SUMMER INTERNSHIP PROJECT REPORT

BACKEND-DESIGN

Prepared By:
Majolika Singh
B.Tech. III-Year

Project Guide:
Mr. Paul Mickey Toppo
Sr. Programming Officer,
Regional Computer Centre
ONGC , Vadodara.

15 TH MAY 2024 - 14TH JUNE 2024

INDEX

Acknowledgement

I would like to sincerely express my deep appreciation to Shri PR Mishra, Executive Director - Basin Manager, Western Onshore Basin, and the other authorities of Oil and Natural Gas Corporation Ltd., Vadodara. I am immensely grateful to Mr. Maharaj Singh, GM (Geophysics-Surface), I/c RCC, WON Basin , ONGC Vadodara, and my mentor, Mr. Paul Mickey Toppo, Sr. Programming Officer, for his valuable time they provided me to undertake my summer internship project work in this esteemed organization.

I would like to express my heart felt gratitude to Ms. Kanika and Mr. Prem Prakash Hansda for their invaluable guidance and support throughout my summer training project. Under their expert mentorship, I have gained tremendous knowledge and hands-on experience in the field. Their dedication, patience, and commitment to my growth have been truly inspiring.

Ms. Kanika's, Sr. Programming Officer, expertise and insightful feedback have immensely contributed to my understanding of the project domain. Her guidance and great mentorship have helped shape my skills and problem-solving abilities.

I am also grateful to Mr. Prem Prakash Hansda, Programming Officer, for his constant support and encouragement, his vast experience and practical approach has provided me with valuable insights and perspectives. The mentorship of all the mentors who has been instrumental in honoring my technical skills and expanding my horizons.

I would like to thank Sri Goutham Chand Chopra, GGM (Mechanical), Head- Skill Development Centre, Vadodara for considering my name for training in this Prestigious Organization.

I am deeply grateful to my parents for their unwavering love, support, and sacrifices, which have been the foundation of my success. Their guidance and belief in me have shaped my character and inspired me to pursue my dreams with determination and resilience.

Organization Background

Oil and Natural Gas Corporation Limited (ONGC) is an Indian multinational oil and gas company headquartered at Dehradun, Uttarakhand, India. Established in 1956, ONGC is a state-owned enterprise and operates as a Public Sector Undertaking (PSU) under the administrative control of the Ministry of Petroleum and Natural Gas.

ONGC is involved in the exploration, development, and production of oil and gas resources both in India and abroad. It is one of the largest oil and gas exploration and production companies in the world and plays a significant role in meeting India's energy requirements. ONGC's operations span various domains, including upstream exploration and production, refining, marketing, and petrochemicals.

The company operates a diverse portfolio of oil and gas assets, including onshore and offshore fields, and has made significant contributions to India's energy security. ONGC has been involved in several major oil and gas discoveries in India and has expanded its operations globally through strategic partnerships and acquisitions.

ONGC is known for its technological expertise, innovation, and commitment to environmental sustainability. The company has undertaken numerous initiatives in the areas of renewable energy, research and development, and corporate social responsibility.

With a strong workforce comprising highly skilled professionals, ONGC continues to play a pivotal role in India's oil and gas industry, contributing to the nation's economic growth and energy self-sufficiency.

ONGC's rich history, extensive operations, technological expertise, and commitment to sustainability position it as a key player in India's energy sector and a significant contributor to the country's economic growth and energy security.

Introduction

This report summarizes my experience and insights gained during the summer training program. I had the privilege to undergo at Oil and Natural Gas Corporation Limited (ONGC). It aims to provide a comprehensive overview of the training, highlighting the key aspects of ONGC's operations, the valuable learning opportunities it presented, and the impact it had on my professional growth.

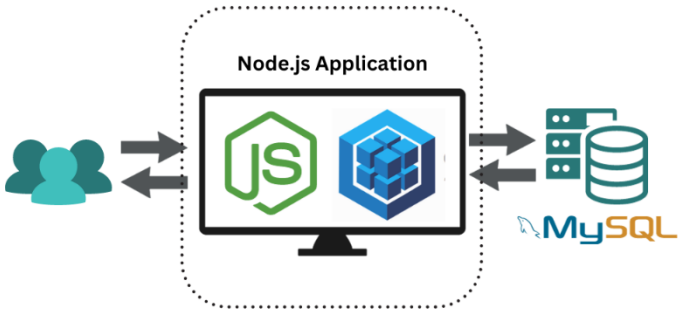
Backend-Design

Backend of Website Development

The backend of website development refers to the server-side part of a web application. It is responsible for managing the business logic, database interactions, user authentication, and other processes that are not directly visible to the user. The backend ensures that the frontend (the part of the application the user interacts with) can function smoothly and securely.

Key Components of Backend Development

1. **Server:**
   - **Role:** The server is a system that handles client requests, processes them, and sends back the appropriate response. It hosts the backend application and ensures that it is accessible to users via the internet.
   - **Technologies:** Apache, Nginx, Microsoft IIS.
2. **Application Logic:**
   - **Role:** This is where the core functionality of the application resides. It includes the business logic, which dictates how data is processed, stored, and retrieved.
   - **Technologies:** Node.js, Python (Django, Flask), Ruby on Rails, Java (Spring), PHP (Laravel, Symfony).
3. **Database:**
   - **Role:** The database stores and manages the application's data. Backend code interacts with the database to perform CRUD operations (Create, Read, Update, Delete).
   - **Technologies:** MySQL, PostgreSQL, MongoDB, SQLite, Redis.
4. **APIs (Application Programming Interfaces):**
   - **Role:** APIs define how different software components should interact. They allow the backend to communicate with the frontend, other backends, or third-party services.
   - **Technologies:** RESTful APIs, GraphQL, SOAP.
5. **Authentication and Authorization:**
   - **Role:** This involves verifying user identities (authentication) and determining what resources or operations a user can access (authorization).
   - **Technologies:** JWT (JSON Web Tokens), OAuth, Passport.js.
6. **Middleware:**
   - **Role:** Middleware functions act as intermediaries that process requests before they reach the final endpoint. They can handle tasks like logging, authentication, and error handling.
   - **Technologies:** Express.js (Node.js).



Responsibilities of Backend Development

1. **Processing Requests:**
   - Handling HTTP requests from the client (browser) and routing them to the appropriate part of the application.

2. **Database Management:**
   - Interacting with the database to fetch, store, update, and delete data as required by the application.

3. **User Management:**
   - Implementing user authentication (login, registration) and authorization (access control).

4. **Data Security:**
   - Ensuring that data is transmitted securely and stored safely, protecting against vulnerabilities like SQL injection and cross-site scripting (XSS).

5. **Performance Optimization:**
   - Ensuring that the backend operates efficiently, with fast response times and the ability to handle high traffic loads.

Technologies and Tools

1. **Programming Languages:**
    - **JavaScript:** Node.js
    - **Python:** Django, Flask
    - **Ruby:** Ruby on Rails
    - **PHP:** Laravel, Symfony
    - **Java:** Spring

2. **Frameworks:**
    - **Express.js (Node.js):** A minimal and flexible Node.js web application framework.
    - **Django (Python):** A high-level Python web framework that encourages rapid development and clean, pragmatic design.

3. **Databases:**
    - **SQL Databases:** MySQL, PostgreSQL, SQLite.
    - **NoSQL Databases:** MongoDB, Redis, Cassandra.

4. **APIs and Web Services:**
    - **RESTful APIs:** Using HTTP requests to GET, PUT, POST, and DELETE data.
    - **GraphQL:** A query language for your API that enables clients to request exactly the data they need.

5. **Authentication Libraries:**
    - **Passport.js:** Authentication middleware for Node.js.
    - **JWT (JSON Web Tokens):** A compact, URL-safe means of representing claims to be transferred between two parties.

Why NodeJS and ExpressJS?

Node.js and Express.js have gained significant popularity in the web development community due to several advantages they offer over other frameworks. Here's an in-depth look at why Node.js and Express.js are often considered better choices for many developers and projects:

Advantages of Node.js

1. **Asynchronous and Event-Driven:**
   - **Non-Blocking I/O:** Node.js uses a non-blocking, event-driven architecture, which allows it to handle multiple operations concurrently. This makes it highly efficient for I/O-bound applications, such as web servers that handle many requests at once.

2. **JavaScript Everywhere:**
   - **Single Language for Frontend and Backend:** With Node.js, developers can use JavaScript for both client-side and server-side development, promoting code reuse and simplifying the development process.

3. **Performance:**
   - **V8 Engine:** Node.js is built on the V8 JavaScript engine from Google Chrome, which compiles JavaScript to machine code, resulting in fast execution.

4. **Rich Ecosystem:**
   - **NPM (Node Package Manager):** Node.js has a vast ecosystem of libraries and modules available through NPM, making it easy to add functionality and integrate third-party services.

Advantages of Express.js

1. **Middleware:**
    - **Middleware:** Express.js uses middleware functions to handle various tasks, such as logging, authentication, and error handling. This modular approach makes it easy to add and configure middleware for specific needs. ‘

2. **Routing:**
    - **Advanced Routing Capabilities:** Express.js provides a robust routing system that allows for defining routes for different HTTP methods, creating dynamic routes with parameters, and handling complex URL patterns.

3. **Integration with Node.js:**
    - **Seamless Integration:** Express.js is built on top of Node.js, leveraging its performance and asynchronous capabilities. This combination provides a powerful environment for building fast and scalable applications.
    - **Full Access to Node.js APIs:** Developers can use all the underlying Node.js APIs and features within an Express.js application.

4. **Community and Ecosystem:**
    - **Wide Adoption:** Express.js is one of the most widely used web frameworks for Node.js, leading to a rich ecosystem of middleware, plugins, and extensions.

Conclusion:

Node.js and Express.js offer a compelling combination of performance, flexibility, and ease of use, making them suitable for a wide range of applications, from simple web servers to complex, real-time applications. Their asynchronous, event-driven architecture, combined with the unified use of JavaScript, rich ecosystem, and strong community support, often make them a better choice for modern web development compared to other frameworks.

To add Expressjs and NodeJS in our systems we have the following commands:

Nodejs:

```
PS C:\Users\new user\Desktop\project1.0> npm init -y
```

Expressjs:

```
PS C:\Users\new user\Desktop\project1.0> npm install express
>>

up to date, audited 129 packages in 1s

17 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\new user\Desktop\project1.0>
```

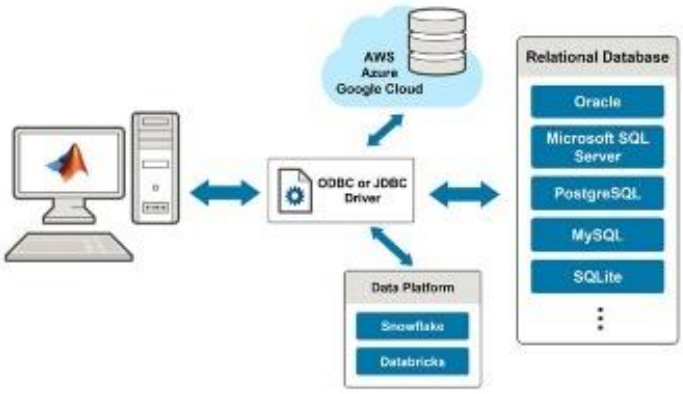
Nodemon (for auto-reloading):

```
PS C:\Users\new user\Desktop\project1.0> npm install --save-dev nodemon
>>

up to date, audited 129 packages in 1s

17 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\new user\Desktop\project1.0>
```

Database Representation

What Are Databases?

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. Databases are managed using Database Management Systems (DBMS), which provide an interface for users and applications to interact with the data.



Databases are the essential data repository for all software applications

The following are the types of databases:

Types of Databases and Their Uses

1. Relational Databases (RDBMS):
   - Description: These databases store data in tables with rows and columns. Each table represents a different entity, and tables can be related to each other through keys.
   - Examples: MySQL, PostgreSQL, Oracle, Microsoft SQL Server.
2. NoSQL Databases:
   - Examples: MongoDB (document store), Redis (key-value store), Cassandra (wide-column store), Neo4j (graph database).
3. In-Memory Databases:
   - Examples: Redis, Memcached.
4. Time-Series Databases:
   - Examples: Influx DB, Timescale DB.
5. Object-Oriented Databases:
   - Examples: db4o, Object DB.
6. Graph Databases:
   - Examples: Neo4j, Amazon Neptune.

MySQL Data base : -

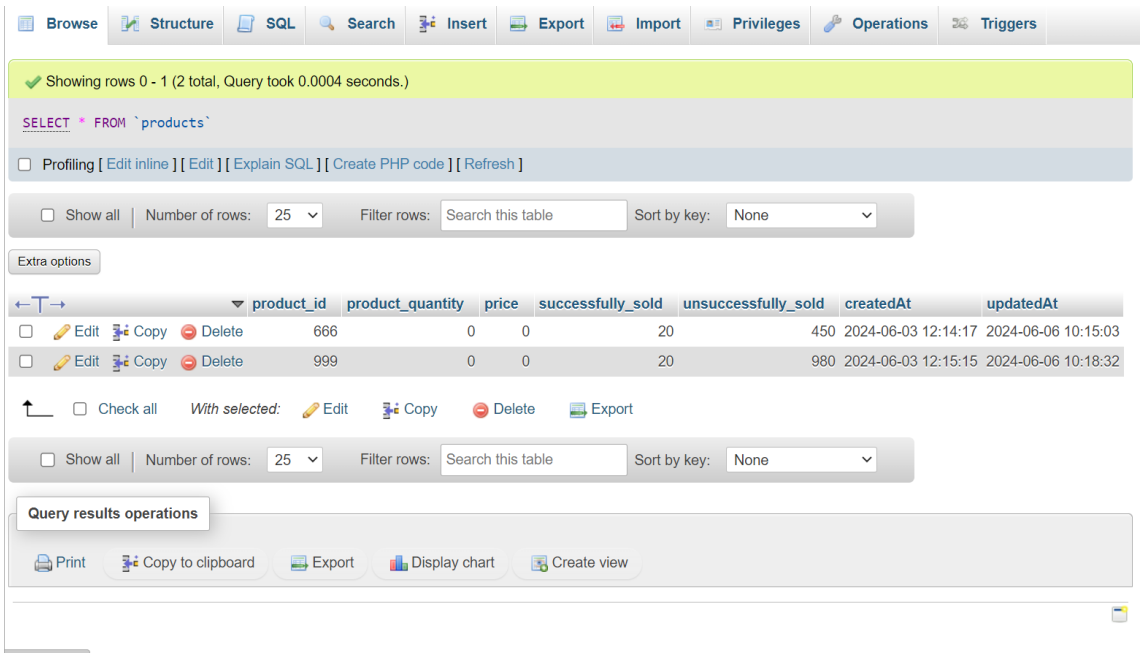
PhpMyAdmin has been utilized in this project as a reliable and efficient tool for managing the MySQL database. It offers a user-friendly web interface that simplifies tasks such as creating, modifying, and querying the database.

➔ Benefits of using phpMyAdmin database system: -

1. User-Friendly
2. Efficient Database Management
3. Simplified SQL Execution
4. Comprehensive Table Management

User Privilege Settings

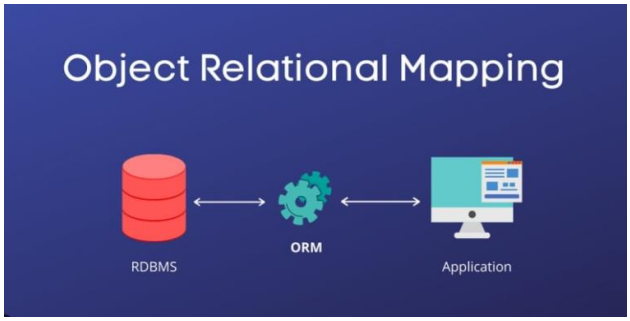
Table created in MySQL(phpMyAdmin) database:



Object Relational Mapping (ORM)

What is an ORM Tool?

An ORM tool is software designed to help OOP developers interact with relational databases. So instead of creating your own ORM software from scratch, you can make use of these tools



Why Use ORMs?

1. Abstraction:
    - ORMs provide an abstraction layer over raw SQL, allowing developers to interact with the database using object-oriented programming principles.

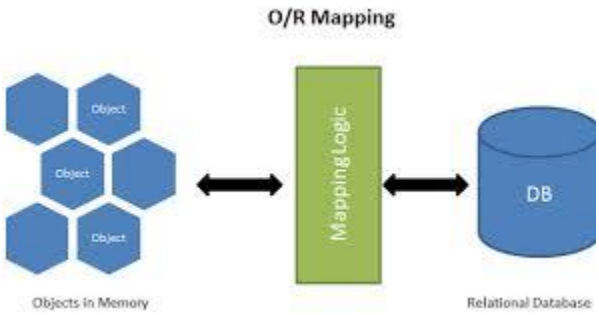
2. Productivity:
    - Developers can write less code to perform CRUD (Create, Read, Update, Delete) operations, which enhances the productivity.

3. Portability:
    - ORMs can facilitate database migration and allow applications to switch between different database systems with minimal code changes.

4. Security:

- By using ORMs, developers can reduce the risk of SQL injection attacks since the ORM library handles query construction.

5. Consistency:
    - ORMs help ensure consistency in database interactions by providing a unified approach to database operations.



Different Types of ORMs for Node.js and Express.js

Here are some popular ORMs that can be used with Node.js and Express.js:

1. Sequelize:
    - Sequelize is a promise-based ORM for Node.js and supports multiple databases like MySQL, PostgreSQL, SQLite, and MSSQL.
2. TypeORM:
    - TypeORM is an ORM that supports TypeScript and JavaScript (ES7) and works with MySQL, MariaDB, PostgreSQL, SQLite, Microsoft SQL Server, and Oracle.
3. Objection.js:
    - Objection.js is a SQL-friendly ORM for Node.js based on the SQL query builder Knex.js.
4. Bookshelf.js:
    - Bookshelf.js is an ORM built on top of Knex.js that supports MySQL, PostgreSQL, and SQLite.
5. Mongoose (for MongoDB):

    - Mongoose is an ODM (Object Data Modelling) library for MongoDB and Node.js.

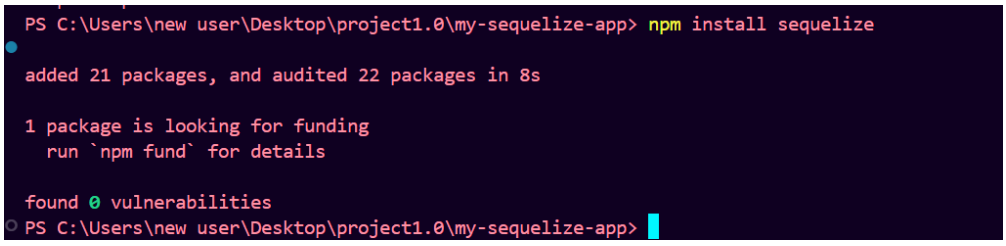
Sequelize

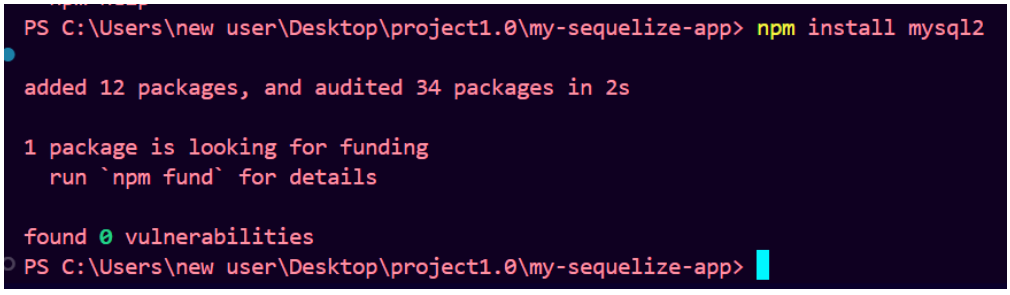
Sequelize is a popular ORM for Node.js and Express.js that offers several advantages compared to other ORMs. Here are some advantages of using Sequelize:

Advantages of Using Sequelize

- Compatibility: Sequelize supports multiple SQL databases, including MySQL, PostgreSQL, SQLite, and MSSQL, making it a versatile choice for different projects.
- Migration Support: It provides a powerful migration toolset, allowing developers to manage database schema changes over time seamlessly.
- Asynchronous Operations: Sequelize uses promises for asynchronous operations, which integrates well with modern JavaScript codebases and ensures non-blocking database interactions.
- Cleaner Code: Promises and async/await syntax lead to cleaner, more readable, and maintainable code.
- Query Builder: Sequelize offers a flexible query builder, allowing complex queries to be constructed easily.

Below given is the procedure to install sequelize in vscode:





Migrations

Migrations in Sequelize are a way to manage and version control your database schema changes. They allow developers to evolve the database schema over time, applying changes incrementally and ensuring consistency across different environments (development, testing, production).

What are Migrations?

Migrations are files that contain some actions to evolve your database schema over time. Each migration file describes a version of your database schema and how to get to that version.

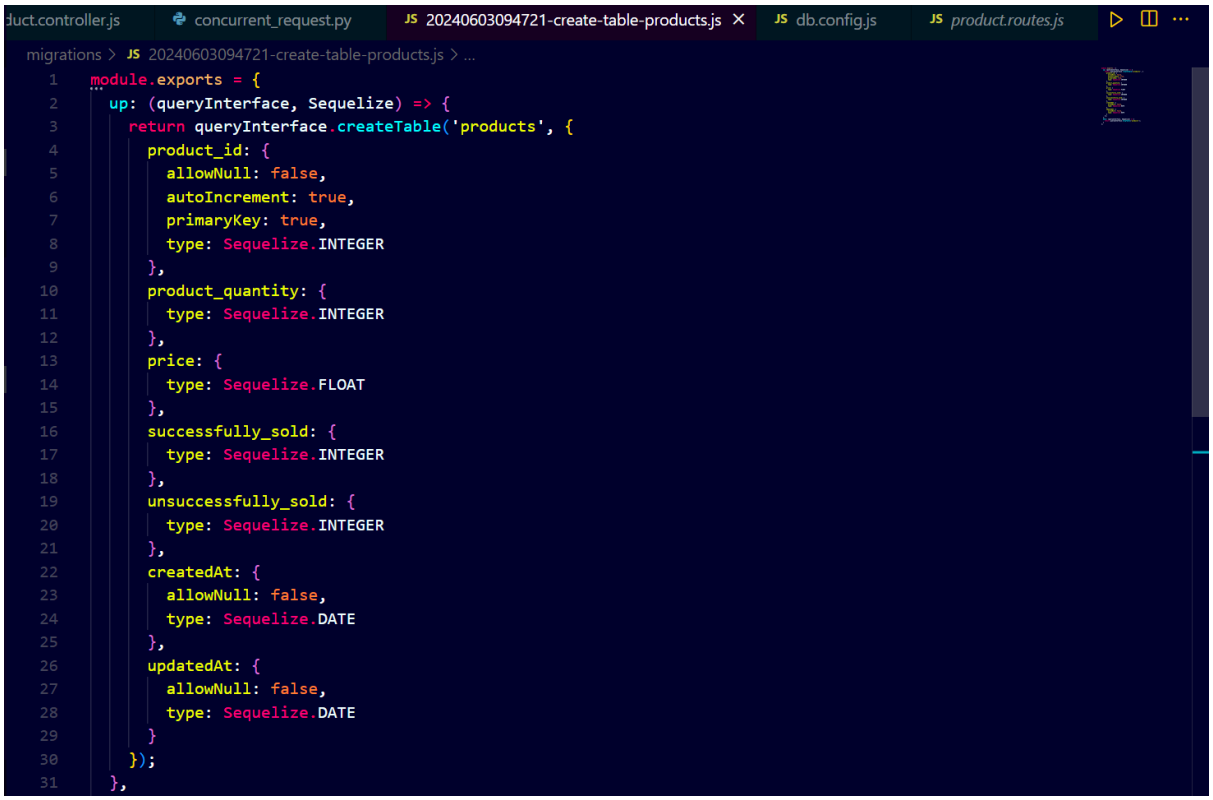
Why Use Migrations?

- Version Control: Keep track of changes to the database schema over time.
- Consistency: Ensure that all developers and environments (e.g., staging, production) have the same schema.
- Rollback Capability: Easily revert changes if a migration introduces an issue.
- Automation: Automate the process of applying schema changes as part of your deployment pipeline.

Conclusion:

Migrations in Sequelize are an essential tool for managing and versioning database schema changes. They provide a structured way to evolve the database schema over time, ensuring consistency and ease of management. By using migrations, you can automate the process of applying changes to the database, rollback changes if necessary, and maintain a clear history of schema evolution.

Creating a Migration file:



Creating a Model: represents the data and the rules to manipulate that data, creating a model helps you interact with the table in your code. Generate a model using Sequelize.

Index.js:

```
const sequelize = require('../configdb/db.config');
const Product = require('./product');

module.exports = {
  sequelize,
  Product
};

```

Product.js:

```
models > JS product.js > ...
const Sequelize = require('sequelize');
const db = require('../configdb/db.config');

const Product = db.define('product', {
  product_id: {
    type: Sequelize.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  product_quantity: {
    type: Sequelize.INTEGER
  },
  price: {
    type: Sequelize.FLOAT
  },
  successfully_sold: {
    type: Sequelize.INTEGER
  },
  unsuccessfully_sold: {
    type: Sequelize.INTEGER
  },
  version: {
    type: Sequelize.INTEGER,
    defaultValue: 0
  }
});

module.exports = {Sequelize,Product};
```

Controllers

Controllers: Controllers handle the user's interactions with the View, process these interactions, and update the Model if necessary.

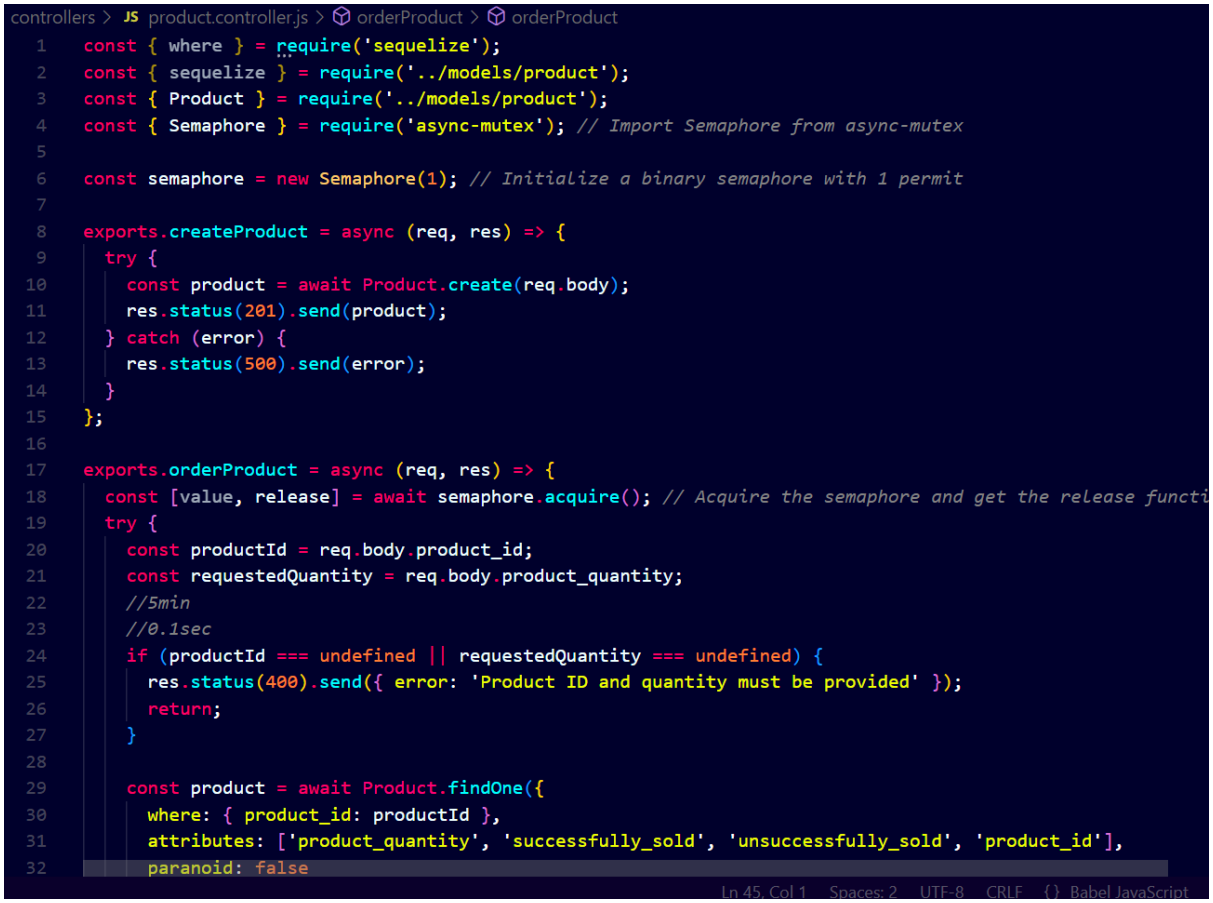
Responsibilities of Controllers

1. Handling Requests:
    - Controllers receive incoming HTTP requests and determine the appropriate action to take based on the request's URL and method (GET, POST, PUT, DELETE, etc.).
2. Processing Data:
    - They process any data included in the request, such as query parameters, body data, or headers. This often involves validating and sanitizing the data before using it.
3. Interacting with Models:
    - Controllers often interact with models, which represent the application's data structure and business logic. This interaction typically involves querying the database, updating records, or performing other data operations.
4. Sending Responses:
    - After processing the request and interacting with the necessary models, controllers send back a response to the client. This response can be in various formats, such as HTML, JSON, XML, etc.
5. Error Handling:
    - Controllers are also responsible for handling errors that occur during request processing and providing meaningful error messages to the client.

Uses of Controllers

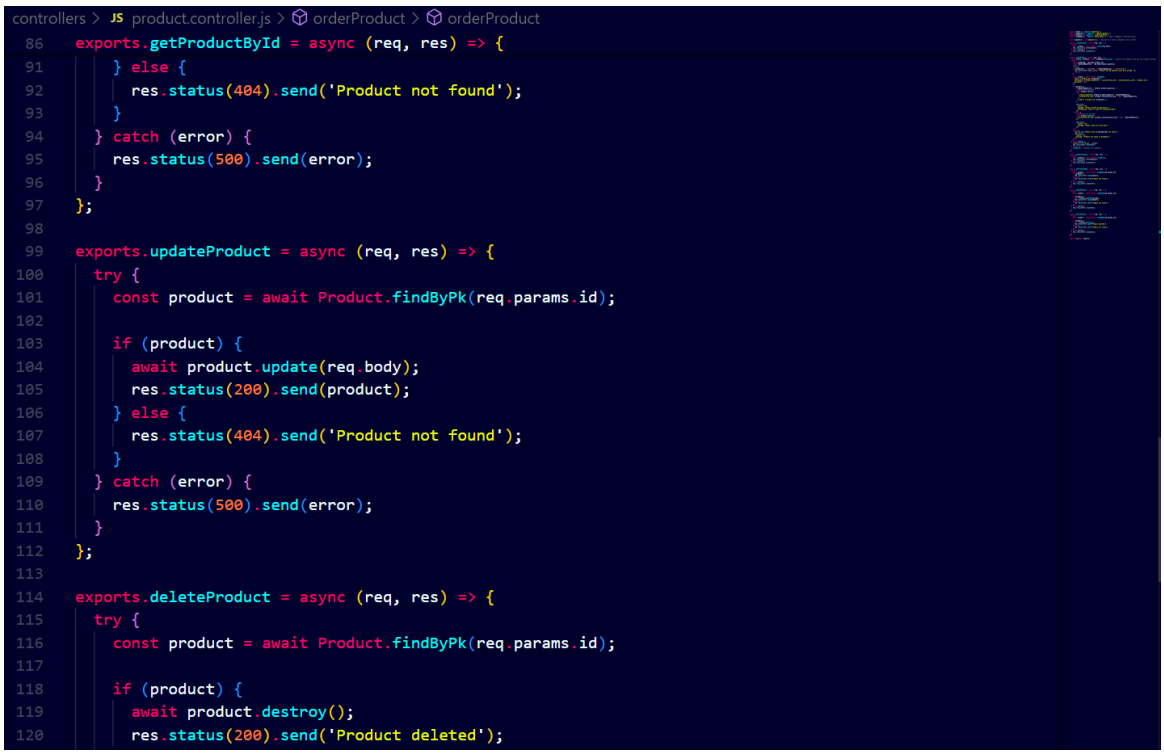
1. Organizing Code:
    - Controllers help in organizing the codebase by separating the concerns of handling requests, processing data, and interacting with the database. This leads to more maintainable and readable code.
2. Implementing Logic:
    - They contain the business logic necessary to perform specific actions based on the user's input and application requirements.
3. Routing Requests:
    - Controllers are linked with routes, which define how URL paths map to specific controller actions. This allows the application to handle different URLs and HTTP methods in an organized manner.

Products.controllers.js:

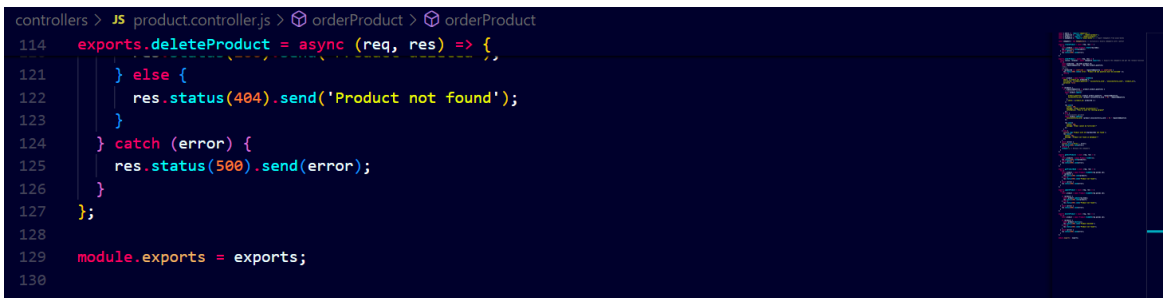


```
controllers > JS product.controller.js > orderProduct > orderProduct
17  exports.orderProduct = async (req, res) => {
29      const product = await Product.findOne({
33      });

35      if (product) {
36        if (requestedQuantity <= product.product_quantity) {
37          // Successful purchase
38          await product.update(
39            {
40              product_quantity: product.product_quantity - requestedQuantity,
41              successfully_sold: (product.successfully_sold || 0) + requestedQuantity
42            },
43            { where: { product_id: productId } }
44          );
45
46          res.json({
47            status: 200,
48            message: "Items ordered successfully!!",
49            information: "This is just for testing purpose"
50          });
51        } else {
52          // Unsuccessful purchase
53          await product.update({
54            unsuccessfully_sold: (product.unsuccessfully_sold || 0) + requestedQuantity
55          });

56
57          res.json({
58            status: 200,
59            message: "Order cannot be fulfilled!!"
60          });
```

```
controllers > JS product.controller.js > orderProduct > orderProduct
17  exports.orderProduct = async (req, res) => {
61        }
62      } else {
63        console.log(`Product with ID ${productId} not found`);
64        res.json({
65          status: 401,
66          message: "Product not found in database!!"
67        });
68      }
69    } catch (error) {
70      console.error('Error:', error);
71      res.status(500).send(error);
72    } finally {
73      release(); // Release the semaphore
74    }
75  };

76
77  exports.getAllProducts = async (req, res) => {
78    try {
79      const products = await Product.findAll();
80      res.status(200).send(products);
81    } catch (error) {
82      res.status(500).send(error);
83    }
84  };

85
86  exports.getProductById = async (req, res) => {
87    try {
88      const product = await Product.findByPk(req.params.id);
89      if (product) {
90        res.status(200).send(product);
```



```
controllers > JS product.controller.js > orderProduct > orderProduct
86  exports.getProductById = async (req, res) => {
91      } else {
92        res.status(404).send('Product not found');
93      }
94    } catch (error) {
95      res.status(500).send(error);
96    }
97  };
98
99  exports.updateProduct = async (req, res) => {
100   try {
101     const product = await Product.findByPk(req.params.id);
102
103     if (product) {
104       await product.update(req.body);
105       res.status(200).send(product);
106     } else {
107       res.status(404).send('Product not found');
108     }
109   } catch (error) {
110     res.status(500).send(error);
111   }
112 };
113
114 exports.deleteProduct = async (req, res) => {
115   try {
116     const product = await Product.findByPk(req.params.id);
117
118     if (product) {
119       await product.destroy();
120       res.status(200).send('Product deleted');
```



```
controllers > JS product.controller.js > orderProduct > orderProduct
114 exports.deleteProduct = async (req, res) => {
121     } else {
122       res.status(404).send('Product not found');
123     }
124   } catch (error) {
125     res.status(500).send(error);
126   }
127 };
128
129 module.exports = exports;
130
```

Race condition

A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time.

Why Do We Need to Eliminate Race Conditions?

1. **Data Integrity:** Race conditions can corrupt data by allowing simultaneous access to shared resources, leading to inconsistent states.
2. **Consistency:** Ensuring operations on shared resources occur in a controlled and consistent manner is critical for the correctness of a program.
3. **Predictability:** Race conditions make the behaviour of the software unpredictable, which is undesirable for testing and debugging.
4. **Security:** In some cases, race conditions can be exploited to cause security vulnerabilities, such as bypassing authentication mechanisms.

How Can We Eliminate Race Conditions?

To eliminate race conditions, we need to ensure that shared resources are accessed in a controlled manner. This involves synchronizing access to these resources so that only one process or thread can access them at a time. Common synchronization mechanisms include semaphores and mutexes.

Semaphores:

- **Definition:** A semaphore is a more general synchronization primitive that can control access to a shared resource by multiple threads based on a counter.
    - **Binary Semaphore (Similar to Mutex):** It can have a value of 0 or 1 and works similarly to a mutex.
    - **Counting Semaphore:** It allows a specific number of threads to access a resource concurrently.

In this project we have included binary semaphore to remove the problem of race condition:

```
const { Semaphore } = require('async-mutex'); // Import Semaphore from async-mutex

const semaphore = new Semaphore(1); // Initialize a binary semaphore with 1 permit

exports.createProduct = async (req, res) => {
  try {
    const product = await Product.create(req.body);
    res.status(201).send(product);
  } catch (error) {
    res.status(500).send(error);
  }
};

exports.orderProduct = async (req, res) => {
  const [value, release] = await semaphore.acquire(); // Acquire the semaphore and get the release functio
  try {
    const productId = req.body.product_id;
    const requestedQuantity = req.body.product_quantity;
    //5min
    //0.1sec
    if (productId === undefined || requestedQuantity === undefined) {
      res.status(400).send({ error: 'Product ID and quantity must be provided' });
      return;
    }
```

```
    if (productId === undefined || requestedQuantity === undefined) {
      res.status(400).send({ error: 'Product ID and quantity must be provided' });
      return;
    }

    const product = await Product.findOne({
      where: { product_id: productId },
      attributes: ['product_quantity', 'successfully_sold', 'unsuccessfully_sold', 'product_id'],
      paranoid: false
    });

    if (product) {
      if (requestedQuantity <= product.product_quantity) {
        // Successful purchase
        await product.update(
          {
            product_quantity: product.product_quantity - requestedQuantity,
            successfully_sold: (product.successfully_sold || 0) + requestedQuantity
          },
          { where: { product_id: productId } }
        );

        res.json({
          status: 200,
          message: "Items ordered successfully!!",
          information: "This is just for testing purpose"
        });
```

Routes

Routes in Node.js, Express.js, and Sequelize projects define the endpoints of the application and specify how incoming HTTP requests should be handled. They play a crucial role in defining the application's API (Application Programming Interface) and determining how clients interact with the server.

Role of Routes: The routing methods of Express.js are derived from HTTP methods, and they include get, post, put, delete, etc. They handle different URL paths and HTTP methods, which correspond to different endpoints.

Benefits of Using Routes:

- Routes help to keep the code organized by separating the handling of different endpoints.
- They allow you to easily define how your application should respond to different types of client requests.
- They make your application more scalable and easier to maintain.

routes > JS product.routes.js > ...

```
const express = require('express');
const router = express.Router();
const productController = require('../controllers/product.controller');

router.post('/', productController.createProduct);
router.get('/', productController.getAllProducts);
router.get('/:id', productController.getProductById);
router.put('/:id', productController.updateProduct);
router.delete('/:id', productController.deleteProduct);
router.post('/orderProduct', productController.orderProduct);

module.exports = router;
```

Database Configuration: It contains the necessary configuration to connect to a MySQL database.

configdb > JS db.config.js > ...

```
const Sequelize = require('sequelize');
module.exports = new Sequelize('product@test', 'root', 'database@123', {
  host: 'localhost',
  dialect: 'mysql'
});
```

The `config.json` file typically includes the following sections:

1. Configuration used during development.
2. Configuration used for running tests.
3. Configuration used in the production environment.

config > {} config.json > ...

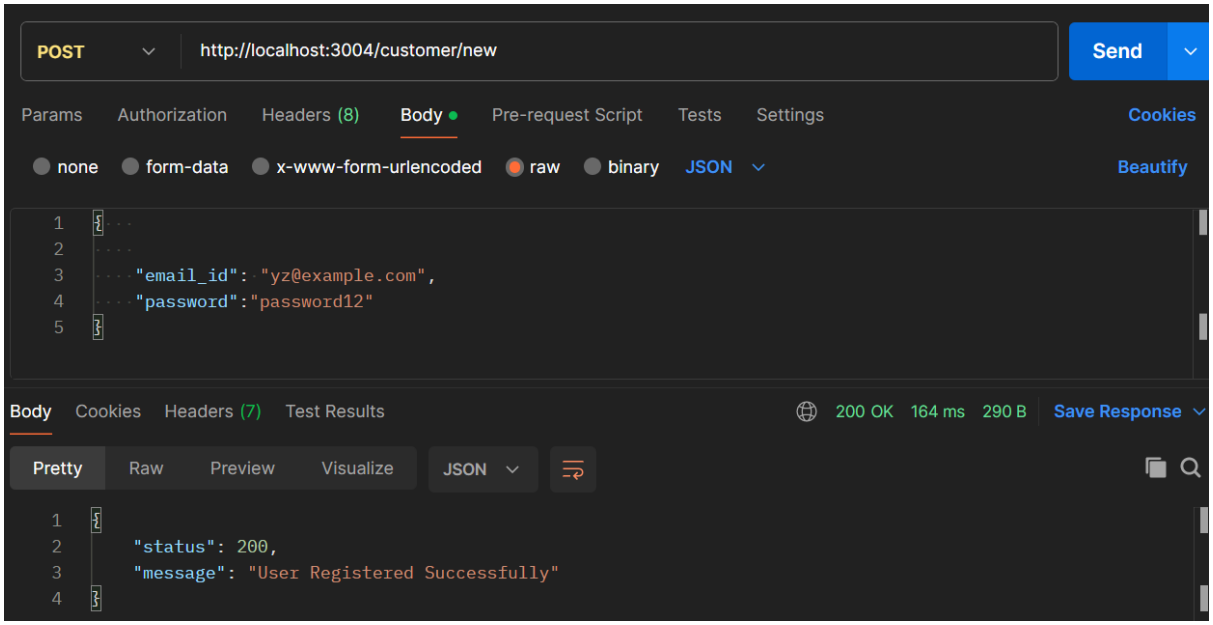
```
{
  "development": {
    "username": "root",
    "password": "database@123",
    "database": "product@test",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
  "test": {
    "username": "root",
    "password": "database@123",
    "database": "product@test",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
  "production": {
    "username": "root",
    "password": "database@123",
    "database": "product@test",
    "host": "127.0.0.1",
    "dialect": "mysql"
  }
}
```

Authentication

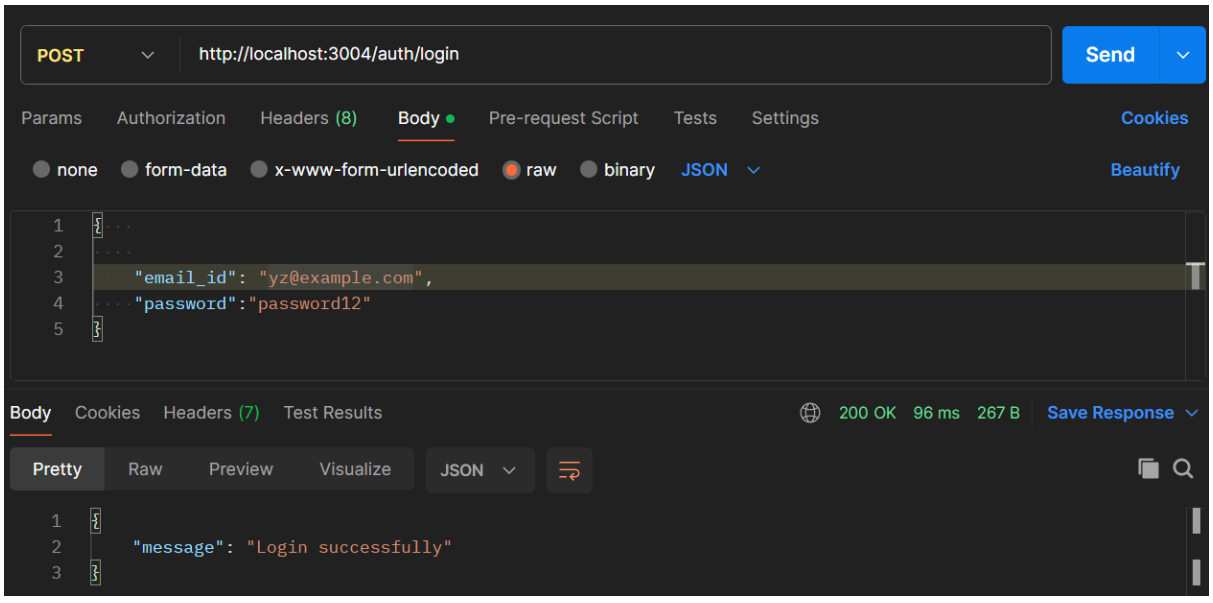
Authentication is a process that verifies the identity of a user, device, or system. It involves validating their credentials against the stored data to ensure that they are who they claim to be.

In the context of a website or an application, authentication typically involves a user entering their username and password. If the entered credentials match the stored data, the user is granted access. This process helps to protect sensitive data and resources from unauthorized access.

Here we have used postman to check if the code introduced for authentication in our project is functional or not:




```
async function login(req, res) {
    const { email_id, password } = req.body;

    try {
        const user = await Customers.findOne({ where: { email_id: email_id } });
        if (user) {
            const match = await argon2.verify(user.password, password);
            if (match) {
                res.status(200).json({ message: 'Login successfully' });
            } else{
                res.status(401).json({ message: 'Username or password is wrong' });
            }
        } else {
            res.status(404).json({ message: 'Username does not exist.' });
        }
    } catch (error) {
        res.status(500).json({ message: error});
    }
}
```



Password Hashing

Password hashing is a process of transforming a plain-text password into a fixed-length string of characters, which is typically a digest that represents the original password.

Hashing is a one-way function, meaning that it's computationally infeasible to reverse the process and retrieve the original password from the hash.

This is crucial for securely storing passwords, as even if the password database is compromised, the actual passwords are not directly exposed.

Why Hash Passwords?

1. **Security**: Plain-text passwords can be easily stolen and misused. Hashing protects the actual password even if the hashed value is exposed.
2. **Integrity**: Hashing ensures that the password has not been tampered with.
3. **Compliance**: Many regulations require sensitive data, such as passwords, to be stored securely.

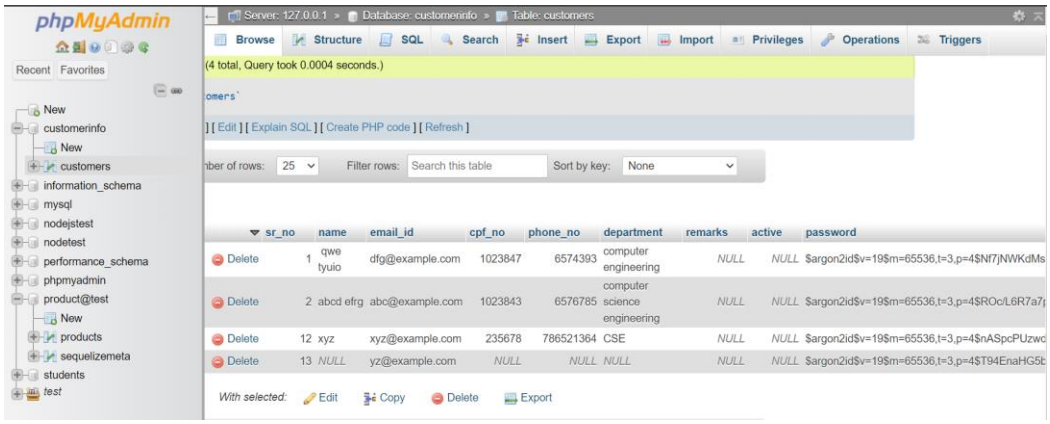
To include hashing in the project we have used argon2 algorithm.

Argon2 Algorithm

Argon2 is a modern password-hashing algorithm. It is designed to be highly secure and efficient, offering protection against various types of attacks, including brute force and side-channel attacks.

```
// Hash the password with argon2
try {
    const hash = await argon2.hash(req.body.password);
    const user_info_object = {
        name: req.body.name,
        email_id: req.body.email_id,
        cpf_no: req.body.cpf_no,
        phone_no: req.body.phone_no,
        department: req.body.department,
        active: req.body.active,
        password: hash  // Store the full hash
    };
```

```
async function update_user_info(req, res) {
    const new_data = {
        name: req.body.name,
        email_id: req.body.email_id,
        cpf_no: req.body.cpf_no,
        phone_no: req.body.phone_no,
        department: req.body.department,
        active: req.body.active,
    };

    // If password is provided, hash it before updating
    if (req.body.password) {
        try {
            new_data.password = await argon2.hash(req.body.password);
        } catch (error) {
            return res.status(500).send({
                message: "An error occurred while hashing the password."
            });
        }
    }
```



Server.js file:

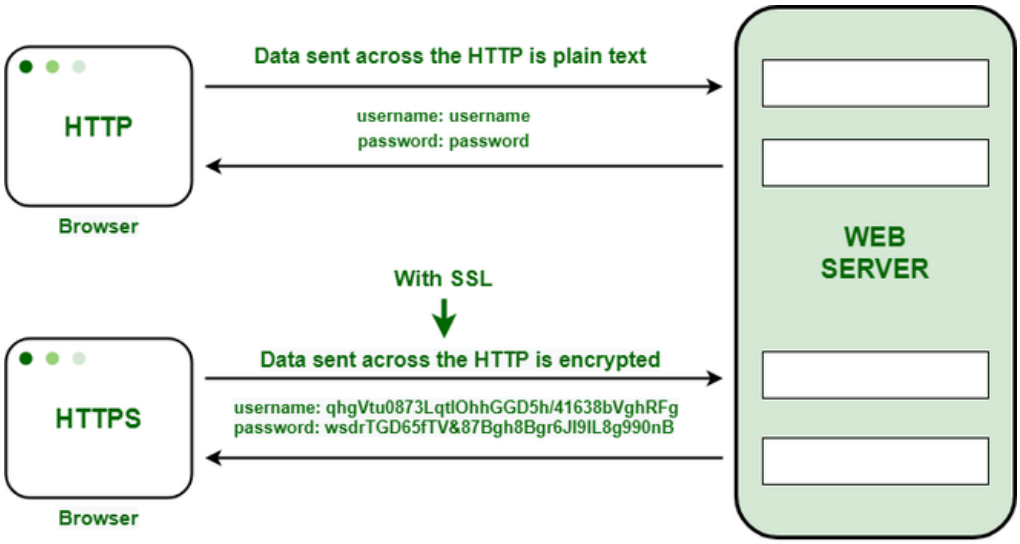
```
server.js > app.listen() callback
1   const express = require('express');
2   const app = express();
3   const bodyParser = require('body-parser');
4   app.use(bodyParser.json());
5
6   const db = require('./config/db.config.js');
7
8   db.sequelize.sync();
9
10  const controller = require('./controller/customer.controller.js');
11  const router = require('./routes/auth.js');
12
13  // route
14  app.use('/auth', router);
15
16  app.post('/customer/new', function (req, res) {
17      controller.create_user_info(req, res);
18  });
19
20  app.get('/userinfo/:cpf_no', function (req, res) {
21      controller.find_user_info_by_cpf_no(req, res);
22  });
23
24  app.put('/userinfo/update', function (req, res) {
25      controller.update_user_info(req, res);
26  });
27
28  app.listen(3004, () => {
29      console.log("The server is running on port 3004");
30  });
```

HTTPS:

HTTPS uses SSL/TLS to encrypt data, providing a secure connection over the internet.

HTTPS helps verify the authenticity of the website, ensuring users they are visiting the legitimate website.

Data integrity is maintained, ensuring that data cannot be modified or corrupted during transfer.



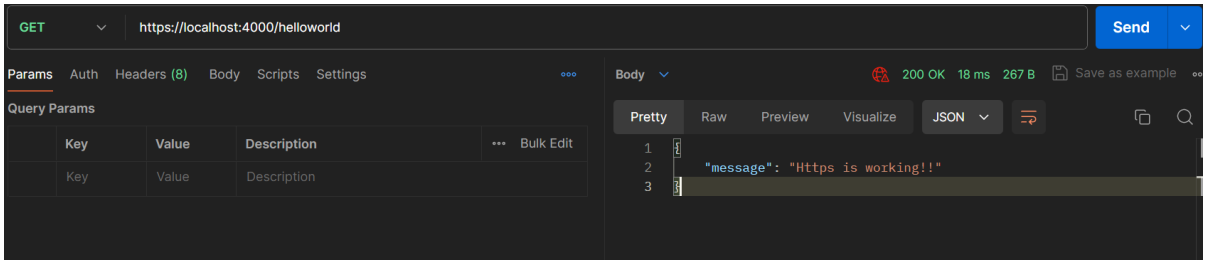
WORKING:

• SSL/**TLS Certificates**: Websites use SSL/TLS certificates to enable HTTPS. These certificates are issued by trusted certificate authorities (CAs) and contain the public key and the identity of the website owner.

• Handshake **Process**: When a browser connects to a website via HTTPS, an SSL/TLS handshake process is initiated. During this process, the server and client agree on encryption methods and exchange keys to establish a secure session.

• Secure **Communication**: Once the secure connection is established, data transmitted between the server and the client is encrypted, ensuring privacy and integrity.

The below is the code used to include https in the project:



server.js authserver.js request.rest .env

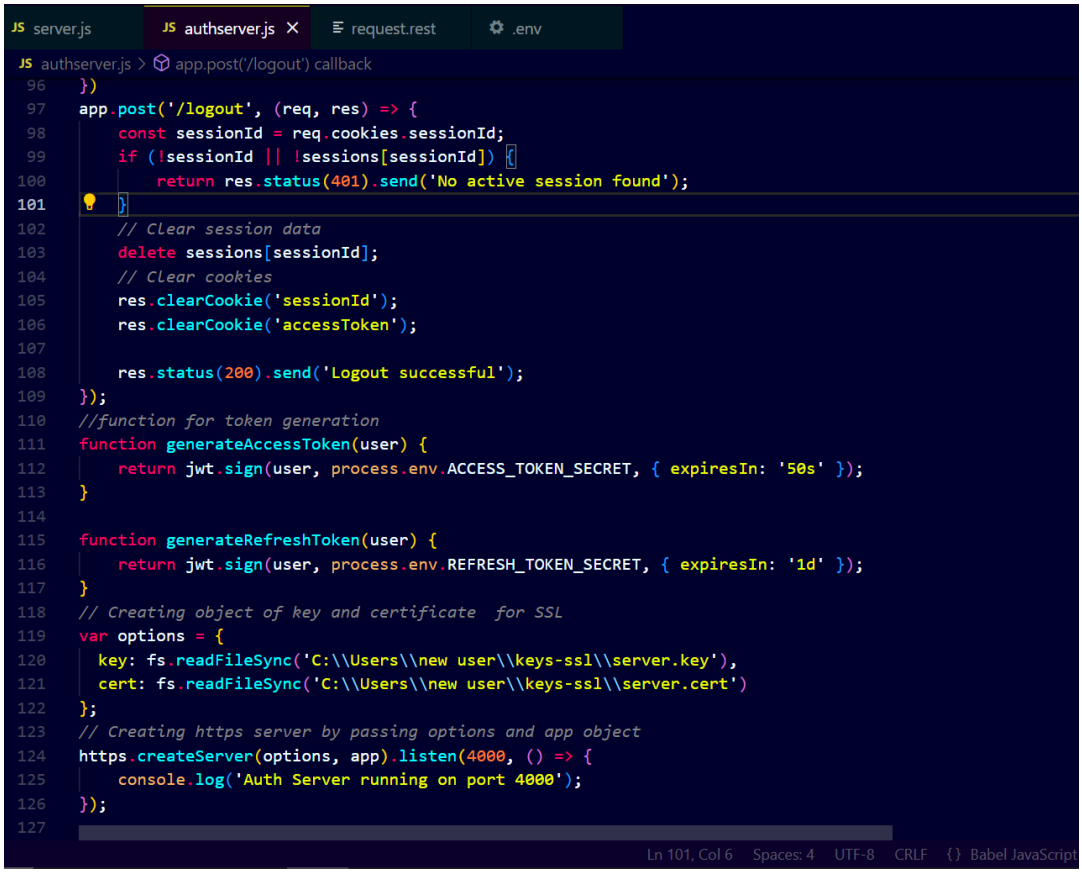
server.js > authenticateToken

```
require('dotenv').config();
const express = require('express');
const jwt = require('jsonwebtoken');
const cookieParser = require('cookie-parser');
const https = require('https');
const fs = require('fs');
const app = express();
app.use(express.json());
app.use(cookieParser());
const posts = [
    { username: 'abcd', title: 'post1' },
    { username: 'efgh', title: 'post2' }
];
app.get('/',(req,res)=>{
    res.send("Hello World!!")
})
app.get('/posts', authenticateToken, (req, res) => {
    res.json(posts.filter(post => post.username === req.user.name));
});

function authenticateToken(req, res, next) {
    const token = req.cookies.accessToken;
    if (token == null) return res.sendStatus(401);

    jwt.verify(token, process.env.ACCESS_TOKEN_SECRET, (err, user) => {
        if (err) return res.sendStatus(403);
        req.user = user;
        next();
    });
}
https.createServer(app).listen(3000, () => {
    console.log('Server running on port 3000');
```

authserver.js > app.post('/logout') callback

```
require('dotenv').config();
const express = require('express');
const jwt = require('jsonwebtoken');
const cookieParser = require('cookie-parser');
const crypto = require('crypto');
const https = require('https');
const fs = require('fs');

const app = express();
app.use(express.json());
app.use(cookieParser());

let sessions = {};

// Login route
app.post('/login', (req, res) => {
    const username = req.body.username;
    const user = { name: username };

    const accessToken = generateAccessToken(user);
    const sessionId = crypto.randomBytes(16).toString('hex'); // Generate a random session ID

    sessions[sessionId] = { username, refreshToken: generateRefreshToken(user) };
    res.cookie('sessionId', sessionId, { httpOnly: true, secure: true }); // Set session ID as a secure an
    res.cookie('accessToken', accessToken, { httpOnly: true, secure: true }); // Set access token as a sec
    res.json({ accessToken: accessToken });
});

// Token route
app.post('/token', (req, res) => {
    const sessionId = req.cookies.sessionId; // Retrieves session ID from cookies
    if (!sessionId || !sessions[sessionId]) return res.sendStatus(401);// Checks if session ID is valid
```


authserver.js > app.post('/logout') callback
C:\Users\new user\Desktop\jwt tokens\authserver.js
{
const { refreshToken, username } = sessions[sessionId];// Retrieves refresh token and username from se
// Verify refresh token
jwt.verify(refreshToken, process.env.REFRESH_TOKEN_SECRET, (err, user) => {
if (err) return res.sendStatus(403); //refresh token is invalid
const accessToken = generateAccessToken({ name: user.name });// Generate new access token
res.cookie('accessToken', accessToken, { httpOnly: true, secure: true }); // token converted to se
res.json({ accessToken: accessToken });
});
});
// Logout route
app.get('/helloworld',(req,res)=>{
res.json({message:"Https is working!!"})
})
app.post('/logout', (req, res) => {
const sessionId = req.cookies.sessionId;
if (!sessionId || !sessions[sessionId]) {
return res.status(401).send('No active session found');
}
// Clear session data
delete sessions[sessionId];
// Clear cookies
res.clearCookie('sessionId');
res.clearCookie('accessToken');
res.status(200).send('Logout successful');
});
//function for token generation
function generateAccessToken(user) {
return jwt.sign(user, process.env.ACCESS_TOKEN_SECRET, { expiresIn: '50s' });
Ln 101, Col 6 Spaces: 4 UTF-8 CRLF {} Babel JavaScript Go Liv


server.js
authserver.js
request.rest
.env
authserver.js > app.post('/logout') callback
})
app.post('/logout', (req, res) => {
const sessionId = req.cookies.sessionId;
if (!sessionId || !sessions[sessionId]) {
return res.status(401).send('No active session found');
}
// Clear session data
delete sessions[sessionId];
// Clear cookies
res.clearCookie('sessionId');
res.clearCookie('accessToken');
res.status(200).send('Logout successful');
});
//function for token generation
function generateAccessToken(user) {
return jwt.sign(user, process.env.ACCESS_TOKEN_SECRET, { expiresIn: '50s' });
}
function generateRefreshToken(user) {
return jwt.sign(user, process.env.REFRESH_TOKEN_SECRET, { expiresIn: '1d' });
}
// Creating object of key and certificate for SSL
var options = {
key: fs.readFileSync('C:\\Users\\new user\\keys-ssl\\server.key'),
cert: fs.readFileSync('C:\\Users\\new user\\keys-ssl\\server.cert')
};
// Creating https server by passing options and app object
https.createServer(options, app).listen(4000, () => {
console.log('Auth Server running on port 4000');
});
Ln 101, Col 6 Spaces: 4 UTF-8 CRLF {} Babel JavaScript

CONCLUSION

In the conclusion, my experience during the summer training at ONGC has been incredibly rewarding and enlightening. Throughout the duration of the training, I had the opportunity to work on various projects and gain hand on experience in developing innovative solutions.

The exposure to real-world scenarios and the guidance of experienced professionals like Ms. Kanika, Mr. Paul Mickey Toppo and Mr. Prem Prakash Hansda has been invaluable. Their expertise and mentorship have played a significant role in shaping my skills and understanding of the industry.

One of the most fulfilling aspects of the training was the opportunity to contribute to the development of the backend designing using different technologies.

There are numerous areas for future improvement in this venture, including:

1. User Interface (UI) Enhancements: By taking continuous feedback from the user the project will have a vast space of improvement.
2. Integration with Additional Systems: The project can be integrated with other relevant systems or technologies to enhance its functionality and extend its capabilities.