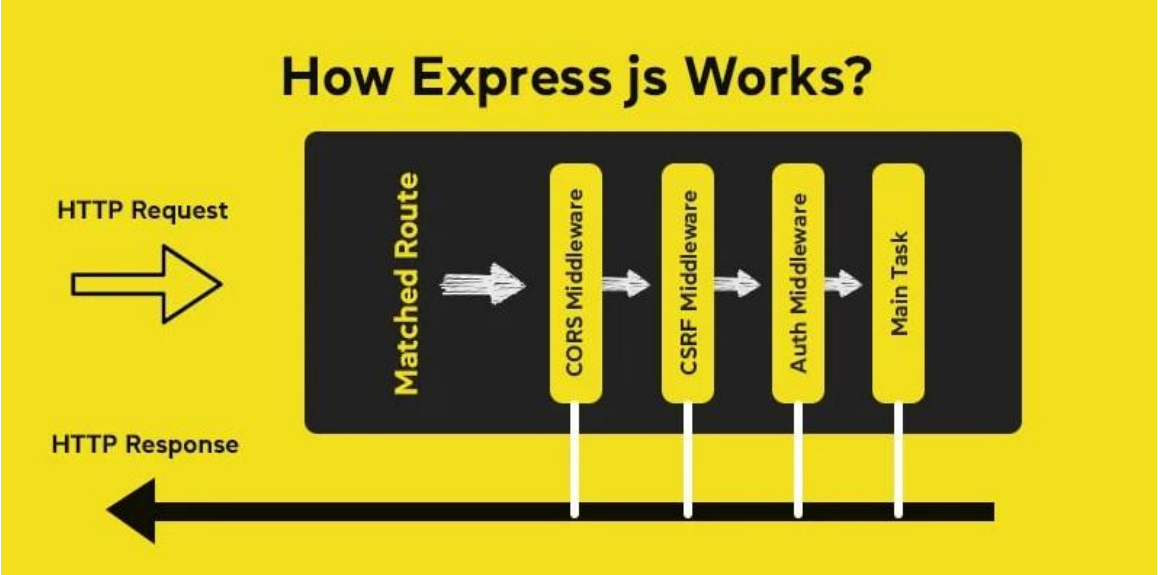


Figure 4: Architecture of ExpressJS request handling (Mayur, 2024)

Express JS is popularly known for features such as:

- fast and scalable server-side development,
- simple routing and middleware processing for web applications,
- support for building APIs, real-time applications, and single-page applications.

In summary, Express JS is a crucial tool for building web applications with Node JS. It simplifies development, provides flexibility, and is supported by a large community of developers.

2.7 PRISMA

Prisma ORM (Object Relational Mapping) is a next generation NodeJS, along with TypeScript for PostgreSQL, MySQL, SQL Server, SQLite, MongoDB, and CockroachDB. Its main features are:

- Prisma Client: An auto-generated query builder for NodeJS and TypeScript with type-safe query builder to tailor data.
- Prisma Migrate: A powerful data modeling and migration system to define the database schema.

Furthermore, Prisma Schema utility is the main method when setting up Prisma. It consists of the following features:

- Data sources: Specify the data in detail for databases to connect with Prisma ORM such as PostgreSQL.

- Generators: Specify the data model which the clients should follow.
- Data model: Define the data model for the application

Figure 5 shows an example of Prisma Schema.

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model User {
  id        Int      @id @default(autoincrement())
  createdAt DateTime @default(now())
  email     String   @unique
  name      String?
  role      Role     @default(USER)
  posts     Post[]
}
```

Figure 5: An example of Prisma Schema that specifies. (Prisma, n.d.)

Overall, Prisma is the newer generation of Object Relational Mapper designed to streamline database access.

2.8 POSTGRESQL

PostgreSQL is an open-source, object-relational database,using the SQL language with advanced features, assisting developers in application development and administrators in ensuring data integrity. To prevent conflicts with its core functionalities or leads to inefficient architectural choices, it needs to comply with SQL standards. (PostgreSQL, n.d.)

In 1986, at the University of California at Berkeley, the making of PostgreSQL started as part of the POSTGRES project and ever since it been actively under development for more than 35 years on the core platform. In time, it had proved its potential in architecture, reliability, data integrity, robust feature set and extensibility, having a large community of supporters to deliver performance. (PostgreSQL, n.d.)

The following four features of PostgreSQL help developers build applications, protects data integrity and implement fault-tolerant environments:

- Data Types
- Data Integrity
- Concurrency, Performance
- Reliability, Disaster Recovery

Along with PostgreSQL, a database management tool, PgAdmin which is used to help maintain the database activities, deployed as a web application by configuring the app to run in server mode. (PostgreSQL, n.d.)

However, PgAdmin and PostgreSQL are two completely different entity. In fact, PostgreSQL is a database engine following SQL standards and listen to the back-end server on a network port. In contrast, pgAdmin is a graphical user interface for database management which enables users to manipulate schema and data on instances within the PostgreSQL engines. (Moez, 2024.)

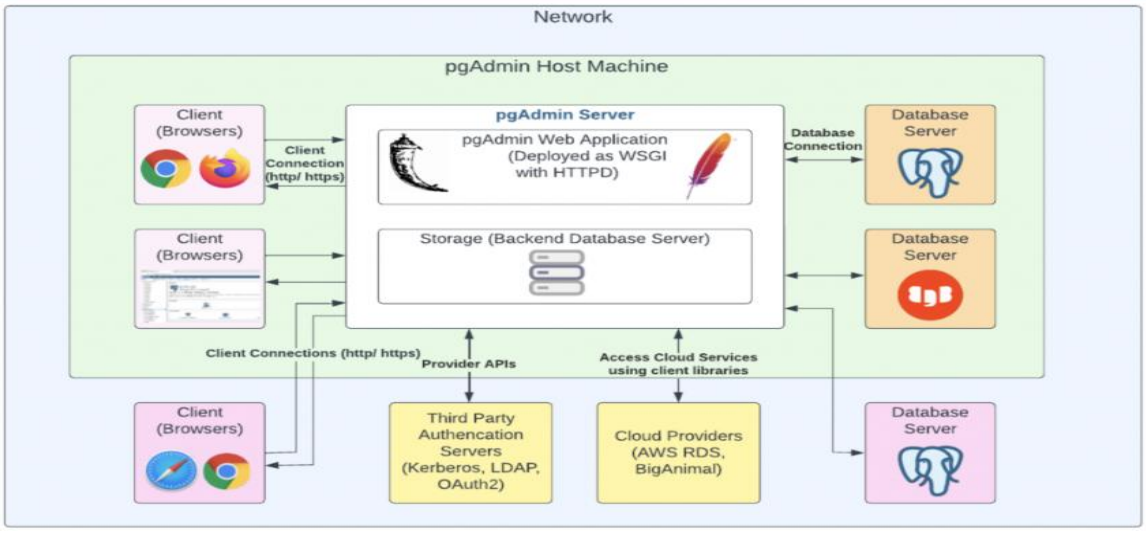


Figure 6: PgAdmin architecture map from PostgreSQL (Yogesh, 2022)

In general, PostgreSQL is highly scalable both in quantity of data as well as the number of users. It is extensible and can be used with pgAdmin as database management. (Moez, 2024.)

2.9 Amazon Web Service (AWS)

AWS is a comprehensive, cloud-based platform offered and developed by Amazon. It offers a variety of services such as IaaS (Infrastructure-as-a-service), PaaS (Platform-as-a-service) and SaaS (Software-as-a-service). (Paul, 2024.)

In 2002, AWS introduced its initial web services using an internal infrastructure designed for its retail operations. By 2006, it expanded into Infrastructure as a Service (IaaS). Amazon was also a pioneer in implementing the pay-as-you-go cloud computing model, allowing users to scale computing power, storage, and throughput based on demand. (Paul, 2024.)

Moreover, this cloud-platform provides various services regarding in networking, storage, middleware, IOT and other software tools through server farms. Owing to this, clients do not need to worry about managing, scaling, and maintaining their applications. One of the most popular and fundamental services offered by AWS is Amazon Elastic Compute Cloud (EC2), allow users to have their own a virtual cluster of computers, which can also interact with REST APIs, Command Line Interface or AWS console. (Paul, 2024.)

Due to its array of services offering, AWS can solve many IT issues and needs. The most beneficial of moving to a cloud environment is that it allows the organization to save money on physical data centers since AWS will run all the services, relieving customers from having to manage their own system. (Paul, 2024.)

2.9.1 AMPLIFY

AWS Amplify includes set of tools and services that provide a friendly user-interface for software developers who want to deploy and build scalable mobile or web applications quickly and easily. It offers features such as authentication, API, storage and hosting, all data is saved and able to be deployed to AWS cloud. (Servifyspheresolutions, 2024.)

Generally, with Amplify, developers can focus on building their applications without needing to worry about the underlying infrastructure or maintenance for their projects. (Servifyspheresolutions, 2024.)

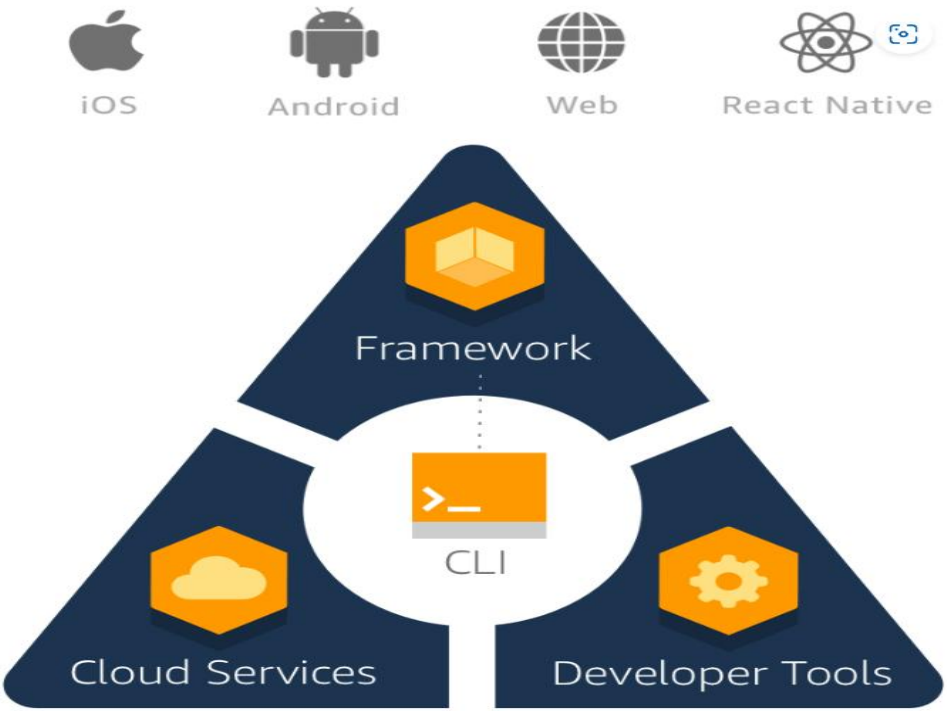


Figure 7: AWS Amplify services (Pushkar, 2021)

There are four core concepts in Amplify:

- Amplify CLI: A command-line tool for creating and managing AWS resources, with a user-friendly UI for configuring and deploying backend services.
- Libraries: JavaScript, iOS, and Android libraries for seamless interaction with AWS cloud services
- UI Components: Pre-built UI components for React,Vue, and Angular to speed up development.
- Hosting: Fully managed web hosting with support for GitHub, GitLab, and Bitbucket.

2.9.2 SIMPLE STORAGE SERVICE (AWS S3)

AWS S3 is a highly scalable object storage service that provides exceptional data availability, security, and performance via a web service interface. It operates on the same robust storage infrastructure which powers the Amazon's e-commerce platform. It can store all types of objects, allowing users to save more complex objects such as projects, internet applications, backup files. It also provides users

with manual features such as optimization, organization, and configuration of the data based on specific requirements. (Rahul, 2024.)

2.9.3 Amazon VPC

Virtual Private Cloud (VPC) is a cloud computing platform, offering users a virtual private cloud which acts as an isolated network in the cloud. Figure 8 shows an example of VPC.

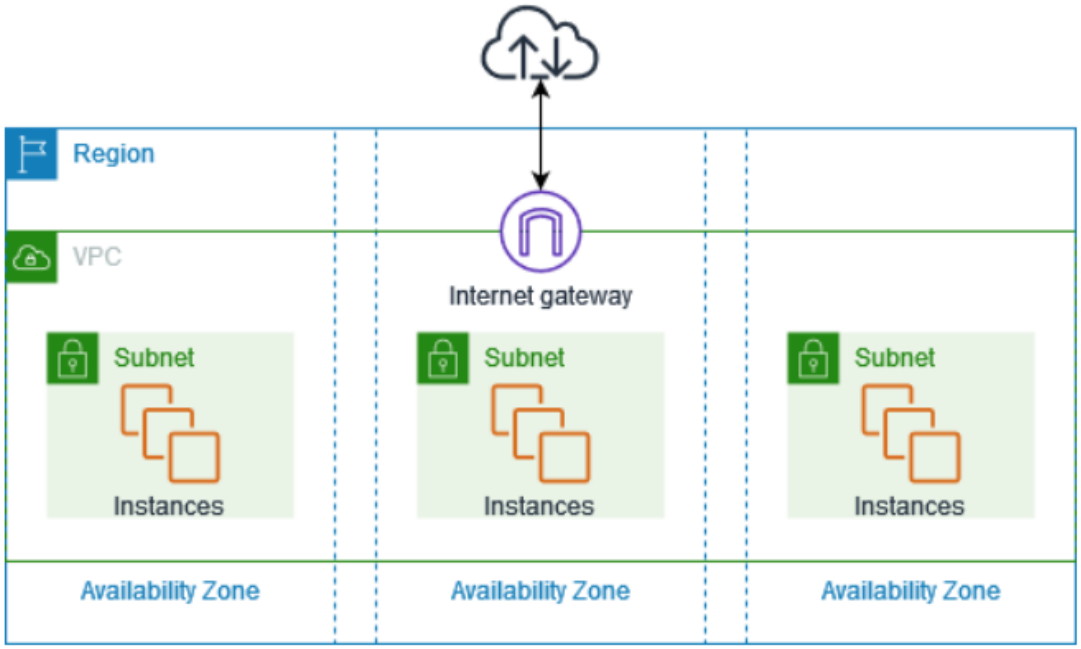


Figure 8: An example of VPC being used with multiple availability zone (AWS, n.d.)

In order to have a proper knowledge of VPC, architecture of a properly functioning VPC is demonstrated. VPC includes of a variety of features such as Internet Gateway, Load Balancer and Subnets. All these services are needed under a VPC in order to build a private , traditional network environment.

Subnet is a defined range of ip addresses within a given network. Subnetting optimizes network traffic by allowing data to travel more efficiently, reducing the need to pass via unnecessary routers to reach the destinations. Users can add as many subnets as they need in one availability zone. However, each subnet must only be in one certain zone. A public subnet is required within the Subnet

Association in Internet Gateway to gain access to the internet while the private subnet will only be used within the VPC network such as connection between database and back-end server. (Simplilearn, 2024.)

Internet Gateway (IGW), one of the resources offered by EC2, enables to access the internet, but one VPC cannot have more than one IGW.
A route table includes a variety of rules which help to direct the network traffic.

Some examples of scenarios where we need VPC:

- Hosting Websites**:** VPC supports public websites and single-tier web applications with security and scalability.
- VPC Peering**:** Enables secure connectivity between web servers, application servers, and databases.
- Traffic Management**:** Controls inbound and outbound traffic with strict security rules.

2.9.4 Amazon Elastic Cloud (EC2)

AWS EC2 (also known as Elastic Compute Cloud) is a secure, resizable, and scalable web service that offers virtual machines pre-configured with a variety of operating systems such as Linux, Windows. AWS handles the underlying infrastructure, allowing users to launch or terminate the instance as needed. (Servifyshperesolutions, 2024.)

To deploy EC2, the user must first setting up the EC2 instance in a secure network such as VPC along with Subnets and Security groups. Figure 9 demonstrates an example of EC2 being deployed in VPC. (What is…, 2024.)

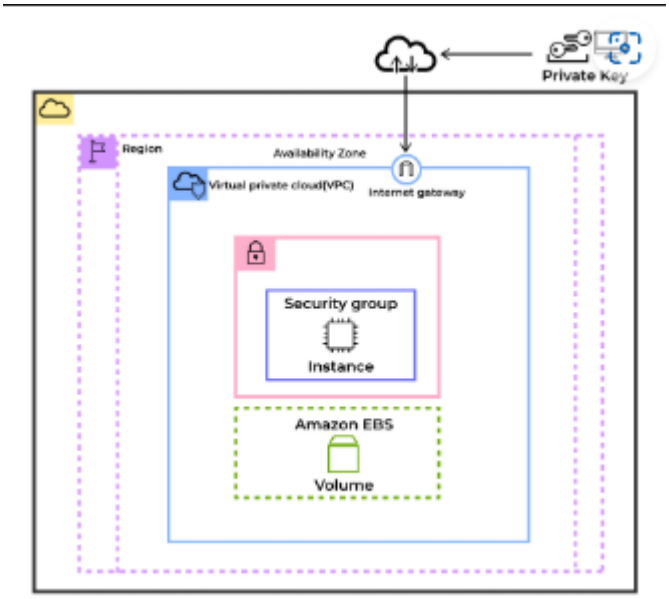


Figure 9: EC2 architecture with its relevant services from AWS cloud computing (What is…, 2024)

Amazon EC2 is widely used for various applications. It allows users to deploy applications without managing the underlying infrastructure, as AWS handles it. EC2 also supports training and deploying machine learning models with high-speed networking (up to 400 Gbps) and optimized storage for cost efficiency. Additionally, it enables hybrid cloud environments by connecting with on-premises databases for seamless integration. (Servifyshperesolutions, 2024.)

Furthermore, EC2 offers users a virtual computing service which can be used according to users' demand. Amazon EC2 has a set of default Amazon Machine Images (AMI) that support different operating systems along with AMI and some built-in resources such as Random Access Memory (RAM) or Read-only Memory (ROM). (Servifyshperesolutions, 2024.)

Figure 10: A series of OS supported by AWS EC2 (What is…, 2024)

2.9.5 Amazon Relational Database Service

Amazon Relational Database Service (RDS) is a service which is managed and maintained completely by AWS, offering a wide range of database engines such as MySQL, PostgreSQL, Oracle and SQL Server. (Jon, 2021.)

In order to smoothly run a database, AWS implements its own security services which will encrypt the data with the help of its Identity Access Management (IAM). Also, the backup of the data as well as the infrastructure will be taken care by AWS since it has the pre-set configuration. (Jon, 2021.)

Originally, database management used to be a scattered range of services due to the fact that its direction of traffic was usually from the webserver to the application server, finally reaching the destination which is the database. With this sophisticated procedure, maintaining database would require a large resource, the kind that could be provided by AWS RDS since it includes most of the resources from EC2 cloud computing platform to Domain Name System (DNS). Every part within the RDS architecture has its own separate set of features completely unique from others. (Jon, 2021.)

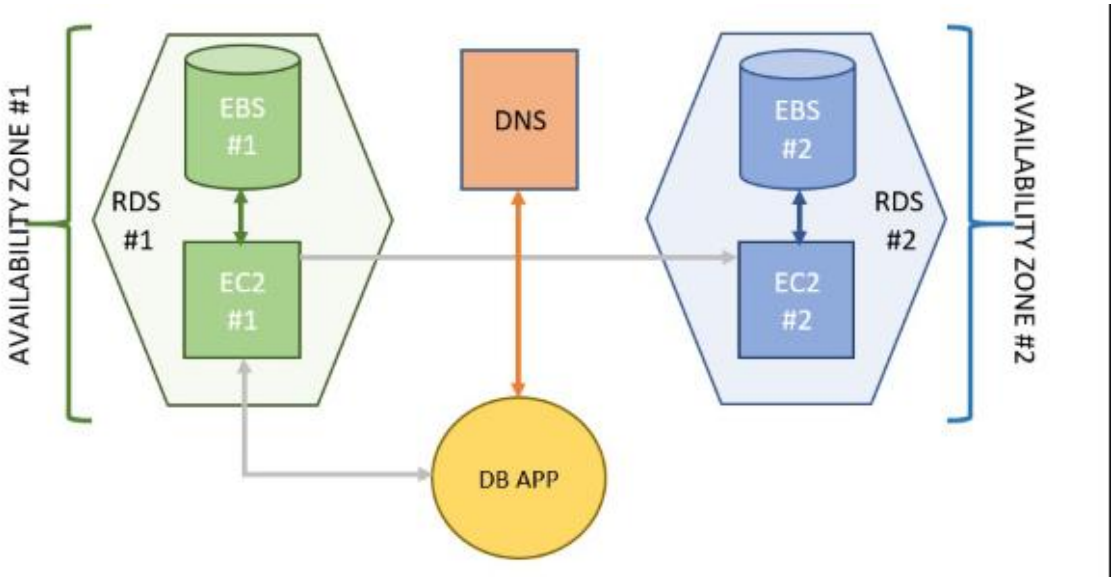


Figure 11: RDS architecture in AWS, with two different zones and DNS (Amazon RDS – Intro…, 2025)

Amazon RDS provides robust features for database management, including availability, security, scalability, and performance. Automated backups and user-driven snapshots enhance availability, allowing easy recovery and database sharing across AWS accounts. Security is enforced through password restrictions for admin access and encryption managed via AWS Key Management Service. Scalability includes **horizontal scaling** to handle high traffic by replicating resources and **vertical scaling** for upgrading storage and processing power when demand increases. Performance is optimized with SSD-backed storage, offering cost-effective general-purpose SSDs for broad workloads. RDS instances are managed by virtual servers and each of these instances comes with pre-configured hardware and software to support the client's choice in database engine such as MySQL or PostgreSQL. (Amazon RDS – Intro…, 2025.)

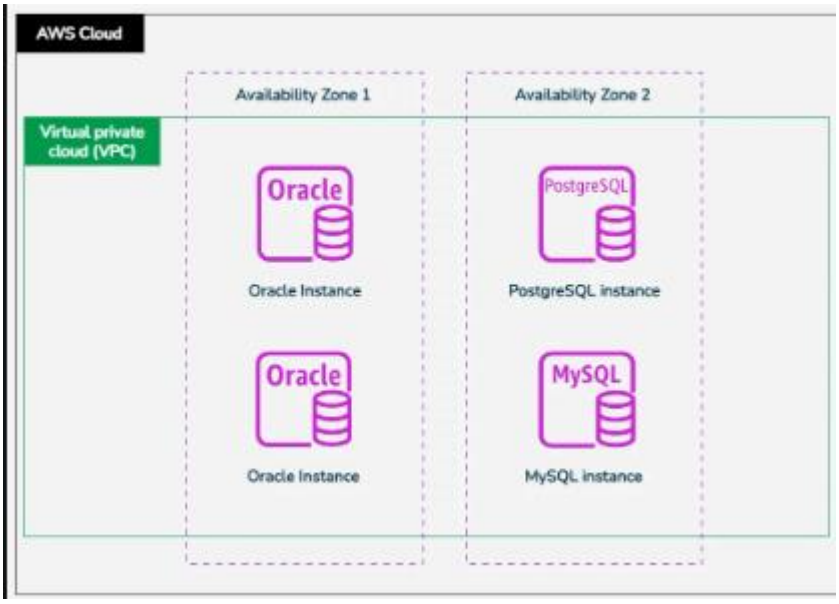


Figure 12: Different database engines in VPC (Amazon RDS – Intro…, 2025)

Overall, RDS provides useful services with regards to managing databases within the cloud environment and automating routine tasks such as backups, patching, and scaling. This automation enables developers and administrators to concentrate on application development rather than database maintenance. (Amazon RDS – Intro…, 2025.)

2.9.6 Amazon Application Program Interface Gateway (AWS API Gateway)

The API Gateway acts as a management tool that bridges users and microservices, enabling the creation, publishing, security, maintenance, and monitoring of APIs at any scale. Serving as an intermediary between clients, applications, and backend services, it routes API requests to the appropriate services while handling tasks such as authentication, authorization, and traffic management. (Amazon Web Service-Introduction to API Gateway, 2023.)

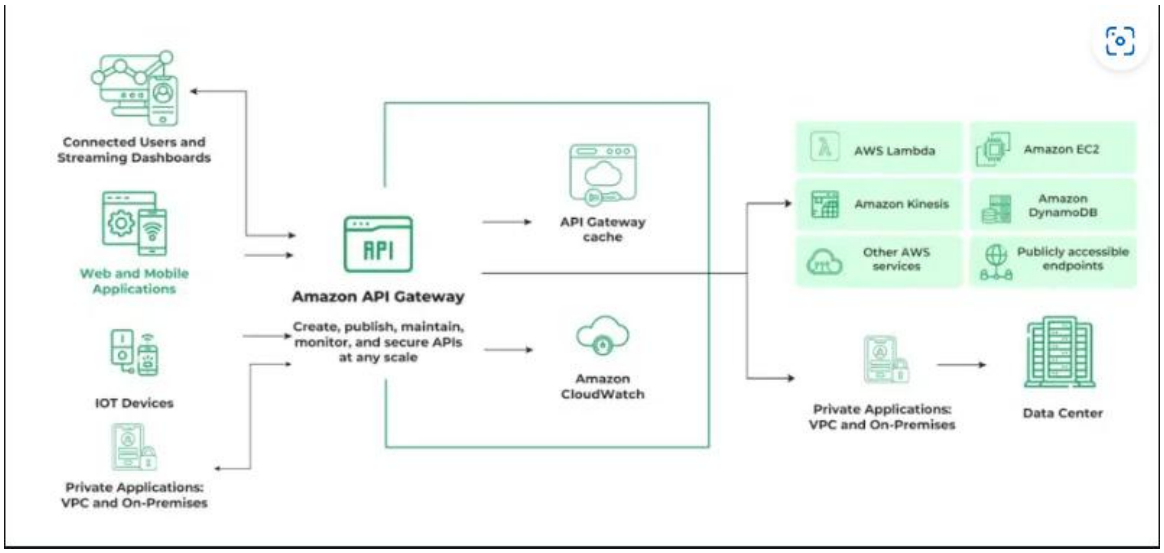


Figure 13: A diagram of how AWS API works (Amazon Web Service – Introduction to API Gateway, 2023)

AWS API Gateway supports four types of APIs. The first is Representational State Transfer (REST), which is an architecture style that imposes specific constraints for building scalable web services. (Amazon Web Service-Introduction to API Gateway, 2023.)

Simple Object Access Protocol (SOAP) is a network protocol with its main feature to exchange structured data between nodes. It relies on XML for message formatting and uses it on top of application-layer protocols. SOAP enables seamless communication across different platforms, programming languages and operating systems. (Amazon Web Service-Introduction to API Gateway, 2023.)

GraphQL APIs is an open-source query language and runtime designed for APIs, enabling efficient data retrieval and manipulation by allowing clients to precisely request the required data from existing sources.

WebSocket APIs enable real-time, multi-directional communication over the web using WebSocket protocol, which is Hypertext Transfer Protocol (HTTP). This framework facilitates the development of interactive applications and can be implemented using different technologies. (Amazon Web Service-Introduction to API Gateway, 2023.)

3 IMPLEMENTATION

3.1 PROJECT GOAL AND OVERVIEW OF WEBSITE APPLICATION

In this study, a web application was to help administrators manage and monitor their business operations without needing to be physically present at the facility. The BitisHunter admin dashboard was used to provide an overview of the total value, money flow, and customer feedback for the business.

The application is primarily intended for users with administrative privileges, allowing them to edit, update, and monitor all company progress. Administrators can also create new products based on their preferences within the application and they will automatically be added to the company’s internal database.

In order to support multiple administrators, the application will be deployed on AWS instead of being hosted locally, enabling access for administrators from different locations. Each premise can use the application with distinct accounts and databases.

3.2 REQUIRED PACKAGES AND DEPENDENCIES

Figure 14, 15 shows the required packages and dependencies for front-end and back-end development.

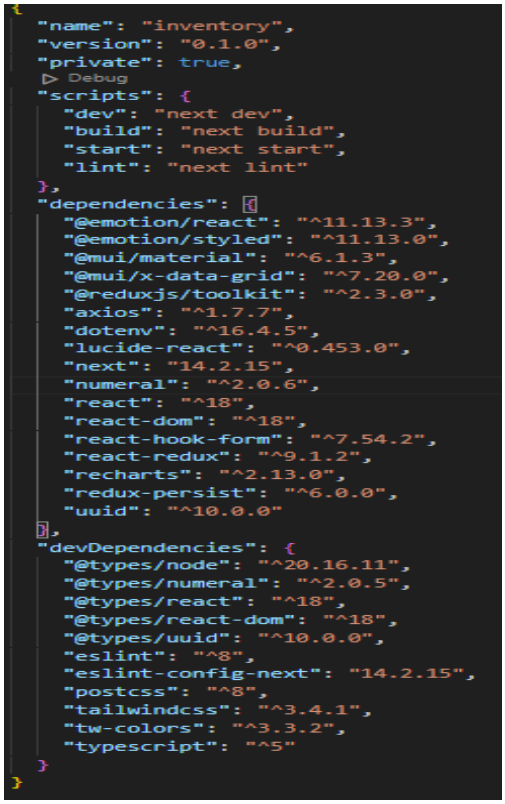


Figure 14: Required packages and dependencies for front-end

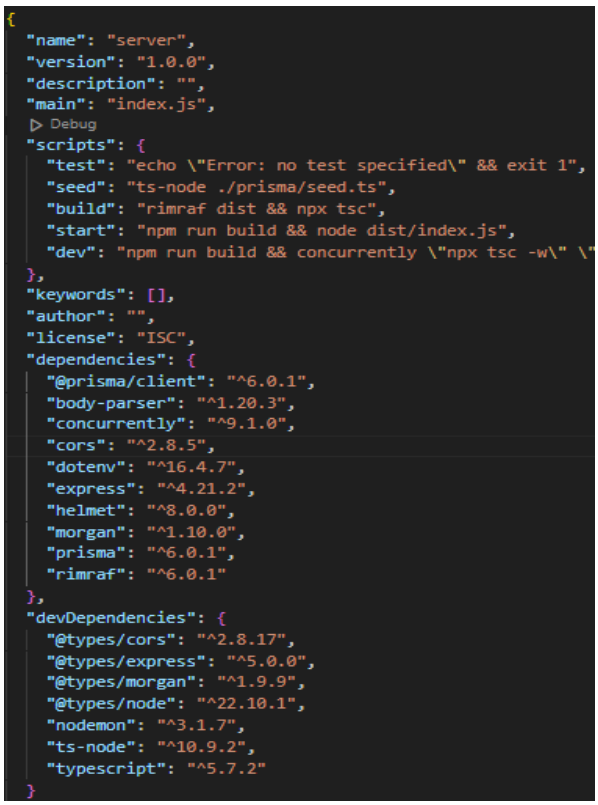


Figure 15: Required dependencies for back-end

Figures 14 and 15 introduce the configuration file mainly for Node.js, Next.js and TypeScript-based projects which are required for the application. The “package.json” file includes scripts for building, running, and developing the application, as well as managing the Prisma Object - Relational Mappers and database seeding. Moreover, the package.json file also contains dependencies such as express, prisma, dotenv and other different libraries. Overall, it serves as the backbone for managing the project’s configuration.

3.3 SETTING UP TAILWIND CSS FOR LIGHT AND DARK MODE

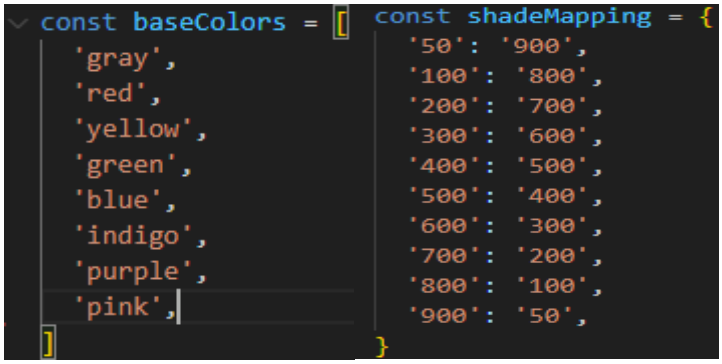
In order to create a custom “light-dark” theme and implement the theme with redux, there are changes which need to be configured within tailwindCSS’s setting file as shown in Figure 16 to include theme support for dark and light mode for the web application using “tw-colors” library.

```
+  const { createThemes } = require('tw-colors');

   module.exports = {
      content: ['./src/**/*.{astro,html,js,jsx,md,mdx,svelte,ts,tsx,vue}'],
      theme: {
         extends: {
-           colors: {
-              'primary': 'steelblue',
-              'secondary': 'darkblue',
-              'brand': '#F3F3F3',
-              // ...other colors
-           }
         },
      },
      plugins: [
+        createThemes({
+           light: {
+              'primary': 'steelblue',
+              'secondary': 'darkblue',
+              'brand': '#F3F3F3',
+              // ...other colors
+           }
+        })
      ],
   };
```

Figure 16: An example of tw-colors scenario.

Two constant variables are created, one is an array for the main colors and other is the shade mapping.



```
const baseColors = [
  'gray',
  'red',
  'yellow',
  'green',
  'blue',
  'indigo',
  'purple',
  'pink',
]
```

```
const shadeMapping = {
  '50': '900',
  '100': '800',
  '200': '700',
  '300': '600',
  '400': '500',
  '500': '400',
  '600': '300',
  '700': '200',
  '800': '100',
  '900': '50',
}
```

Figure 17: Shade mapping

In figure 17, there are two variable, one for colors and other for the colors's shade level. The function "generateThemeObject" in Figure 18 maps light theme shades (key) to their corresponding dark theme shades (value). Here is an example of how the function replaces the shade level of color gray. If the light theme's gray is 50, dark theme's gray will be 900 when changed to the dark theme. If the dark theme's gray is 900, light theme's gray will be 50 when changed to the light theme.

```
const generateThemeObject = (colors: any, mapping: any, invert = false) => {
  const theme: any = {};
  baseColors.forEach((color) => {
    theme[color] = {};
    const entries = Object.entries(mapping);
    entries.forEach(([key, value]: any) => {
      const shadeKey = invert ? value : key;
      theme[color][key] = colors[color][shadeKey];
    });
  });
  return theme;
}
```

Figure 18: Function generateThemeObject

Firstly, this function uses three parameters, colors, mapping and invert. The first parameter,"colors", is the array base colors, "mapping" variable is the "key-value" for colors and constant "invert" is a Boolean value. An empty array variable is needed to hold colors families. With the provided baseColors, the function iterates and initializes an empty object to store its shades. Basically, it contains

colors as its property for the “theme” constant. Figure 19 shows an example of the variable.

```
const theme = {
  gray: { '50': '#f9fafb', '100': '#f3f4f6', ... },
  red: { '50': '#fee2e2', '100': '#fecaca', ... },
  yellow: { '50': '#fefce8', '100': '#fef3c7', ... },
  ...
};
```

Figure 19: An example of theme constant

Later, variable “entries” used Object.entries() method , which retrieves an array of an object’s enumerable pairs such as key and value for the “shadeMapping” variable . The result is an array of key-value pair like in Figure 20.

```
[['50', '900'], ['100', '800'], ...]
```

Figure 20: An example results of Object entries method

Figure 21 shows an example of the output. The first line is for the light theme and the second line is the dark theme which is transferred to the darker gray theme.

```
theme['gray']['50'] = colors['gray']['50']; // For light theme
theme['gray']['50'] = colors['gray']['900']; // For dark theme
```

Figure 21: Result of dark theme and light theme

lightTheme stores the colors for the light mode, while dark theme holds dark colors for the dark mode in the application.

```
const lightTheme = generateThemeObject(colors, shadeMapping);
const darkTheme = generateThemeObject(colors, shadeMapping, true);
```

Figure 22: lightTheme and darkTheme variable

According to instructions from tw-colors and Next.js configuration for tailwind, custom colors is passed to the createThemes functions as its argument.

```
const themes = {
  light: {
    ...lightTheme,
    white: '#ffffff'
  },
  dark: {
    ...darkTheme,
    white: colors.gray['950'],
    black: colors.gray['50']
  }
}

const config: Config = {
  darkMode: 'class',
  content: [
    "./src/pages/**/*.{js,ts,jsx,tsx,mdx}",
    "./src/components/**/*.{js,ts,jsx,tsx,mdx}",
    "./src/app/**/*.{js,ts,jsx,tsx,mdx}",
  ],
  theme: {
    extend: {
      colors: {
        background: "var(--background)",
        foreground: "var(--foreground)",
      },
    },
  },
  plugins: [createThemes(themes)],
};
```

Figure 23: constant themes which are assigned to createThemses as a parameter

3.4 SETTING UP REDUX CONFIGURATION

Figure 23 shows the setup for the storage function which specifies the storage engine to be used for persisting the state.

Function “createNoopStorage”, as shown in figure 24, prevents erros if there is no localStorage available. After that, the storage uses createWebStorage from redux-persist which wraps around localStorage.

```
const createNoopStorage = () => {
  return {
    getItem() {
      return Promise.resolve(null);
    },
    setItem(value: string) {
      return Promise.resolve(value);
    },
    removeItem() {
      return Promise.resolve();
    },
  };
};

// If we are in the server, we use a noop storage, else we use local storage
const storage =
  typeof window === "undefined"
    ? createNoopStorage()
    : createWebStorage("local");
```

Figure 24: Noopstorage function for redux

Next, a new variable is created to implement redux slice from reduxjs/toolkit with its creteSlice function. Its purpose is to manage two global UI states which are isSideBarCollapsed and isDarkMode.

```
export const globalSlice = createSlice({
  name: "global",
  initialState,
  reducers: {
    /**
     * Function to set the sidebar collapsed state
     */
    setIsSidebarCollapsed: (state, action: PayloadAction<boolean>) => {
      state.isSideBarCollapsed = action.payload;
    },

    /**
     * Function to set the dark mode state
     */
    setIsDarkMode: (state, action: PayloadAction<boolean>) => {
      state.isDarkMode = action.payload;
    },
  },
});
```

Figure 25: Redux slice to control state of sidebar and darkmode

A short description of Redux's slice is that it defines a section of the application's state by accepting an initial state and a set of reducer functions and store them to an immutable storage. It automatically generates actions creators and types corresponding to the reducers, making it a standardized and efficient way to write Redux logic.

Furthermore, reducer is an object containing Redux "case reducer" functions which intends to handle a specific action type. However, there is a rule for "reducer" functions that the reducer function should only calculate the new state value according to the state and action in the arguments.

The initialState is false for both of darkMode and sideBarCollapsed like in the Figure 26.

```
const initialState: InitialStateType = {
  isSideBarCollapsed: false,
  isDarkMode: false,
};
```

Figure 26: InitialState of both sidebar and darkmode

Finally, both these actions are dispatched in components, and the reducer is exported as globalSlice.reducer for use in the Redux store.

Next, to manage the query and configured endpoints, RTK Query (Redux Toolkit Query) is needed to manage API requests and caching efficiently in the application.

```
export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: process.env.NEXT_PUBLIC_API_BASE_URL }),
  reducerPath: "api",
  tagTypes: ["Dashboard", "Product", "Users", "Expenses"],
  endpoints: (build) => ({
    // API fetching dashboard data with RTK Query
    getDashboard: build.query<Dashboard, void>({
      query: () => "/dashboard",
      providesTags: ["Dashboard"],
    }),
    getProducts: build.query<Product[], string | void>({
      query: (search) => ({
        url: "/products",
        params: search ? { search } : {},
      }),
      providesTags: ["Product"],
    }),
    createProduct: build.mutation<Product, newProduct>({
      query: (newProduct) => ({
        url: "/products",
        method: "POST",
        body: newProduct,
      }),
      invalidatesTags: ["Product"],
    }),
    getUsers: build.query<Users[], void>({
      query: () => "/users",
      providesTags: ["Users"],
    }),
    getExpenseByCategory: build.query<ExpenseByCategorySummary[], void>({
      query: () => "/expenses",
      providesTags: ["Expenses"],
    }),
  }),
});
```

Figure 27: RTK query managing api endpoints

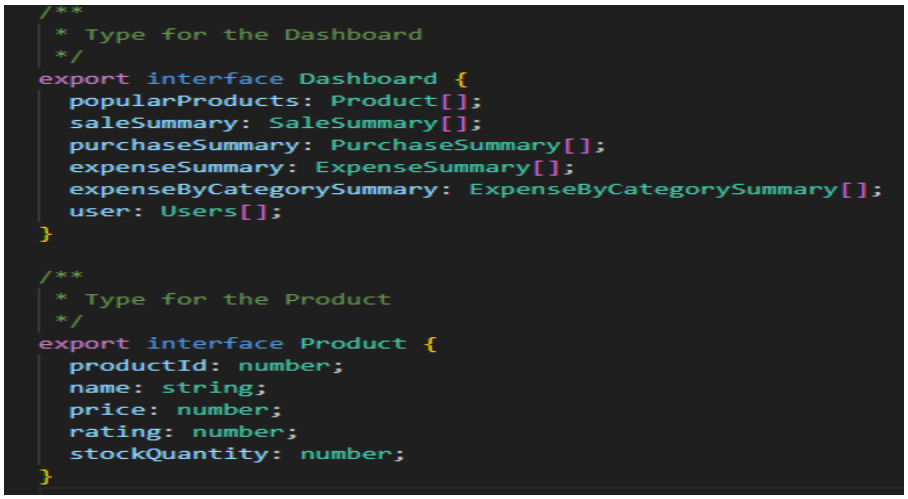
createApi is one of the core functionalities of RTK Query. It allows users to define a set of endpoints that describe how to fetch data from backend APIs and other async sources, including the configuration of how to retrieve data and manipulate data.

ReducerPath is a unique key that the service is mounted in the store within the application such as if a user invokes “createApi” multiple times, an unique value is required each time, and the default value is “api”.

TagType is an array of string tag type names. It is optional, however, since the application handles with multiple endpoints, and defining them would be a better choice so that they could be used for caching and invalidation. Also, it is required

to have their “provideTags” and enabled them with “invalidateTags” when configuring endpoints.

Endpoints are a set of operations with two different types: query and mutation.

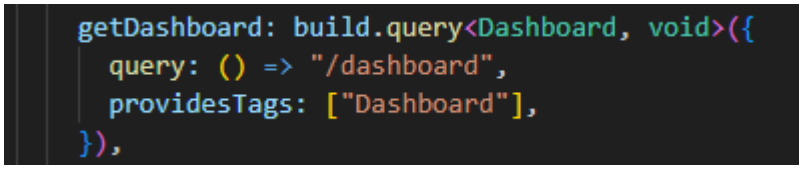


```
/**
 * Type for the Dashboard
 */
export interface Dashboard {
  popularProducts: Product[];
  saleSummary: SaleSummary[];
  purchaseSummary: PurchaseSummary[];
  expenseSummary: ExpenseSummary[];
  expenseByCategorySummary: ExpenseByCategorySummary[];
  user: Users[];
}

/**
 * Type for the Product
 */
export interface Product {
  productId: number;
  name: string;
  price: number;
  rating: number;
  stockQuantity: number;
}
```

Figure 28: Some interface required for the type safety

The interfaces presented in Figure 28 are to ensure type safety and autocomplete support when working with API responses as well as Redux state.



```
getDashboard: build.query<Dashboard, void>({
  query: () => "/dashboard",
  providesTags: ["Dashboard"],
}),
```

Figure 29: API endpoint

In Figure 29, the endpoint “dashboard” fetches the dashboard’s data which is corresponding to Dashboard interface and the “provideTags”, enabling for automatic caching and re-fetching if any mutation invalidates this tag. Most of the endpoints are similar, but the product endpoint, is a mutation for POST method.

```
createProduct: build.mutation<Product, newProduct>({
  query: (newProduct) => ({
    url: "/products",
    method: "POST",
    body: newProduct,
  }),
  invalidatesTags: ["Product"],
}),
```

Figure 30: API endpoint mutation for createProduct

Since the method was POST, instead of “build.query”, the query uses build.mutation for a mutation - a request that modifies data for POST, PUT and DELETE method.

Finally, all the queries are extracted and exported to all the hooks from the api slice, using RTK Query and are used within the React components.

```
export const {
  useGetDashboardQuery,
  useGetProductsQuery,
  useCreateProductMutation,
  useGetUsersQuery,
  useGetExpenseByCategoryQuery,
} = api;
```

Figure 31: Exporting hooks to fetch and modify data

```
const persistConfig = {
  key: "root",
  storage,
  whitelist: ["global"],
};

const rootReducer = combineReducers({
  global: globalReducer,
  [api.reducerPath]: api.reducer,
});

const persistedReducer = persistReducer(persistConfig, rootReducer);

export const makeStore = () => {
  return configureStore({
    reducer: persistedReducer,
    middleware: (getDefaultMiddleware) =>
      getDefaultMiddleware({
        serializableCheck: {
          ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
        },
      }).concat(api.middleware),
  });
};
```

Figure 32: Finishing up redux configuration

ReduxPersist is a library that helps to save the Redux store in storage such as localStorage, sessionStorage so it is available even when the page is reloaded. The key “root” defines the key name for the storage. The property “storage” is the storage variable defined above in Figure 23, in the section of setting up dark mode in tailwindCSS, which is the location of the saved state.

RootReducer variable using combineReducers for globalReducer (for the sidebar and darkMode functions) and API-related state (the data fetch from RTK Query).

PersistedReducer is a modified version of the root reducer which can automatically store and rehydrate the persisted state from the selected storage. This will add persistence to the global slice of the Redux state and store the global state in the selected storage, making sure that even if the page is refreshed, the state is saved.

Finally, the function makeStore configures a Redux store with a persisted reducer, attempting to customize the middleware using redux-thunk. Later, the function set “persistedReducer” as the main reducer, ensuring that specific parts of the state were saved across the page. Moreover, it customizes the middleware by applying “getDefaultMiddleware” while disables state checks for some Redux Persist actions since , by default, Redux Toolkit enforces serializability checks to ensure state and actions are correctly stored. The function appends “api.middleware” from RTK Query, ensuring fetching, caching and automatic are up-to-date within Redux store.

In order to utilize the function “makeStore”, a client component is needed to initiate redux store and shared with the application using React-Router Provider Component.

```
/**
 * Store Provider component
 *
 * @param children - The children to render
 *
 * @returns The Store Provider component
 */
export default function StoreProvider({
  children,
}: {
  children: React.ReactNode;
}) {
  const storeRef = useRef<AppStore>();
  if (!storeRef.current) {
    // Create the store instance the first time this renders
    storeRef.current = makeStore();

    // Refetching data, keeping data in sync with server
    setupListeners(storeRef.current.dispatch);
  }

  return <Provider store={storeRef.current}>{children}</Provider>;
}
```

Figure 33: StoreProvider component with client – side

Furthermore, function “makeStore” is saved into our main layout file.

```tsx
/**
 * Dashboard Wrapper component
 *
 * @param children - The children to render
 *
 * @returns The Dashboard Wrapper component
 */
const DashBoardWrapper = ({ children }: DashBoardWrapperProps) => {
  return (
    <StoreProvider>
      <DashboardLayout>{children}</DashboardLayout>
    </StoreProvider>
  );
};
```

Figure 34: StoreProvider component within the application's main layout

Overall, all the configuration for Redux had been successfully implemented.

3.5 Back-end Implementation

3.5.1 Prisma's Schema and Database Design

Figure 35 illustrates a database schema to manage models such as products, sales, purchases, expenses, and users. The Products table and others contain each product's detail like name, price, rating, quantity and stock quantity. Each table has a primary key (unique key) to identify its record. For instance, prodcutId in the Products table and saleId in the Sales table.

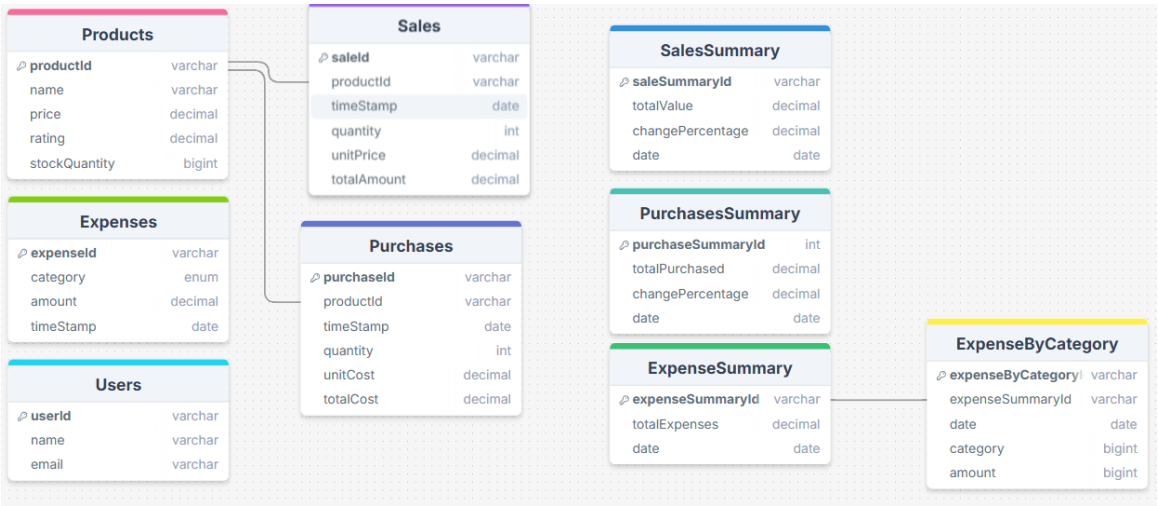


Figure 35: Architecture of database

Moreover, “foreign keys” also known as (FK) create relationships between tables. In the Sales’s table, “productid” is a foreign key, linking it to the Products’s table. It ensures data integrity, preventing unused records and maintaining data consistency.

```
model Sales {
  saleId      String   @id
  productId   String
  timestamp   DateTime
  quantity    Int
  unitPrice   Float
  totalAmount Float
  product     Products @relation(fields: [productId], references: [productId])
}
```

Figure 36: A model schema

Since the application is used with NextJS framework and Prisma, a Prisma’s schema is required in order to create the database accordingly. Specifically, as shown in Figure 36, Sales table had a foreign key (productId), which corresponded to the unique key from Products table, in the schema’s file to show its relationship.

3.5.2 Controllers

Controller “getDashboard” is for the main homepage dashboard, it accesses to Prisma’s products with “findMany” built-in method to retrieve its relevant data, however, only five of the most sale’s products were shown.

```
export const getDashboard = async (
  req: Request,
  res: Response
): Promise<void> => {
  try {
    const popularProducts = await prisma.products.findMany({
      take: 5,
      orderBy: {
        stockQuantity: "desc",
      },
    });
```

Figure 37: Controller to fetch data with descending order

Other categories from tables such as Sales, Expense are created in the same technique as the variable “popularProdcuts” in Figure 37.

```
    res.json({
      popularProducts,
      saleSummary,
      purchaseSummary,
      expenseSummary,
      expenseByCategorySummary,
    });
  } catch (error) {
    res.status(500).json({ error: "Error while retrieving dashboard data" });
  }
};
```

Figure 38: Return response in json format for requested data

After a successful configuration, the controller should return the requested data. However, it will return to status 500 and an error message if it fails to fetch a specific value.

The controllers are mostly same as “getDashboard function”, but for the product page, there is also a method to create a new product. This method is also known as POST method.

```
export const createProduct = async (
  req: Request,
  res: Response
): Promise<void> => {
  try {
    const { productId, name, price, rating, stockQuantity } = req.body;
    const product = await prisma.products.create({
      data: {
        productId,
        name,
        price,
        rating,
        stockQuantity,
      },
    });
    res.status(201).json(product);
  } catch (error) {
    res.status(500).json({ message: "Failed to create product" });
  }
};
```

Figure 39: Create new products

In Figure 39, firstly, the function deconstructs the value from the request’s body which includes productId, name, price, rating and stockQuantity. If the function

finds the requested value, it creates a new variable to store the new product which has the same properties as the product's interfaced as discussed above.

3.5.3 Routes

```
const router = Router();

router.get("/", getProducts);
router.post("/", createProduct);
```

Figure 40: Creating route for product's page

In Figure 40, the code snippet defines an router using Router() function from ExpressJS. The URL path is the first argument and the second argument is a callback function that will be called if a GET / POST method is requested. The remaining routes for other categories such as dashboard, expenses and users are configured with the same method.

```
// Routes
app.use("/dashboard", dashboardRoutes);
app.use("/products", productRoutes);
app.use("/users", userRoutes);
app.use("/expenses", expenseRoutes);

// Server
const port = Number(process.env.PORT) || 3001;

app.listen(port, "0.0.0.0", () => {
  console.log(`Server is running on port ${port}`);
});
```

Figure 41: Adding middleware to the application's request

Figure 41 shows the server side is adding middleware to a specific path according to application's request - processing pipeline. For instance, the "app.use" function applies functions to several paths such as "dashboardRoutes", "/products", "/users" and "/expenses", then the controller calls the middleware

functions, which is “getDashboard” function in the Figure 37, used for endpoint “/dashboard”.

Finally, the server is set to run on local port (8000), but if there is no PORT value defined in the environment file (.env), it is set to use port 3001 and port 3000 for client side. Moreover, it can be checked whether it has successfully retrieved the requested data or failed in the attempt.

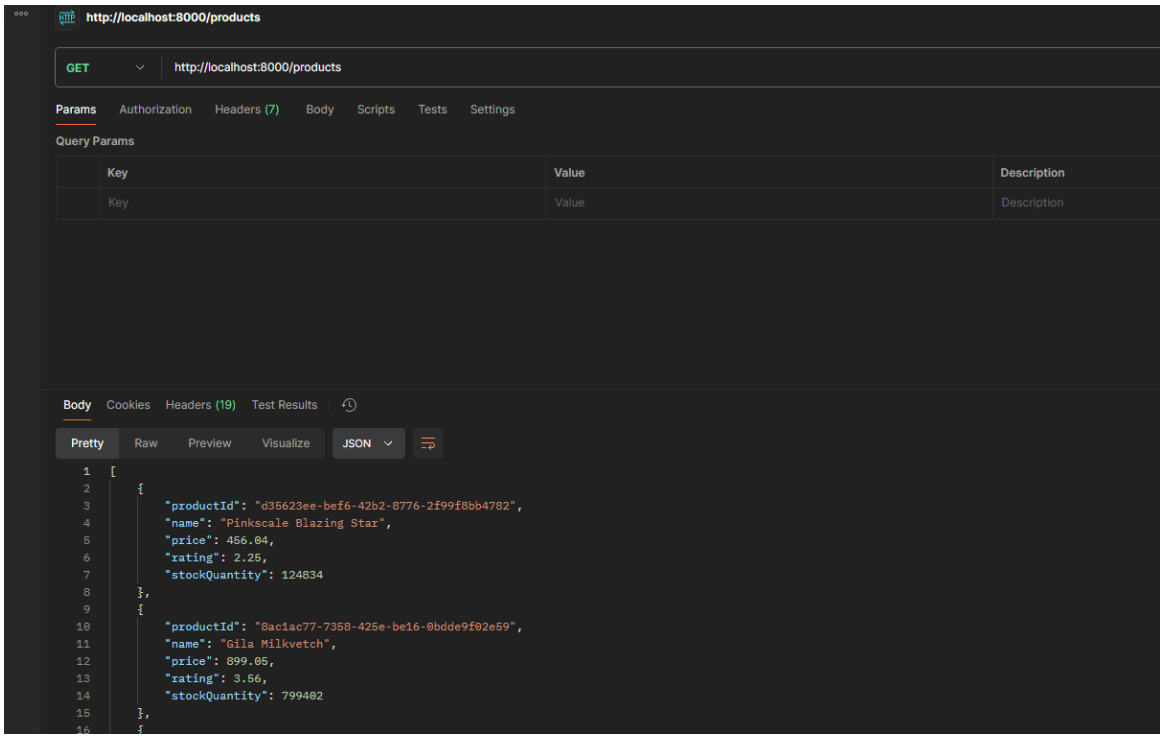


Figure 42: Successful retrieved request data from products table

3.6 Front-End Implementation

3.6.1 Dashboard Page

Figure 43 shows the landing page of the application.

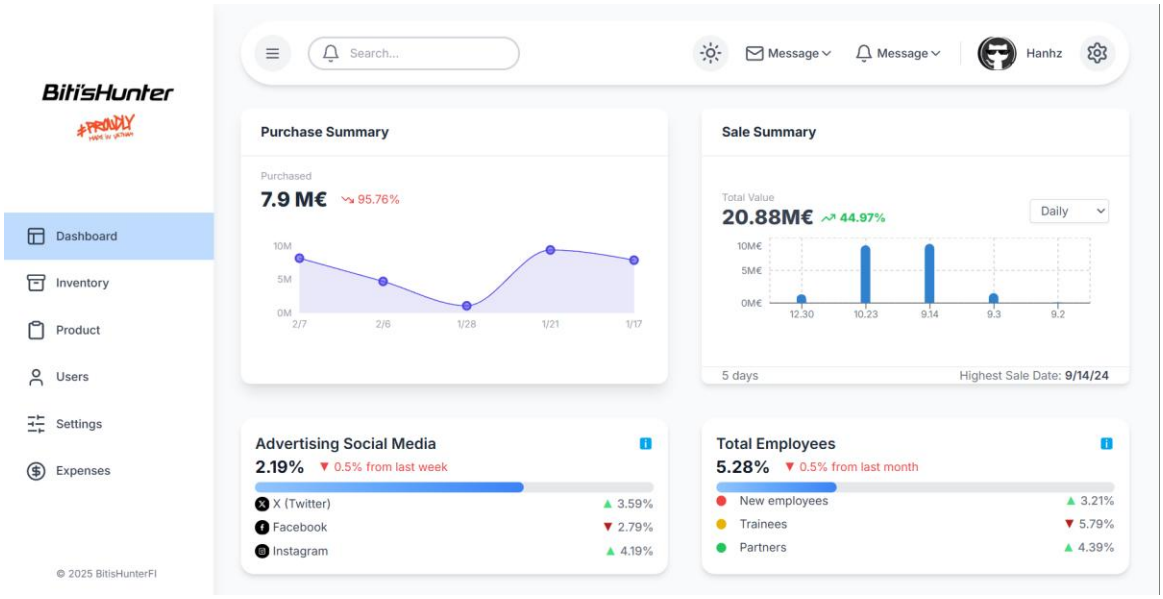


Figure 43: Dashboard homepage

3.6.1.1 Navbar

Figure 44 shows the navbar which appears on every web page of the application. It includes the search bar, dark mode button, message, notification, user's avatar and the setting icon which redirects users to the setting page.

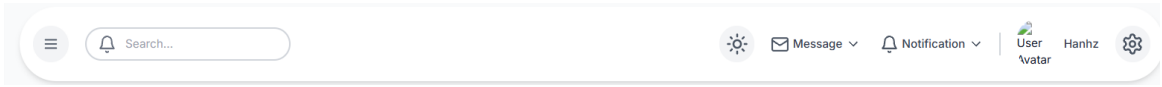


Figure 44: Navbar

```
const dispatch = useAppDispatch();

const isSidebarCollapsed = useAppSelector(
  (state) => state.global.isSideBarCollapsed
);

const toggleSidebar = () => {
  dispatch(setIsSidebarCollapsed(!isSidebarCollapsed));
};

const isDarkMode = useAppSelector((state) => state.global.isDarkMode);

const toggleDarkMode = () => {
  dispatch(setIsDarkMode(!isDarkMode));
};
```

Figure 45: Redux state management in Navbar component

Figure 45 shows the Redux implementation using hooks to manage global state. It uses “useAppSelector” to access specific values and “useAppDispatch” to dispatches actions. The function “toggleSidebar” updates the sidebar’s status by calling “setIsSidebarCollapsed”, whereas “toggleDarkMode” modifies the dark mode setting using “setIsDarkMode”. Both of these functions are used by dispatch function, which is obtained through “useDispatch” built-in method.

3.6.1.2 Sidebar

Sidebar also has the same functions from redux to control the sidebar’s behavior. However, the main difference is the links between each different web’s pages such as dashboard’s page, inventory’s page. A component is created since the architecture of the Sidebar’s link are the same for these pages as those shown Figure 46.

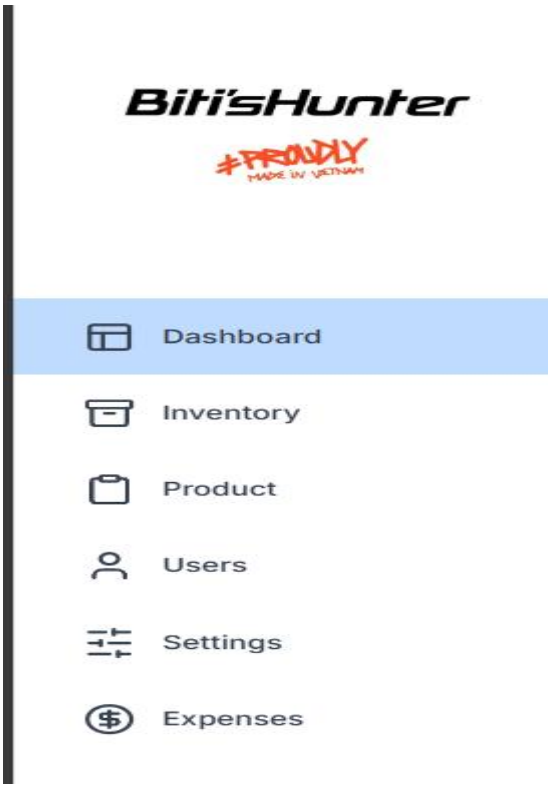


Figure 46: Side bar

```
<SidebarLink
  href="/dashboard"
  icon={Layout}
  label="Dashboard"
  isCollapsed={isSidebarCollapsed}
/>
<SidebarLink
  href="/inventory"
  icon={Archive}
  label="Inventory"
  isCollapsed={isSidebarCollapsed}
/>
```

Figure 47: An example of some website's path within a component

Both Sidebar and Navbar should exist in the entire application for better user interface and user experience, which are passed as components within the website's layout file.

3.6.1.3 Expense Summary Chart

Figure 48 shows the expense chart from library recharts containing data from local database.

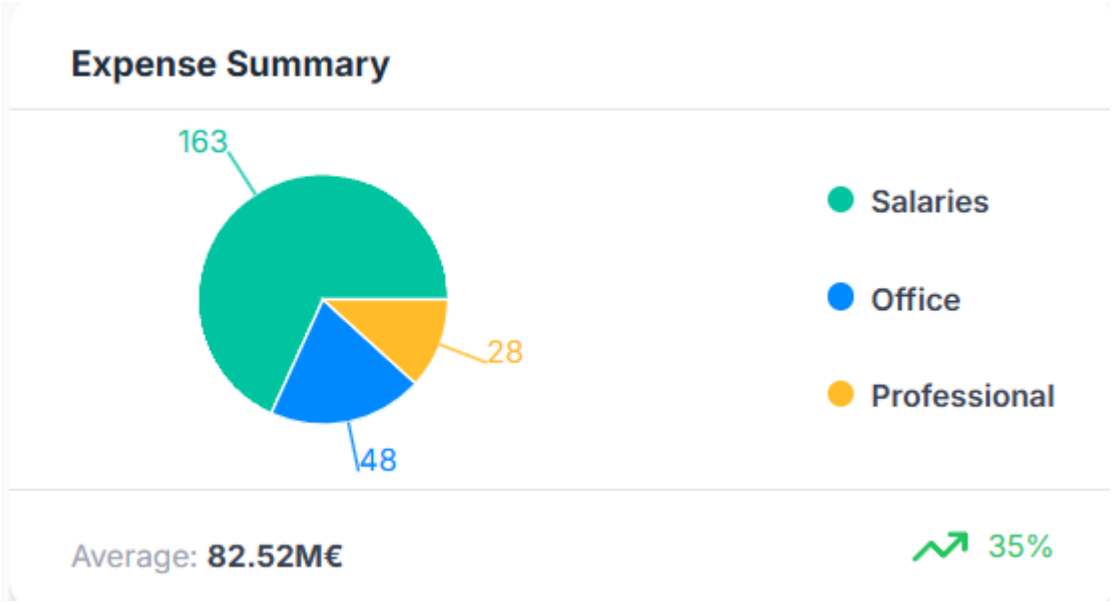
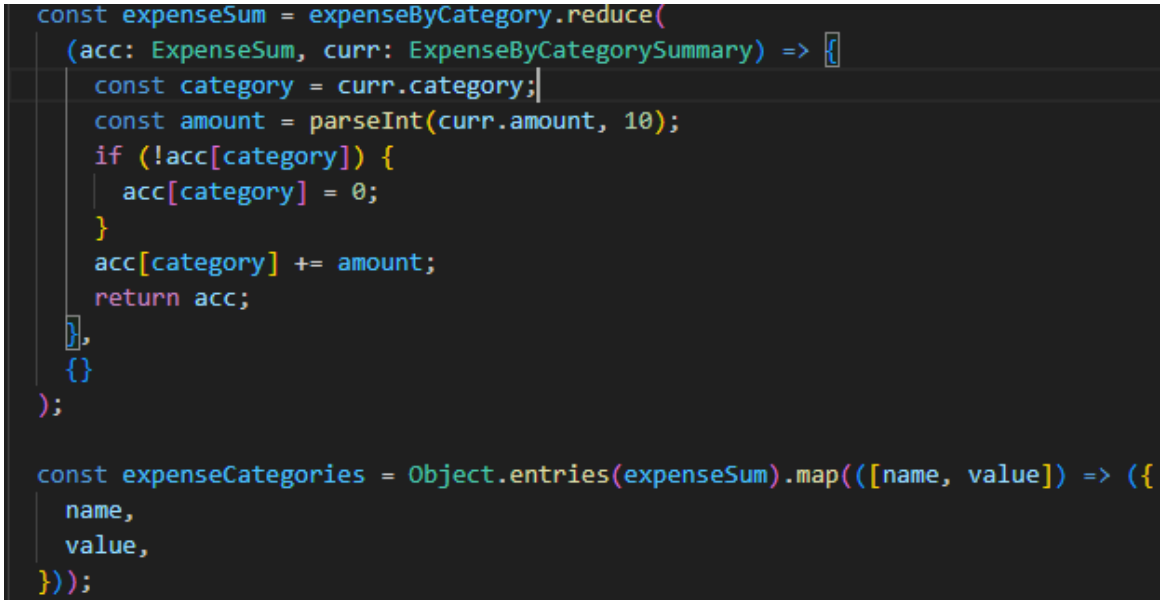


Figure 48: Expense Summary Chart



```
const expenseSum = expenseByCategory.reduce(
  (acc: ExpenseSum, curr: ExpenseByCategorySummary) => {
    const category = curr.category;
    const amount = parseInt(curr.amount, 10);
    if (!acc[category]) {
      acc[category] = 0;
    }
    acc[category] += amount;
    return acc;
  },
  {}
);

const expenseCategories = Object.entries(expenseSum).map(([name, value]) => ({
  name,
  value,
}));
```

Figure 49: Logic to create data for the pie chart

Firstly, the function “expenseSum” in Figure 49 calculates the total expenses for each category and creates a summary of the expenses. By using reduce function to iterate over an array of the provided variable “expenseByCategory”, where each item contains a category and an value, the variable “category” name is saved in the newly created constant “category” as well as the variable "amount” is transformed into an integer instead of a string. If the accumulator (also known as acc) does not exist, it is set to 0. Then, the parsed amount is added to the corresponding category’s total in the accumulator. Finally, the “expenseSum” has an object which includeds the category name and the values, representing the total sum of each category.

Furthermore, in Figure 49, “expenseCategories” main’s purpose is to convert the “expenseSum” object into an array of objects, each containing the category’s name and its respective total value for the chart.

3.6.1.4 Sale Summary

Figure 50 illustrates a total value of 20.88 million euros and indicates a growth rate of 44.97%, meaning that the sales increased over each year. The chart itself displays data from over several dates, with the highest sales value marked on 14 of September 2024.

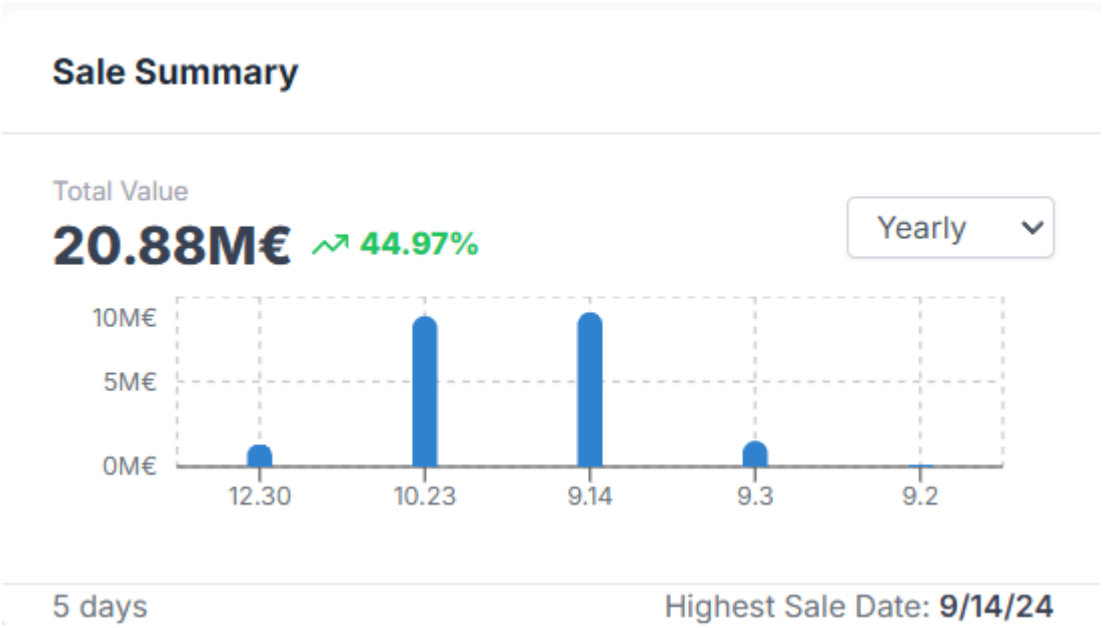


Figure 50: Bar chart for sale summary

```
const saleData = data?.saleSummary || [];

const totalValueSum =
  saleData.reduce((acc, curr) => acc + curr.totalValue, 0) || 0;

const averageChangePercentage =
  saleData.reduce((acc, curr, _, arr) => {
    return acc + curr.changePercentage! / arr.length;
  }, 0) || 0;

const highestValueData = saleData.reduce((acc, curr) => {
  return acc.totalValue > curr.totalValue ? acc : curr;
}, saleData[0] || {});

const highestValueDate = highestValueData.date
  ? new Date(highestValueData.date).toLocaleDateString("en-US", {
      month: "numeric",
      day: "numeric",
      year: "2-digit",
    })
  : "N/A";
```

Figure 51: Logic to get the data rendered and the highest sales value date

The code snippet in Figure 51 shows the method to handle the data from back - end and delivers it to front - end. First, a variable was created to get the data from the “Dashboard”. The “totalValueSum” uses reduce built-in function to iterate over every item in “saleData” and accumulate the total values, otherwise, it is assigned

to 0. Variable “averageChangePercentage” calculates the average of the “changePercentage” property across all items in “saleData”. Then, the reduce function is used to sum up the “changePercentage” values and is divided by the length of the array (arr.length). Finally, the variable “highestValueData” uses “reduce” to compare each item’s “totalValue” to find the highest total value in the “saleData”. After successfully retrieves the highest value data, the date property is formatted into a readable string with “toLocaleDateString”.

3.6.2 Inventory Page

Figure 52 shows the design of the inventory page using DataGrid component from Material User Interface (MUI).

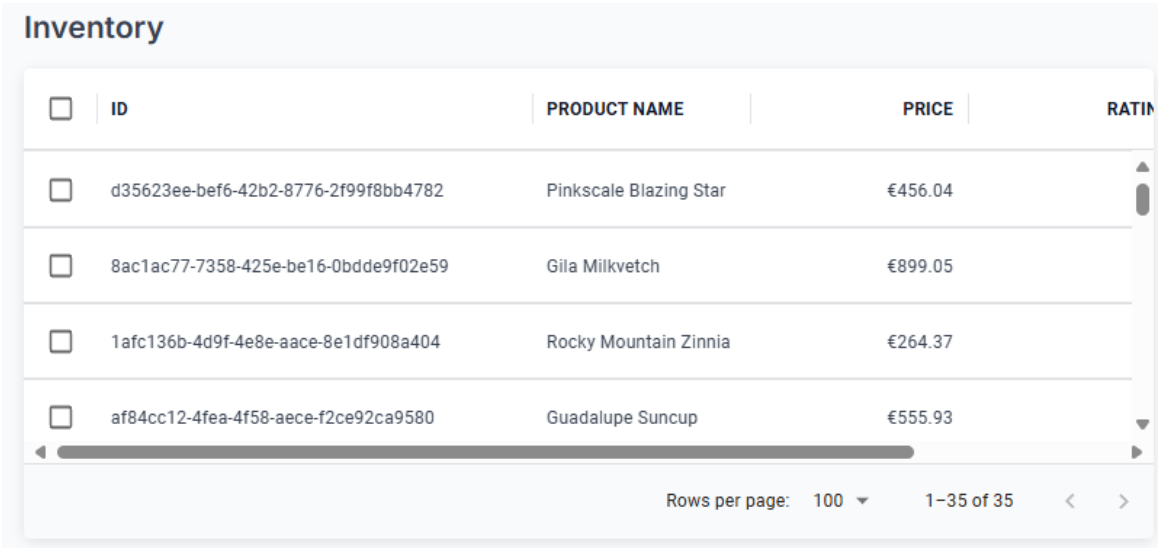


Figure 52: Inventory page

```
const columns: GridColDef[] = [
  { field: "productId", headerName: "ID", width: 300 },
  { field: "name", headerName: "Product Name", width: 150 },
  {
    field: "price",
    headerName: "Price",
    width: 150,
    type: "number",
    valueGetter: (_, row) => `€${row.price}`,
  },
  {
    field: "rating",
    headerName: "Rating",
    width: 150,
    type: "number",
    valueGetter: (_, row) => (row.rating ? row.rating : "N/A"),
  },
  {
    field: "stockQuantity",
    headerName: "Stock Quantity",
    width: 150,
    type: "number",
  },
];
```

Figure 53: Columns data required for DataGrid

```
<DataGrid
  rows={data}
  columns={columns}
  getRowId={(row) => row.productId}
  checkboxSelection
/>
```

Figure 54: DataGrid component from MUI

The data is fetched from Redux's hooks, which is created when configured Redux, and passed to the rows property to render the data.

3.6.3 Products Page

In the same way with inventory's page, the product page retrieves data from the "useGetProductsQuery" hook from Redux. However, the page contains a button to add product, which can be added directly to the database since the application's server already configured the POST method for the product's endpoint.

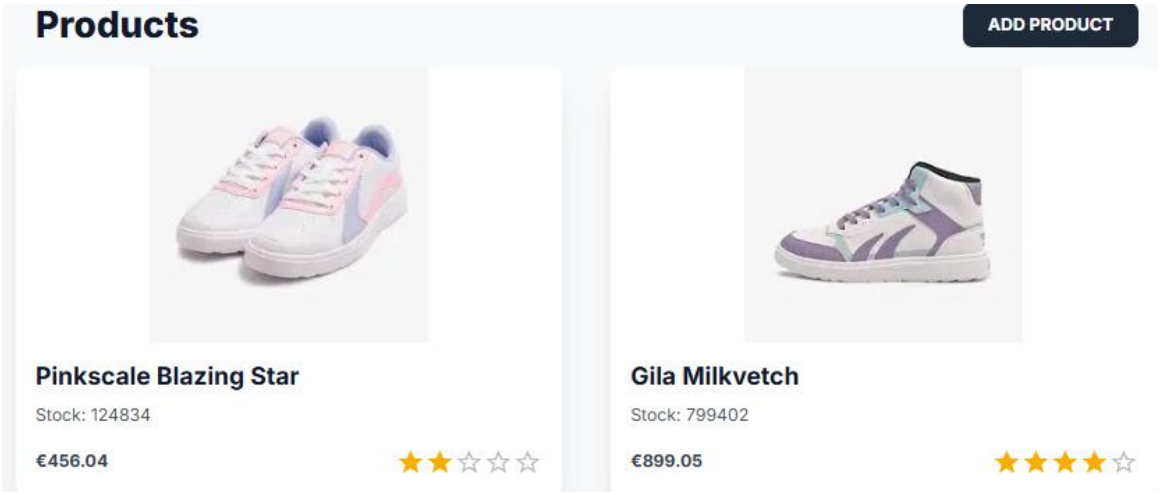


Figure 55: Product page

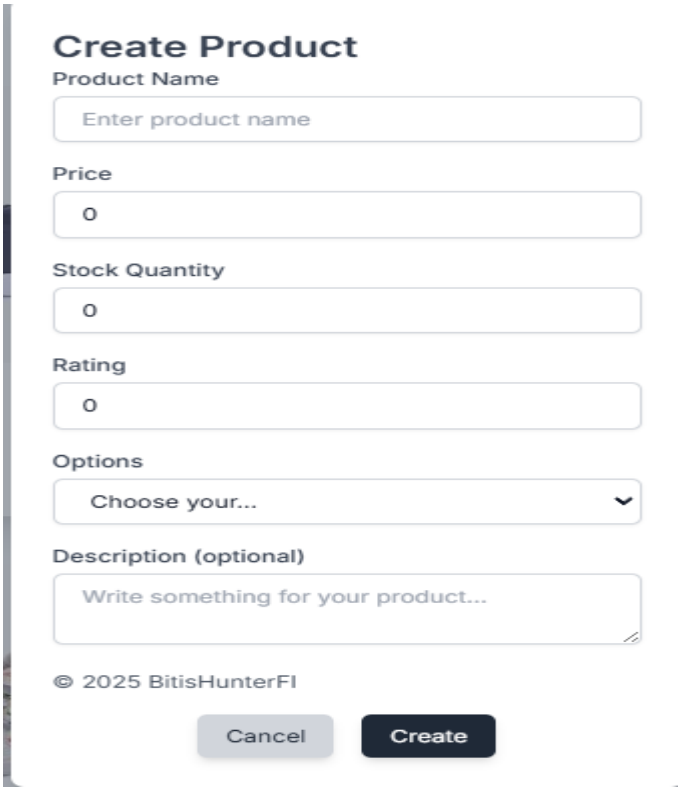


Figure 56: Create product button design

3.6.4 User Page

User's page is mostly the same way with inventory's page design. It also contains the columns for the Datagrid as well as the data which is retrieved using "useGetUsersQuery" hooks.

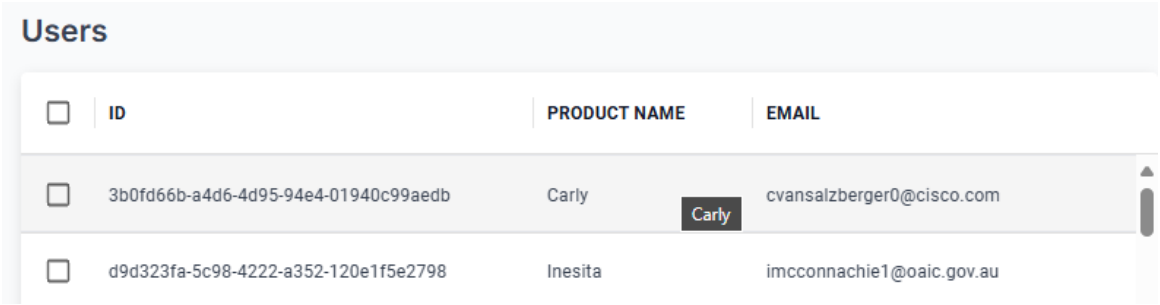


Figure 57: User page table using Datagrid component from MUI

```
const columns: GridColDef[] = [
  { field: "userId", headerName: "ID", width: 300 },
  { field: "name", headerName: "Product Name", width: 150 },
  { field: "email", headerName: "Email", width: 300 },
];

const UsersPage = () => {
  const { data, isError, isLoading } = useGetUsersQuery();
```

Figure 58: Columns data and data for user's page

3.6.5 Expense Page

Figure 59 shows the expense dashboard page with data fetched from back-end.

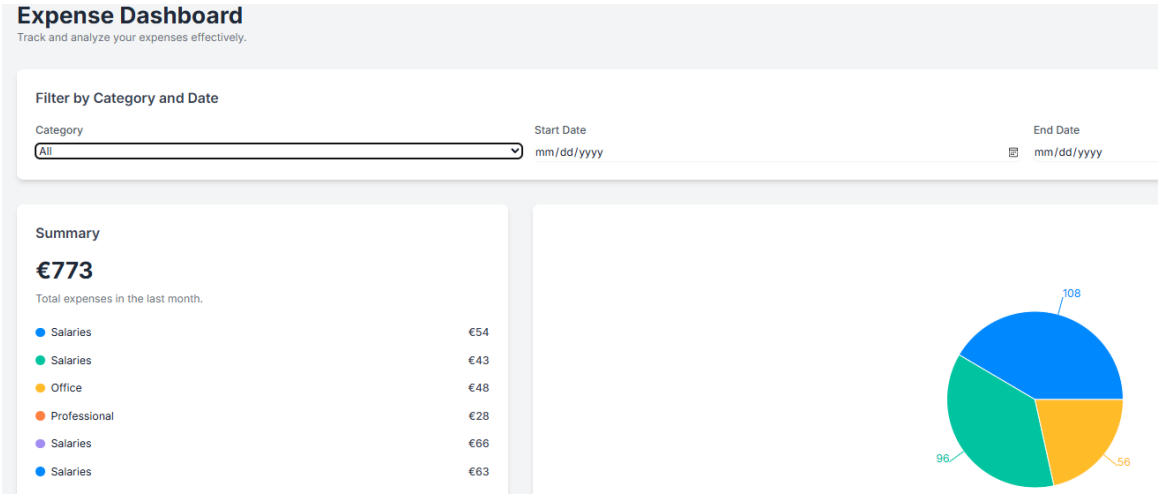


Figure 59: Expense page

```
const parseDate = (datestring: string) => {
  const date = new Date(datestring);
  return date.toLocaleDateString().split("T")[0];
};

const totalExpenses = useMemo(() => {
  return expenses.reduce((acc, data) => acc + parseInt(data.amount), 0);
}, [expenses]);
```

Figure 60: parseDate function and totalExpenses constant

The code snippet in Figure 60 includes two variables “parseDate” and “totalExpenses”. The first variable is used to convert a date string into a standard date format, stripping out the time portion. On the other hand, the “totalExpenses” is calculated using the “useMemo” hook, where it sums up the amount of all expenses within the expense array, passing each amount as an integer.

```
const expenseCategories: ExpenseCategoriesItem[] = useMemo(() => {
  const categories: ExpenseCategories = expenses
    .filter((data: ExpenseByCategorySummary) => {
      const matchesCategory =
        categoryFilter === "All" || data.category === categoryFilter;
      const dateDate = parseDate(data.date);
      const matchesDate =
        !startDate ||
        !endDate ||
        (dateDate >= startDate && dateDate <= endDate);
      return matchesCategory && matchesDate;
    })
    .reduce((acc: ExpenseCategories, data: ExpenseByCategorySummary) => {
      const amount = parseInt(data.amount);
      if (!acc[data.category]) {
        acc[data.category] = { name: data.category, amount };
        acc[data.category].color =
          colors[Object.keys(acc).length % colors.length];
        acc[data.category].amount += amount;
      }
      return acc;
    }, {});
  return Object.values(categories);
}, [expenses, categoryFilter, startDate, endDate]);
```

Figure 61: “expenseCategories” function logic to use in the pie chart

The “expenseCategories” array is calculated with “useMemo” function to ensure efficient recalculation only when dependencies change. First, it filters the expenses based on the selected category and date range. Then, it uses “reduce” method to accumulate totals for each category, adding the corresponding amount for each category and assigning the color to each category using a predefined colors array. The result is a converted array of expense categories, containing the total amount and color for each category.

3.7 Deploying To Amazon Web Services

Figure 62 shows the architecture of the deployment of NextJS application to Amazon Web Services including some services from AWS such as: EC2, AWS Amplify, AWS RDS, AWS S3.

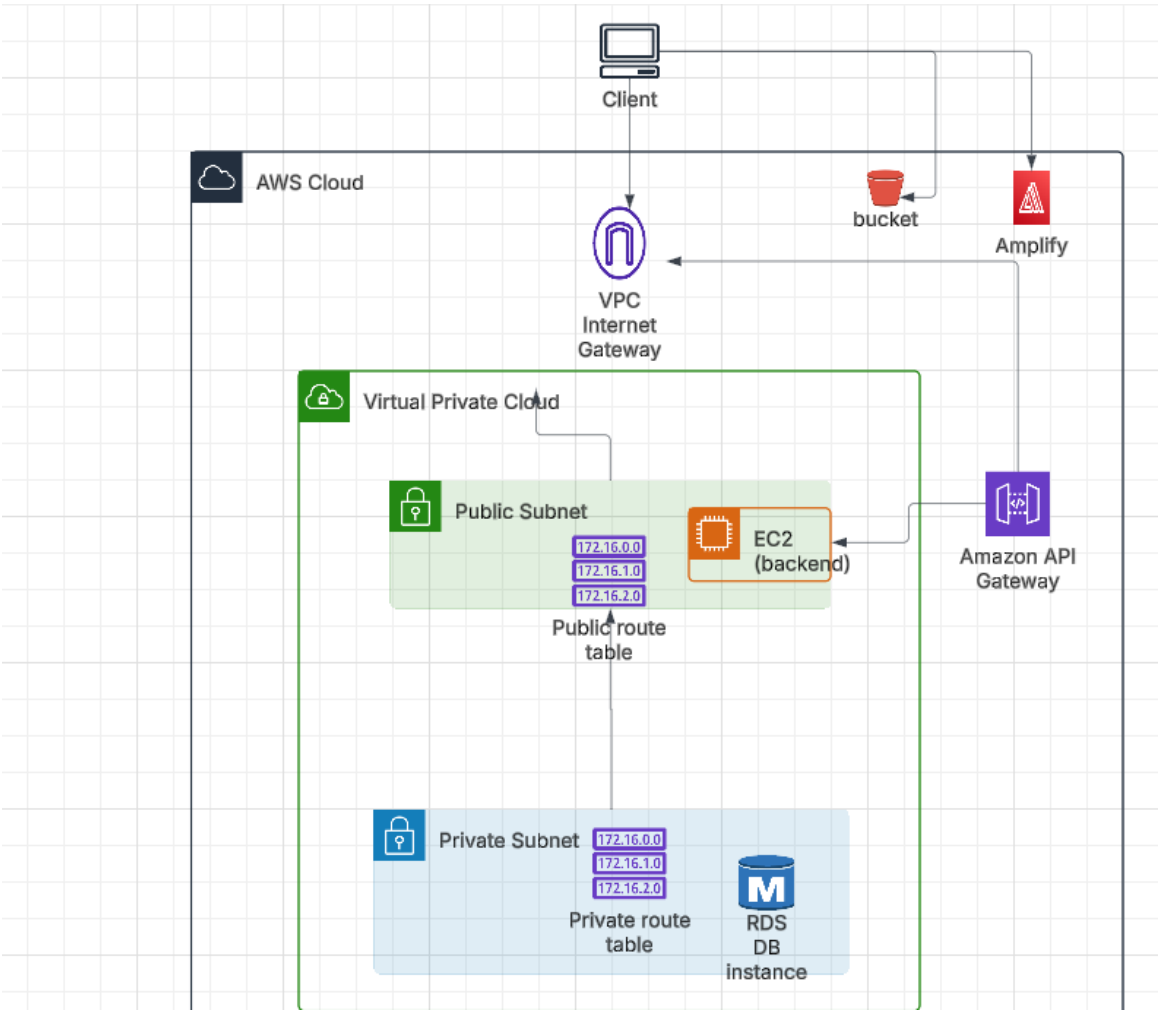


Figure 62: Web application architecture on Amazon Web Services

3.7.1 Configuring Virtual Private Cloud

Firstly, a Virtual Private Cloud is created according to the AWS VPC documentation which includes VPC details such as name, IPV4 CIDR range or tenancy. The successfully creating VPC should be similar in Figure 63.

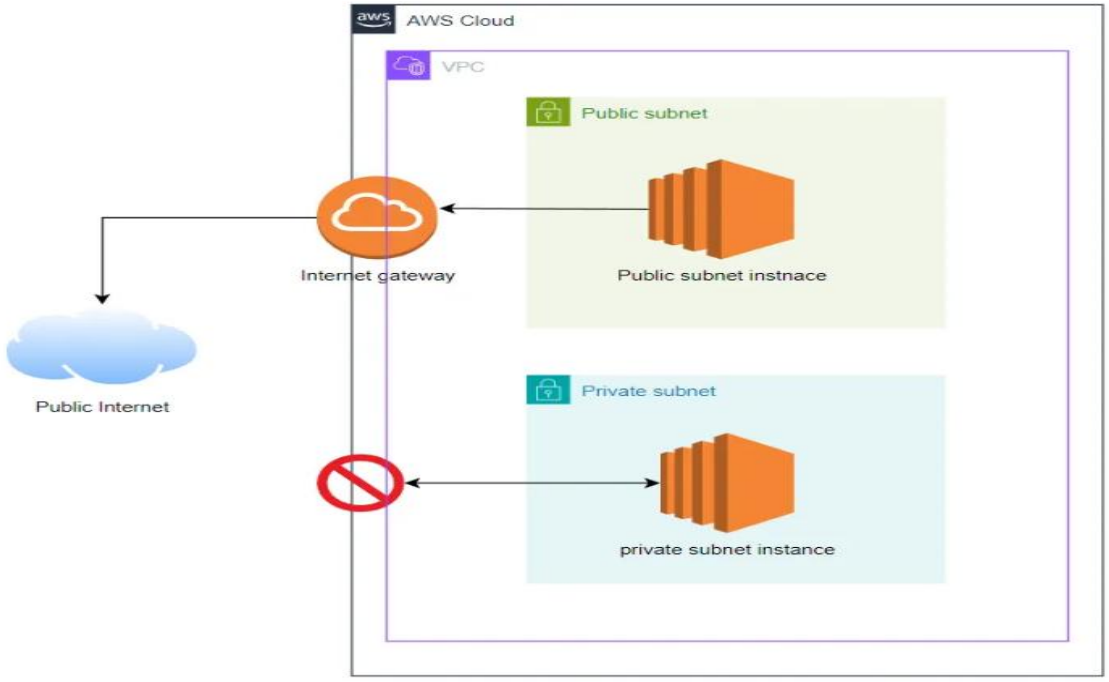


Figure 63: Virtual Private Cloud Configuration Design

Figure 64: A successfully created Virtual Private Cloud

3.7.2 Subnet

As seen in Figure 63 above, VPC contains services such as subnet, internet gateway, route table

Secondly, there are two subnets that need to be configured, public and private subnet. By logging on to AWS console, the subnet can be found within the VPC

services with a label “create a subnet”. Here are images of configuration for subnet within VPC in Figure 65, 66.

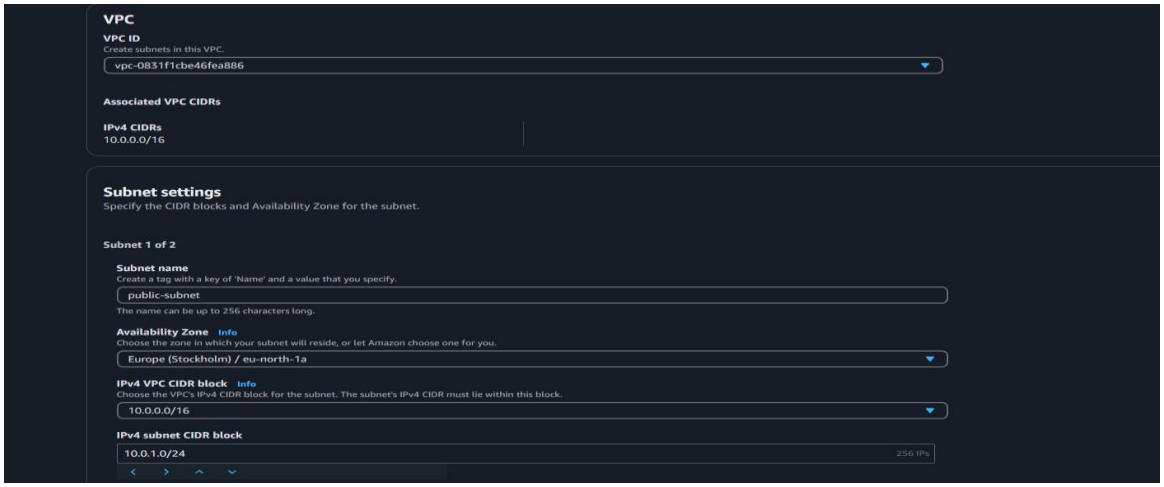


Figure 65: Assigning public subnet for VPC

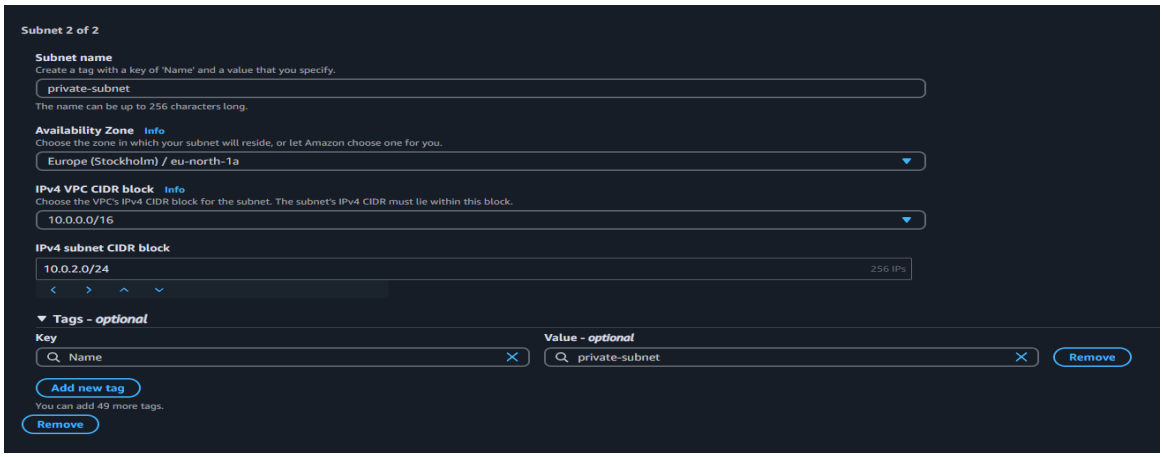


Figure 66: Assigning private subnet for VPC

After subnets are created, users can see the subnet which are bound with the VPC as illustrated in Figure 67.

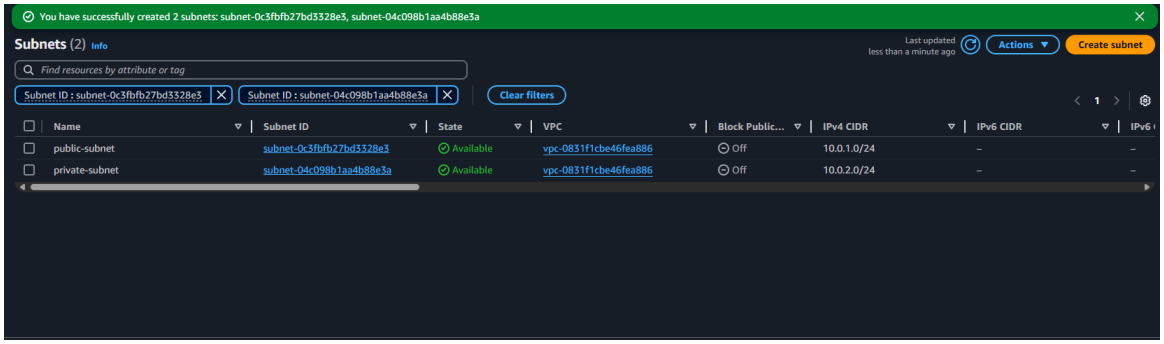


Figure 67: Both public and private subnet had been successfully assigned to VPC

3.7.3 Internet Gateway

An internet gateway is required for the VPC to connect to the internet.

The Internet Gateway should be seen on the left panel of the VPC configuration page, and the setting image from Internet Gateway is shown in Figure 68.

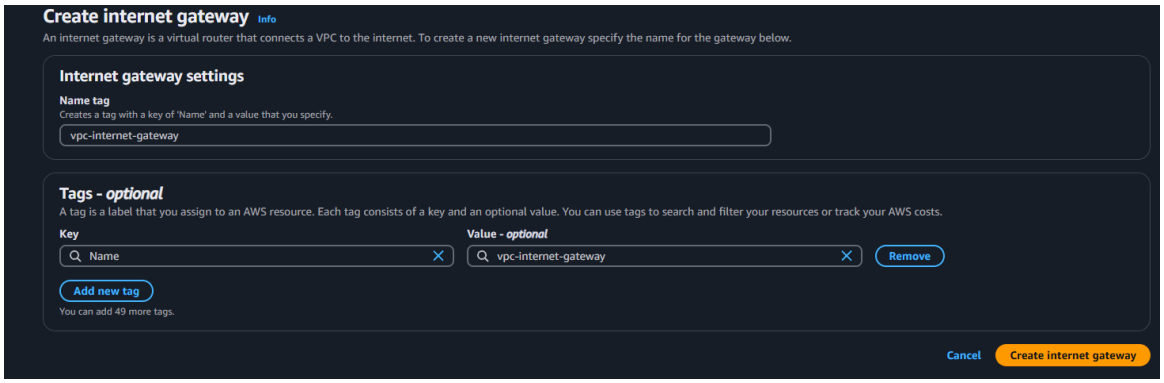


Figure 68: Creating an Internet Gateway

Finally, the Internet Gateway can be attached to the VPC created above like in the Figure 69.

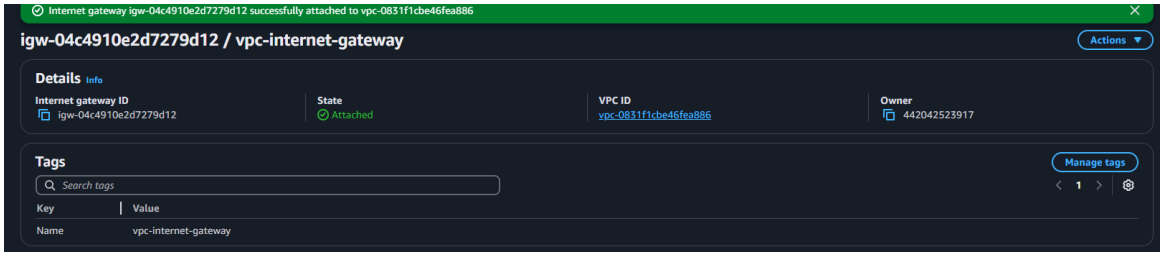


Figure 69: An Internet Gateway is attached to the provided VPC

3.7.4 Route Table

There are two route tables, public and private, that must be configured. Figure 70 shows the content inside the setting page for the route table

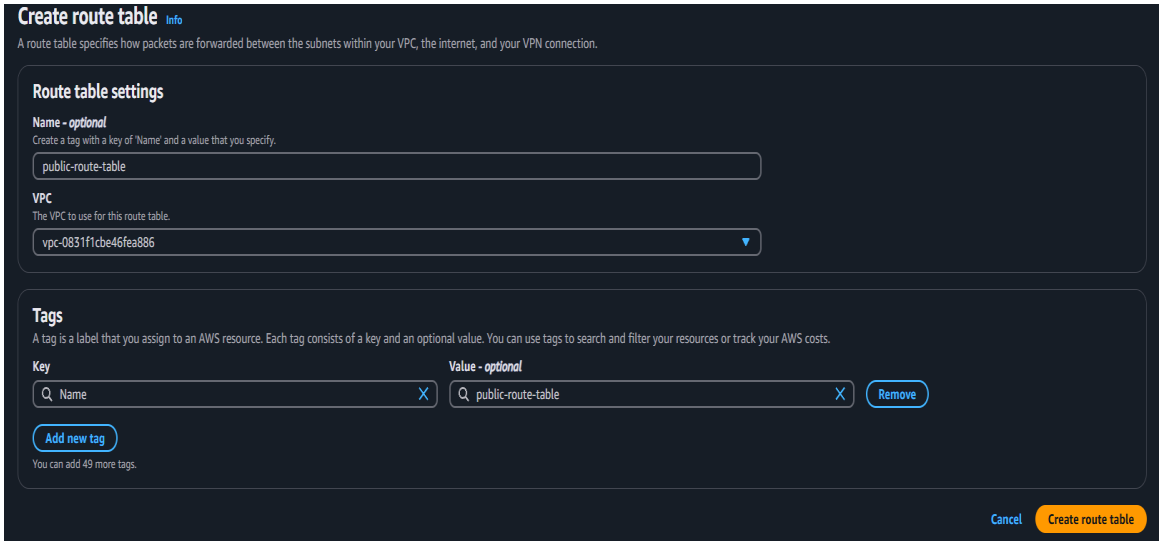


Figure 70: Setting page for the route table

After configuring the route table, the setting page should notify its creators as presented in Figure 71, 72.

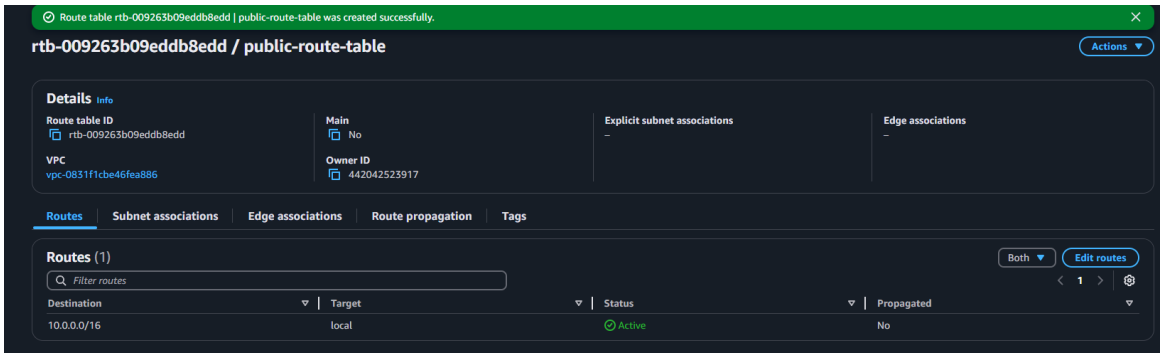


Figure 71: Route table for public

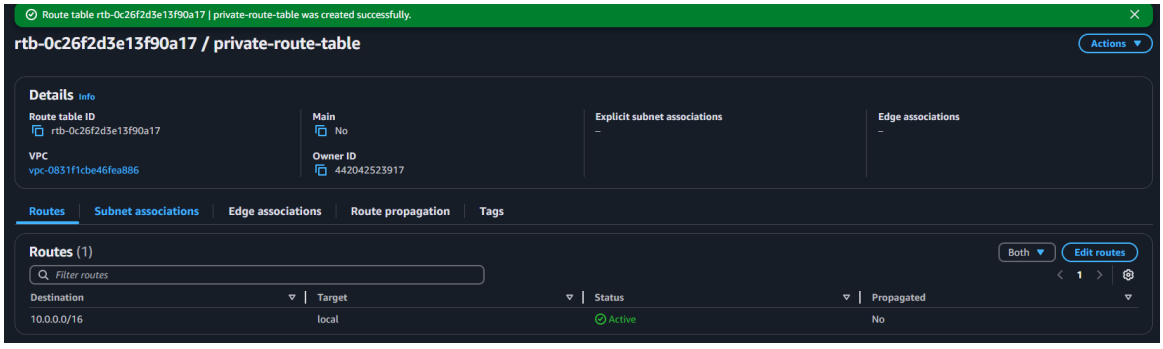


Figure 72: Route table for private

As mentioned above, there are private and public subnet inside a VPC, which are needed to be explicitly associated since each subnet must be assigned to a specific route table to define how traffic is directed inside and outside the VPC.

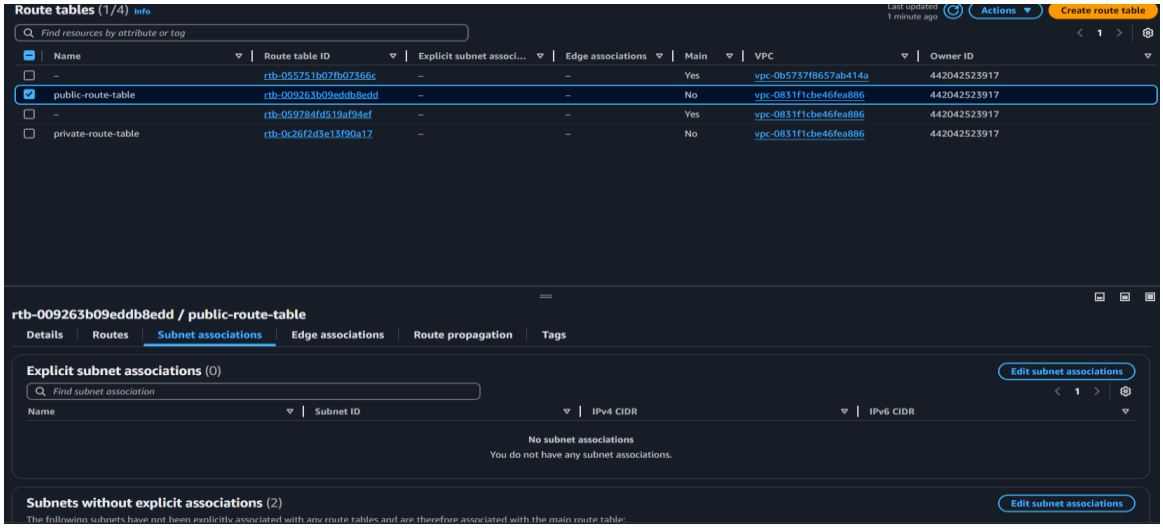


Figure 73: Editing subnet association within route table

As seen in Figure 73, both public and private route table are missing their corresponding subnet association.

If both of subnets are configured correctly to its related route-table, the result is the same as what is shown in Figure 74.

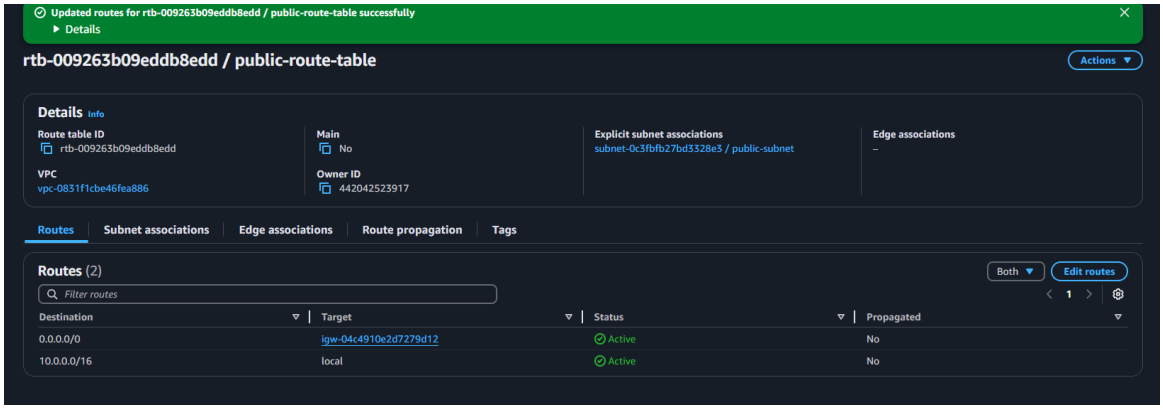


Figure 74: Public subnet had been assigned to the public route table

This indicates that the route table had been correctly configured.

3.7.5 AWS EC2

When provisioning EC2 instances, it is vital to ensure that they are deployed within the appropriate VPC Subnet. For instance, public instance should be

designated the public network and public subnet, and private instance is assigned with private network and private subnet.

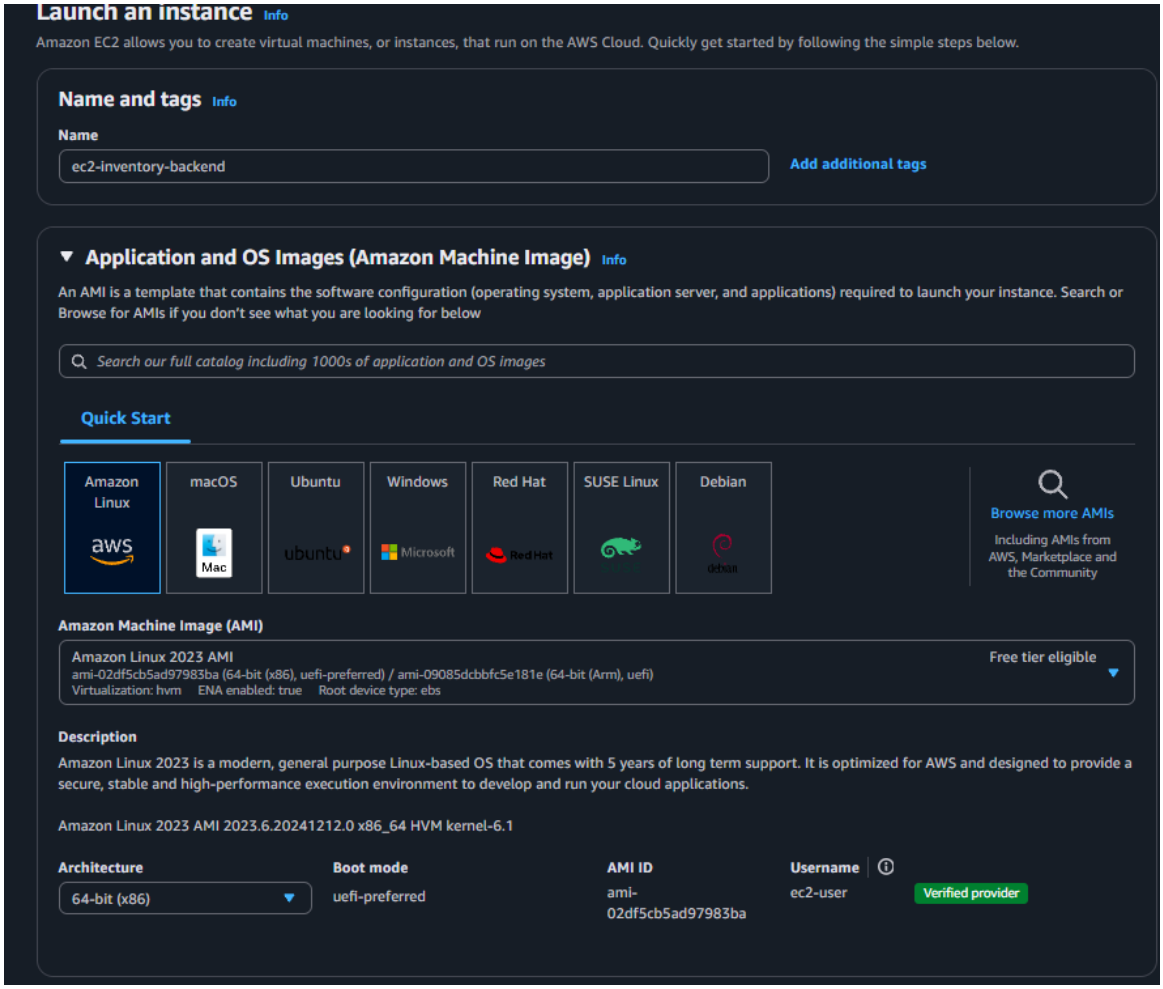


Figure 75: A section when setting up EC2 instance

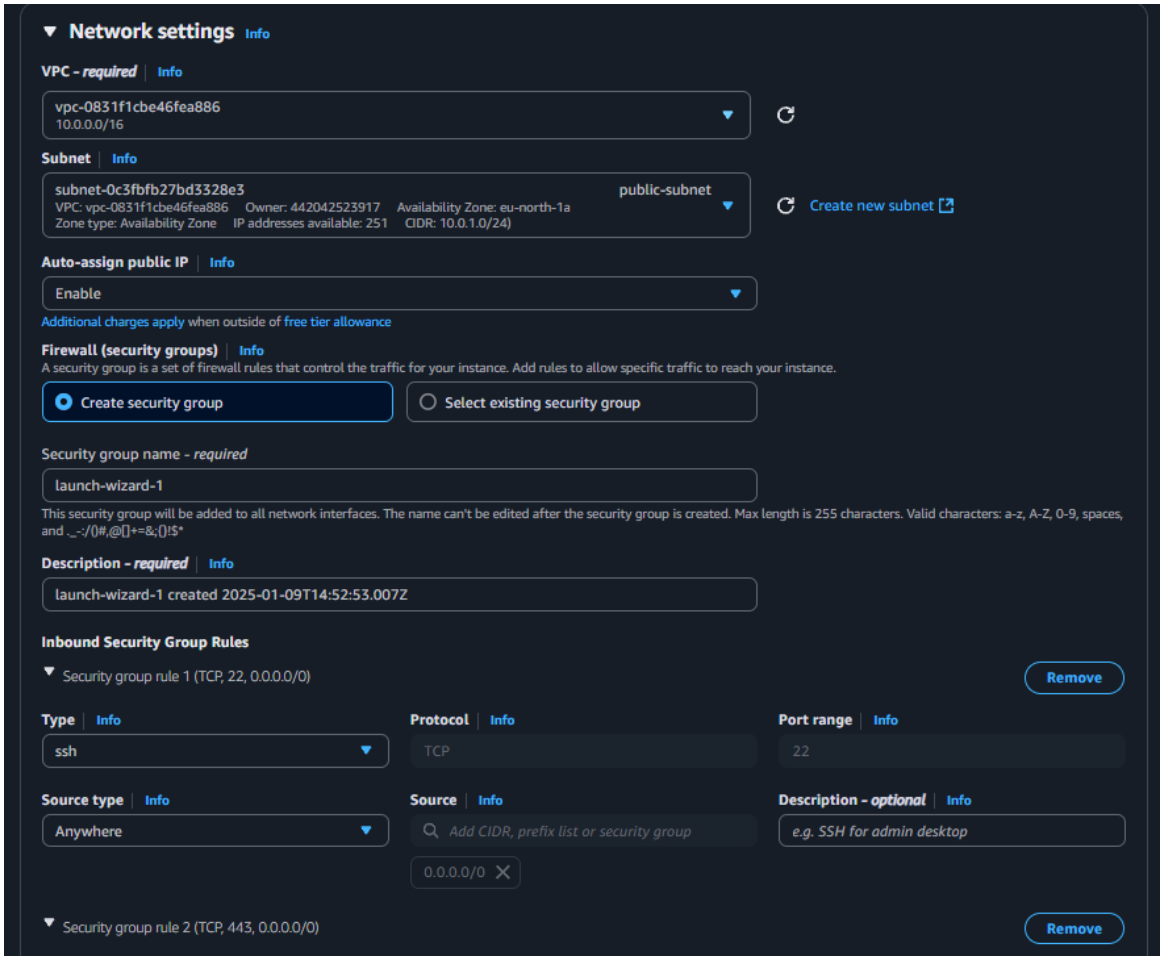


Figure 76: Networking section within EC2 setting up

Figure 77 shows the result if setting up EC2 successfully.

Figure 77: EC2 instance is now available to connect

Then, a connection to EC2 is established and some dependencies are required to download for the Linux environment.

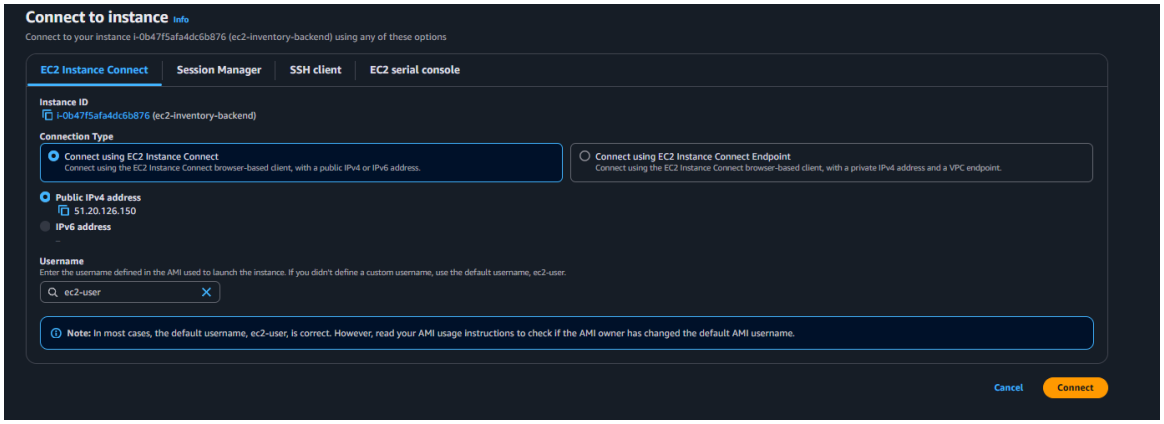


Figure 78: Connecting to EC2 instance

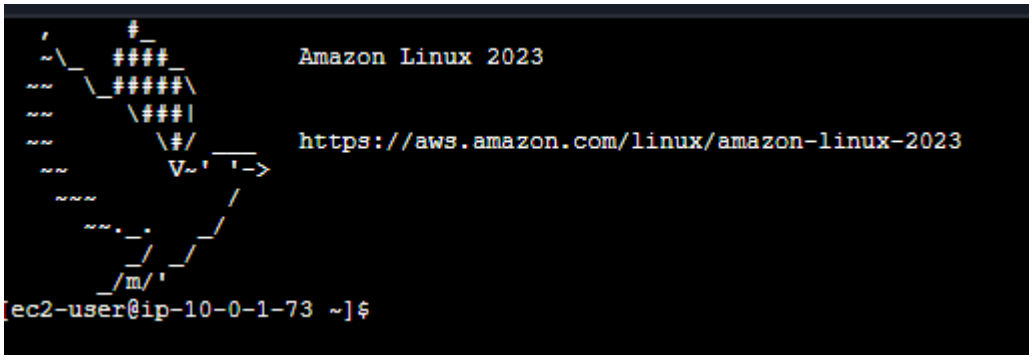


Figure 79: Successfully connecting to EC2 instance

After testing EC2 instance is running correctly, the repository for the code base will be cloned to the virtual machine

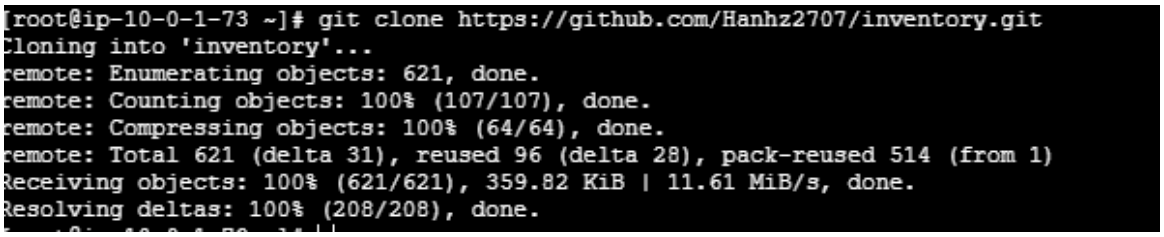


Figure 80: Cloning codebase to EC2

Next, PM2 is required to download since it is the process management tool to keep the application running with NodeJS.

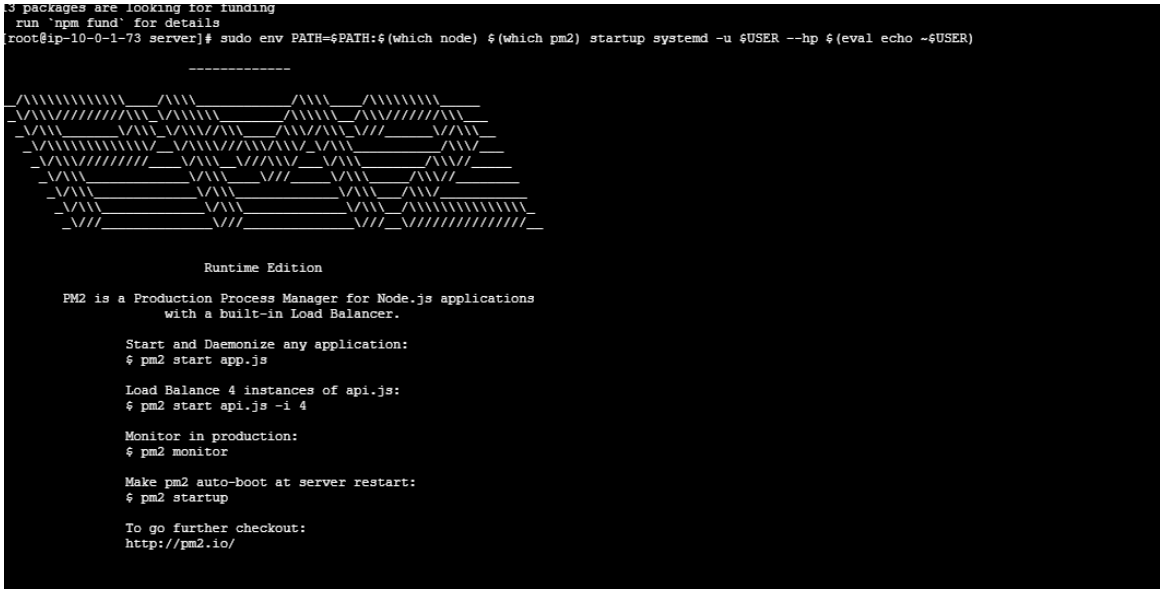


Figure 81: Successfully installing PM2 in virtual machine

3.7.6 Relational Database Service Route Table Association

In figure 82, a database subnet group is created using the private subnet. The setting of this subnet in general is the name of the subnet group, VPC, availability zone and subnets (private).

Subnets (2)

Availability zone	Subnet name	Subnet ID	CIDR block
eu-north-1b	private-subnet-2	subnet-01323ec99230434b9	10.0.3.0/24
eu-north-1a	private-subnet	subnet-04c098b1aa4b88e3a	10.0.2.0/24

Figure 82: Subnet group for RDS

After finishing creating subnet for RDS, the RDS instance of PostgreSQL can be configured. There are some images of RDS's setting page presented in Figures 83 and 84.

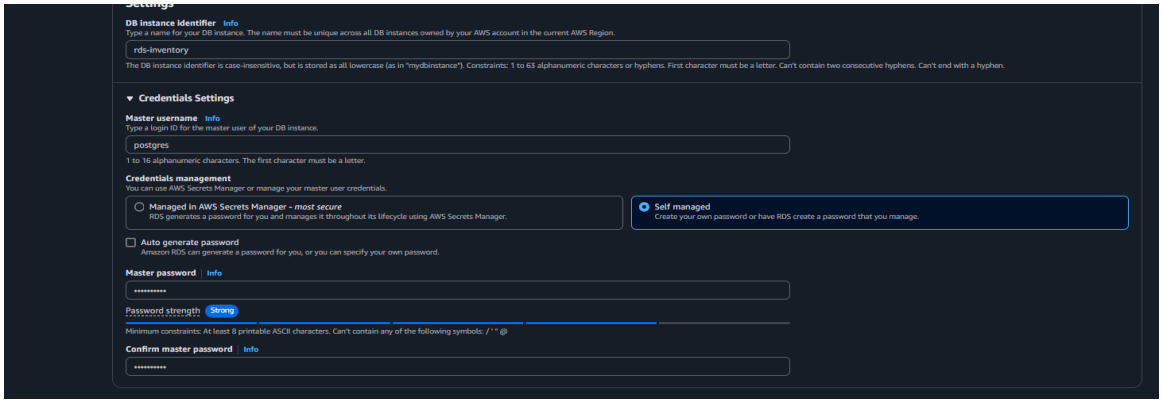


Figure 83: Setting up RDS

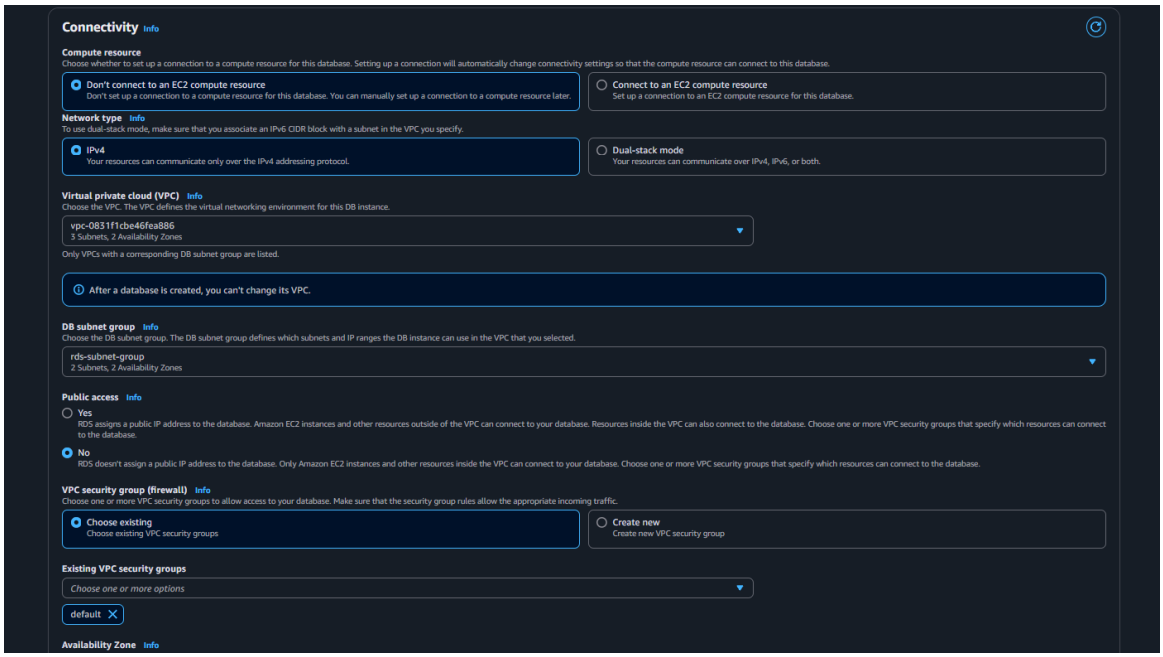


Figure 84: Configuring connectivity for RDS instance

Figure 85 shows the result if RDS is configured correctly.

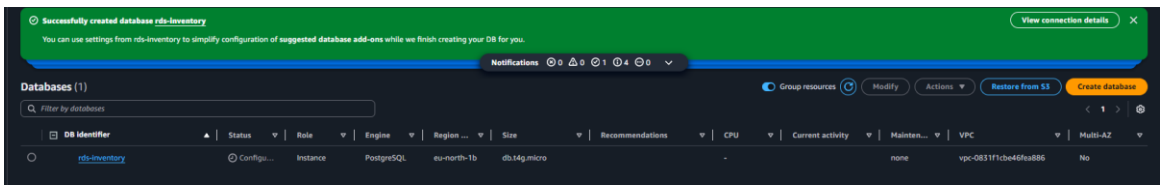


Figure 85: Successfully creating RDS instance for database

Finally, within EC2 console, migrate the database and run the back-end server.

Figure 86: Migrating data in prisma

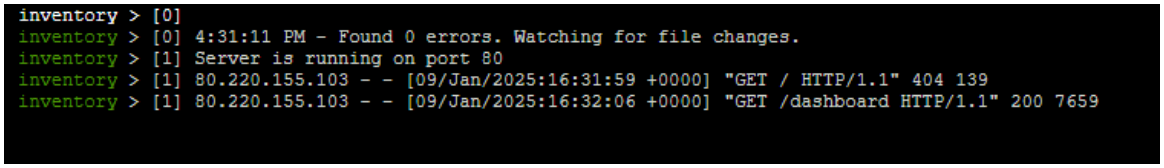


Figure 87: Retrieving data in public ip address

3.7.7 Configuring AWS API Gateway

Figure 88 shows the API Gateway configuration page, which an API type, HTTP API, is selected.

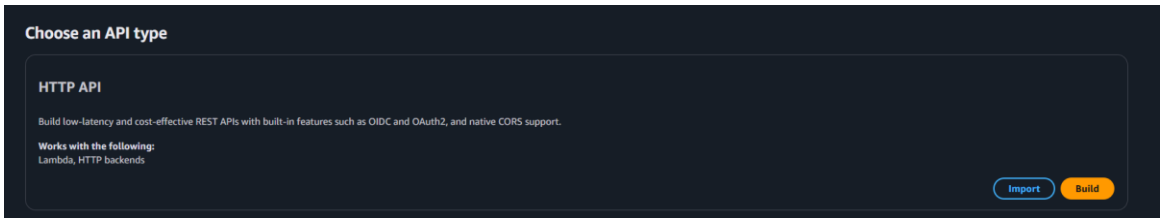


Figure 88: Choosing APIs type

Then, integrations are created based on the routes defined above such as to retrieve dashboard data, the application use “/dashboard” route with GET method.

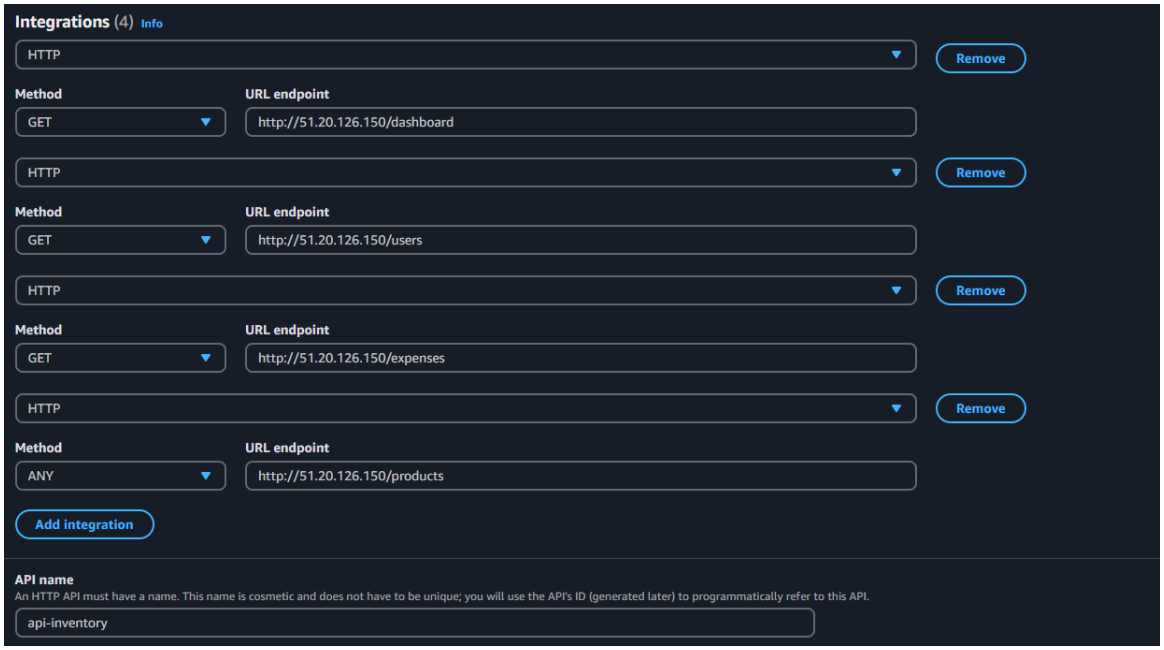


Figure 89: Adding integrations for API

Figure 89 shows the method this application use and the address from the EC2 instance, which is “https://51.20.126.150/”.

3.7.8 Deploying application to AWS Amplify

The configuration steps are straightforward since it uses the provided repository with some setting up such as choosing the framework, app’s name or frontend build command.

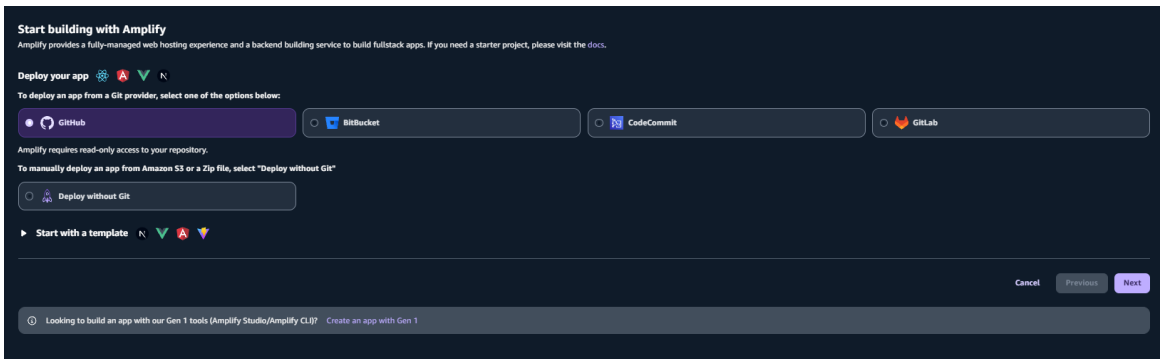


Figure 90: Deploying to AWS Amplify

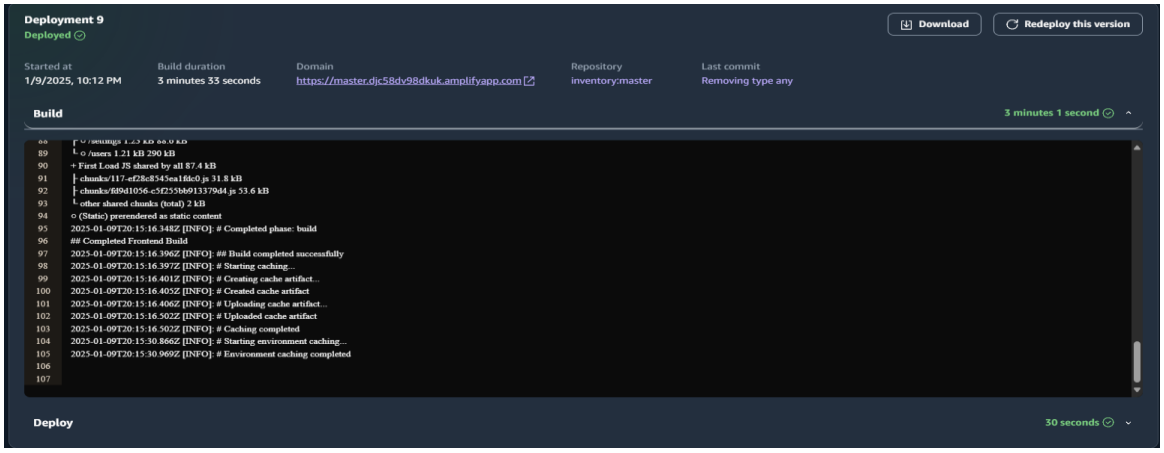


Figure 91: Building console in Amplify

Figure 92: Application URL path with Amplify

4 FUTURE DEVELOPMENT

The procedures introduced in this thesis hold a significant potential for future development. One crucial improvement would be the implementation of real authentication. Currently, the application relies on mockData, but enabling admins to log in with their credentials would greatly improve the website's security. Each admin should have unique access, allowing them to manage the store associated with their account.

Additionally, with advancements in Artificial Intelligence, integrating a chatbot could be highly beneficial for store managers. A tool like OpenAI's ChatGPT could assist in managing store operations, offering insights, and streamlining processes.

5 CONCLUSION

In conclusion, Bitit's Hunter Admin Dashboard achieved the objectives that were set in this study. The main purpose of the applications is to help managers in managing their associated stores.

This study utilized a comprehensive stack of technologies such as NextJS framework, ExpressJS, TypeScript, TailwindCSS, PgAdmin 4 and Amazon Web Services. Specifically, NextJS along with TypeScript and TailwindCSS are used as the front-end technology while ExpressJS served as the server-side component. In addition, PgAdmin 4 was also used as the database management tool, and AWS was used as the cloud services for the application.

The implementation phase defined how the front-end and back-end interact. Managers can retrieve the all the needed data from stores such as Expense or Income, or they can deploy list or creates new products without having to regard the configuration in the database since the application will also update the database if new products are created.

Further improvements for the website might entail adding authentication and implementing AI technology. These utilities will play an important role when it comes to helping managers to control the store’s operations as well as increasing the security.

REFERENCES

Abbey, B. 2025. M – N Associates overhauls Vietnam's biggest sneaker brand. WWW document. Available at: https://www.creativeboom.com/news/m-n-associates-overhauls-vietnams-biggest-sneaker-brand/#:~:text=Biti%27s%20Hunter%20makes%20sneakers%20that,roots%20in%20the%20country%27s%20culture. [Accessed 05 Feb 2025]

Derek, A. 2023. A Brief History of TypeScript: From Origin to Modern Adoption. WWW document. Available at: https://medium.com/totally-typescript/a-brief-history-of-typescript-from-origin-to-modern-adoption-791368ec4b91 [Accessed 20 Apr 2023]

Introduction to TypeScript. WWW document. Available at: https://www.geeksforgeeks.org/introduction-to-typescript/ [Accessed 21 Jan 2025]

Introduction to Tailwind CSS. WWW document. Available at: https://www.geeksforgeeks.org/introduction-to-tailwind-css/ [Accessed 07 Oct 2024]

Ahmed Abu, B. 2023. WWW document. A comprehensive guide to NextJS. Available at: https://medium.com/@ahmed.nums345/a-comprehensive-guide-to-next-js-5f3b03b49def [Accessed 05 Sep 2023]

Rany, E. 2021. WWW document. What is Redux?. Available at: https://medium.com/swlh/what-is-redux-b16b42b33820#:~:text=Redux%20is%20simply%20a%20store,or%20read%20the%20store%20randomly. [Accessed 30 Jan 2021]

Viacheslav, O. 2024. WWW document. When and Why to use Node. Available at: https://dashdevs.com/blog/why-use-node-js-/ [Accessed 09 Jun 2024]

Express.js Tutorial. WWW document. Available at https://www.geeksforgeeks.org/express-js/ [Accessed 16 Dec 2024]

Mayur, P. 2024. WWW document. What is ExpressJS in NodeJS? – Backend Framework for Web Apps. Available at: https://www.excellentwebworld.com/what-is-expressjs-in-node-js/ [Accessed 17 Sep 2024]

Prisma Overview. Prisma, 2025. WWW document. Available at: https://www.prisma.io/docs/orm/prisma-schema/overview (n.d)

Moez, A. 2024. WWW document. What is PostgreSQL? How it works, use cases, and resources. Available at: https://www.datacamp.com/blog/what-is-postgresql-introduction [Accessed 03 Jul 2024]

PostgreSQL Documents. WWW document. What is PostgreSQL. Available at: https://www.postgresql.org/about/ (n.d.)

Yogesh, M. 2022. WWW document. Understanding pgAdmin 4 Architecture. Available at: https://www.enterprisedb.com/blog/understanding-pgadmin-4-architecture [Accessed 27 May 2022]

Paul, K. 2024. WWW document. What is AWS? Ultimate guide to Amazon Web Services. Available at: https://www.techtarget.com/searchaws/definition/Amazon-Web-Services [Accessed 15 Sep 2024]

Servifyspheresolutions. 2024. WWW document. What is AWS Amplify. Available at: https://medium.com/@servifyspheresolutions/what-is-aws-amplify-3d15a10fbc4a [Accessed 11 Jul 2024]

Pushkar, T. 2021. WWW document. What is AWS Amplify. Available at: https://dev.to/push9828/what-is-aws-amplify-4f28 [Accessed 24 Mar 2021]

Rahul, A. 2024. WWW document. What is AWS S3: Overview, Features and Storage Classes Explained. Available at: https://www.simplilearn.com/tutorials/aws-tutorial/aws-s3 [Accessed 10 Sep 2024]

What is Amazon VPC. WWW document. Available at: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html (n.d)

Simplilearn. 2024. WWW document. Dissecting AWS's Virtual Private Cloud (VPC). Available at: https://www.simplilearn.com/tutorials/aws-tutorial/aws-vpc [Accessed 10 Sep 2024]

Servifyspheresolutions. 2024. WWW document. What is Amazon EC2?. Available at: https://medium.com/@servifyspheresolutions/what-is-amazon-ec2-5e21dcc53bb1 [Accessed 04 Feb 2024]

What is Elastic Compute Cloud (EC2). WWW document. Available at: https://www.geeksforgeeks.org/what-is-elastic-compute-cloud-ec2/ [Accessed 03 Jul 2024]

Jon, W. 2021. WWW document. What is Amazon RDS and How It Works. Available at: https://www.cbtnuggets.com/blog/technology/devops/what-is-amazon-rds-and-how-it-works [Accessed 14 Oct 2021]

Amazon RDS – Introduction to Amazon Relational Database System. WWW document. Available at: https://www.geeksforgeeks.org/amazon-rds-introduction-to-amazon-relational-database-system/ [Accessed 04 Jan 2025]

Amazon Web Service – Introduction to API Gateway. WWW document. Available at: https://www.geeksforgeeks.org/amazon-web-service-introduction-to-api-gateway/ [Accessed 08 Dec 2023]