FINAL PROJECT OF COMPUTER SYSTEM SECURITY

Università degli Studi di Messina

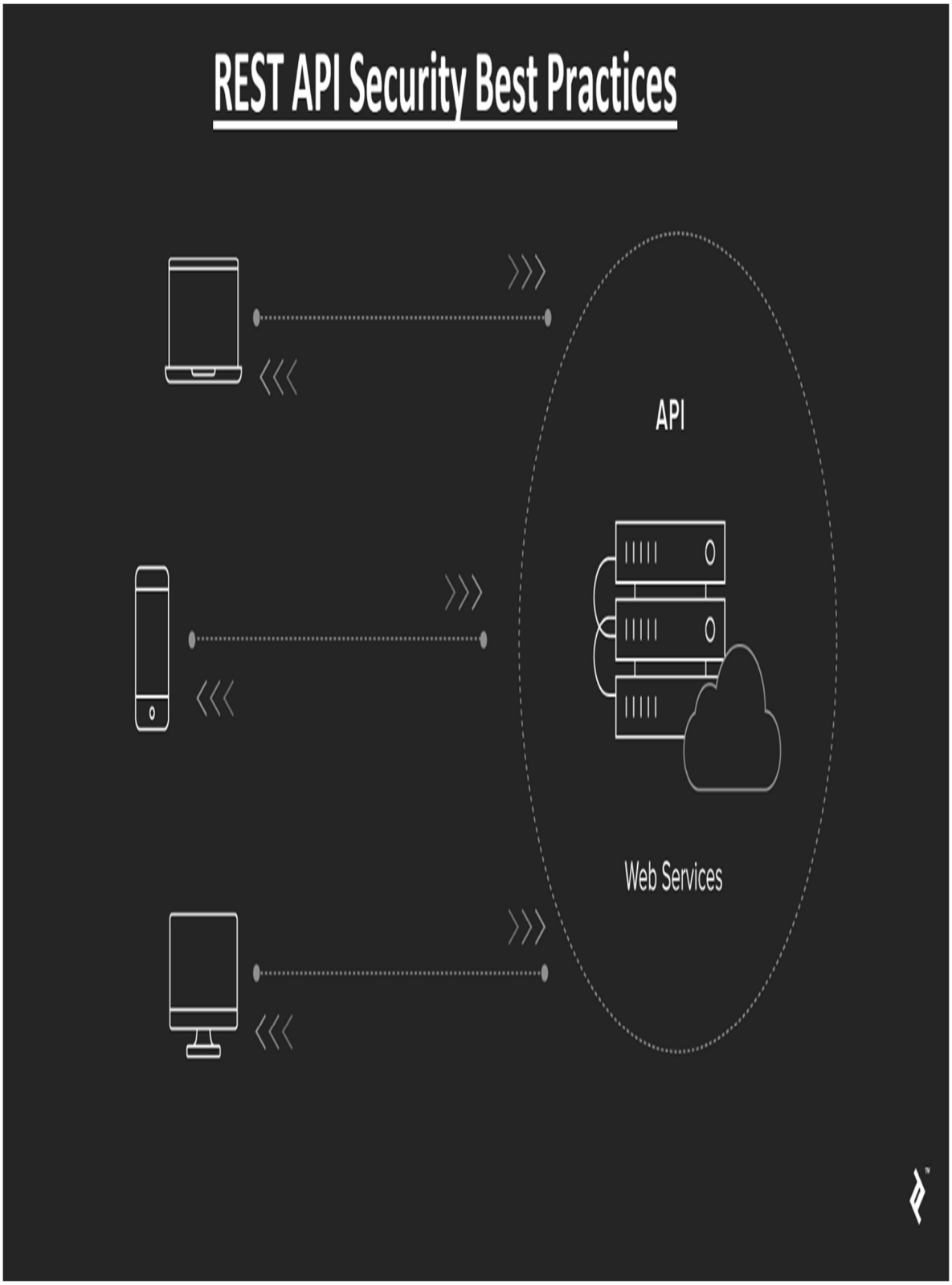
Submitted by: Kulwant Singh Rathore (555368)

Amita Rajak (545238)

Guided by: Prof. Massimo Villari

Course: Engineering and Computer Science

SecureNet API Shield: Unified DNS and JWT Security



CONTENTS

ABSTRACT

This report outlines a comprehensive guide for developing, deploying, and securing a server-side application on Ubuntu 20.04, integrating Secure DNS (DNSSEC) and JWT (JSON Web Token) authentication. The integration of these technologies aims to enhance security by:

Secure DNS (DNSSEC): Protects against DNS attacks like spoofing and cache poisoning by ensuring the authenticity of DNS data using cryptographic signatures.

JWT Authentication: Manages user authentication and authorization by generating secure tokens that verify user identities and control access to API endpoints.

The project combines these security measures to provide dual-layer protection, ensuring data integrity and user trust. It covers server setup, dependency management, API testing, database creation, UI development, deployment, and performance evaluation, offering a robust security framework for modern web services.

OVERVIEW

This project aims to create a robust, secure, and scalable server-side application by integrating Secure DNS (DNSSEC) and JWT (JSON Web Token) authentication on an Ubuntu 20.04 server. The following sections provide a detailed overview of each component and its role in the project:

Server Setup

Configuring Ubuntu 20.04 as the hosting environment is the first step:

Installation: Install the latest version of Ubuntu 20.04 on a server or virtual machine.

System Updates: Regularly update the system to ensure it has the latest security patches and features.

Software Packages: Install essential software packages such as Node.js, npm (Node Package Manager), MongoDB, and other necessary tools.

Dependency Management

Managing dependencies is crucial for project stability and maintainability:

Node.js and npm: Use npm to install and manage Node.js packages required for the application.

Express.js: A lightweight web application framework for Node.js to handle HTTP requests and routing.

Mongoose: An Object Data Modeling (ODM) library for MongoDB, providing schema-based solutions to model application data.

JWT Libraries: Libraries such as jsonwebtoken to handle the creation and verification of JWT tokens. DNSSEC and TLS Libraries: Tools to implement and manage Secure DNS and encrypted traffic.

API Testing

Using Postman for API testing ensures the functionality and security of the API endpoints:

Create Collections - Organize API endpoints into collections for structured testing.

Send Requests - Simulate various HTTP requests (GET, POST, PUT, DELETE) to the API.

Validate Responses - Check responses for correct data, status codes, and error handling.

JWT Validation - Ensure that JWT authentication is properly implemented and that tokens are correctly generated and validated.

Authentication

Implementing JWT for secure user authentication and authorization:

Token Generation - Create JWT tokens upon user registration and login, embedding user information and permissions.

Token Validation - Verify JWT tokens on each request to ensure that only authenticated users can access protected resources.

Role-Based Access Control - Assign roles to users (e.g., ADMIN, USER) and control access to resources based on these roles.

Database Creation

Setting up and managing MongoDB for data storage:

MongoDB Installation - Install MongoDB on the server.

Mongoose Integration - Use Mongoose to define schemas and models for the application data.

CRUD Operations - Implement Create, Read, Update, and Delete operations to manage data.

Data Security - Ensure that all data transactions are secure and only accessible through authenticated requests.

UI Development

Building a responsive user interface using HTML, CSS, and React:

HTML - Structure the web pages using HTML.

CSS - Style the web pages with CSS to ensure they are visually appealing and responsive.

React - Use React to build dynamic and interactive user interfaces. Components will be created to handle different parts of the UI, ensuring reusability and efficient rendering.

API Integration - Use Axios or the Fetch API to connect the UI with the backend services, ensuring secure data exchange.

Deployment and Hosting

Deploying the application to cloud platforms like AWS or Heroku:

Prepare Application: Bundle and prepare the application for deployment.

Choose Hosting Platform: Select a cloud hosting provider (AWS, Heroku) based on scalability and cost considerations.

Set Up Hosting Environment: Configure the server environment, ensuring support for Secure DNS and JWT authentication.

Continuous Deployment: Implement CI/CD pipelines to automate deployment and updates.

Security

Implementing DNSSEC and JWT, and configuring firewalls and secure access protocols:

DNSSEC Implementation: Configure DNSSEC to ensure that DNS data is authenticated and has not been tampered with.

TLS Encryption: Use TLS to encrypt traffic between the client and server, ensuring data confidentiality and integrity.

JWT Authentication: Implement JWT to secure API endpoints, ensuring that only authenticated and authorized users can access resources.

Firewall Configuration: Use Uncomplicated Firewall (UFW) to allow only necessary traffic (e.g., ports 443 for HTTPS and 22 for SSH) and implement rate limiting to protect against brute force attacks.

SSH Access: Configure SSH for secure remote access to the server, using key-based authentication for added security.

Integration Benefits

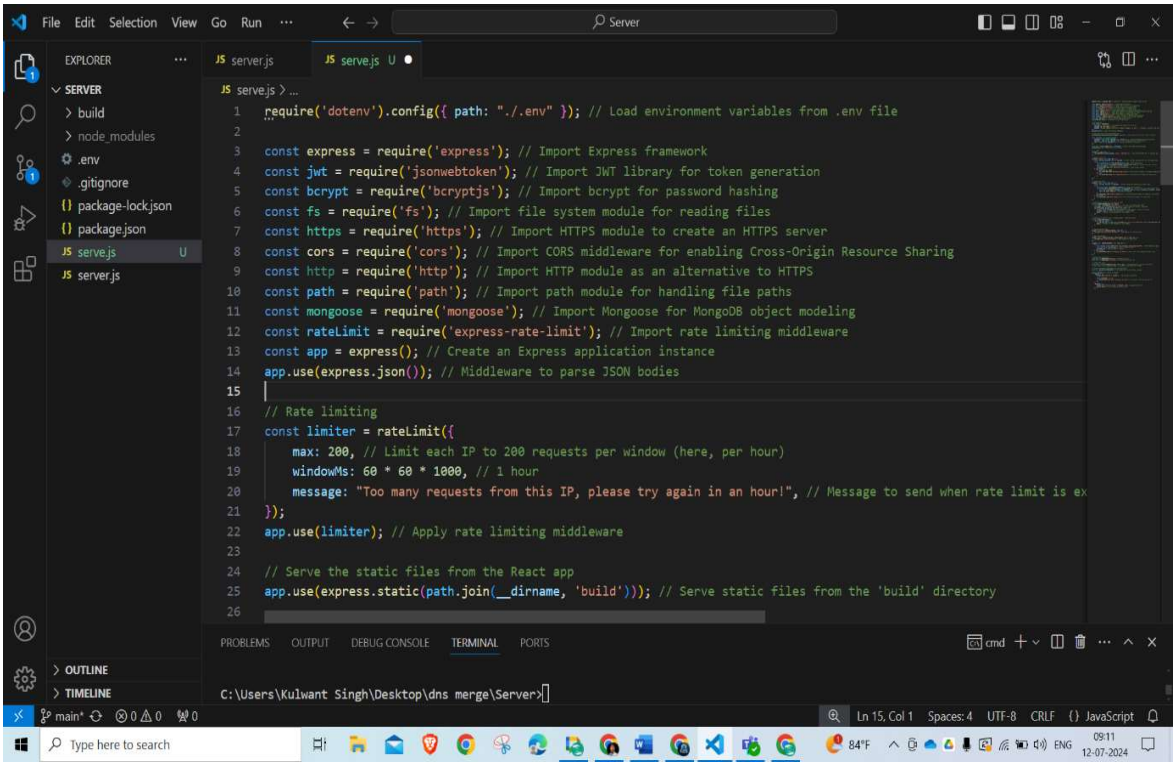
By integrating Secure DNS and JWT, the application benefits from enhanced security at both the network and application levels. DNSSEC protects against DNS attacks, ensuring the integrity and authenticity of DNS data, while JWT provides a robust mechanism for user authentication and authorization, controlling access to API endpoints. This dual-layer protection significantly reduces the risk of cyber threats, ensuring a secure and reliable application.

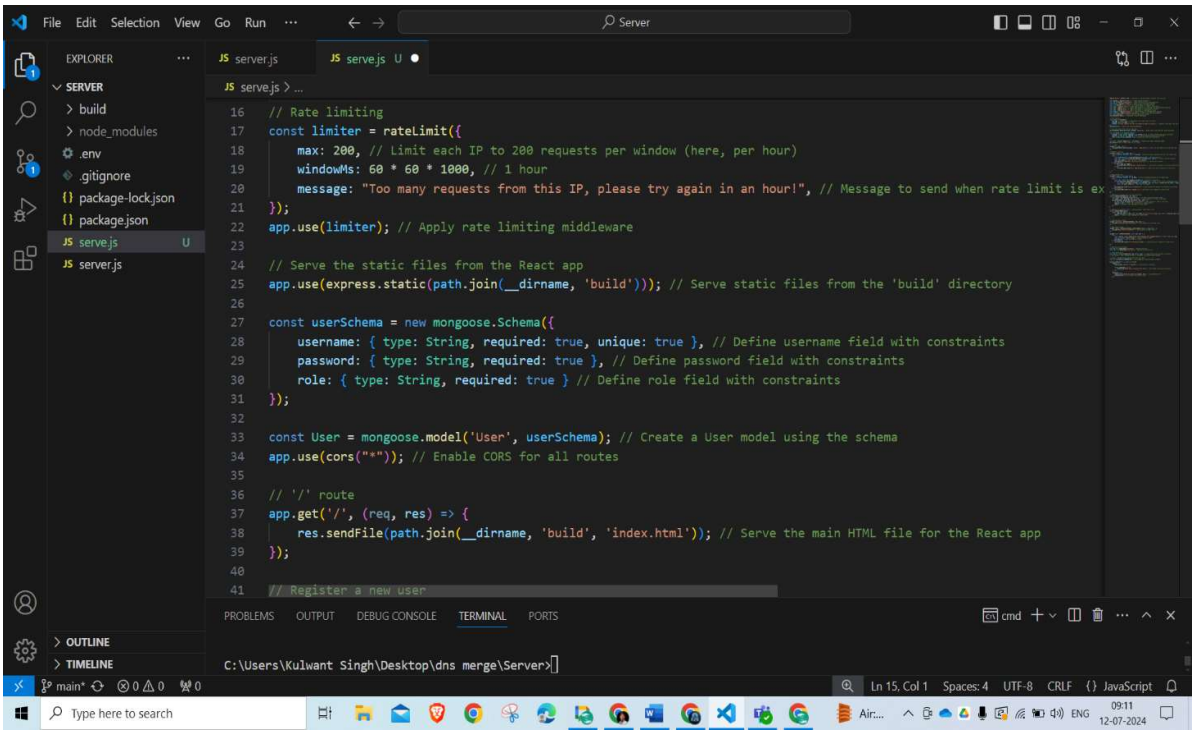
This comprehensive approach not only addresses current security needs but also lays a strong foundation for future enhancements and scalability.

BACKEND/SERVER-SIDE DEVELOPMENT AND INTREGRETION

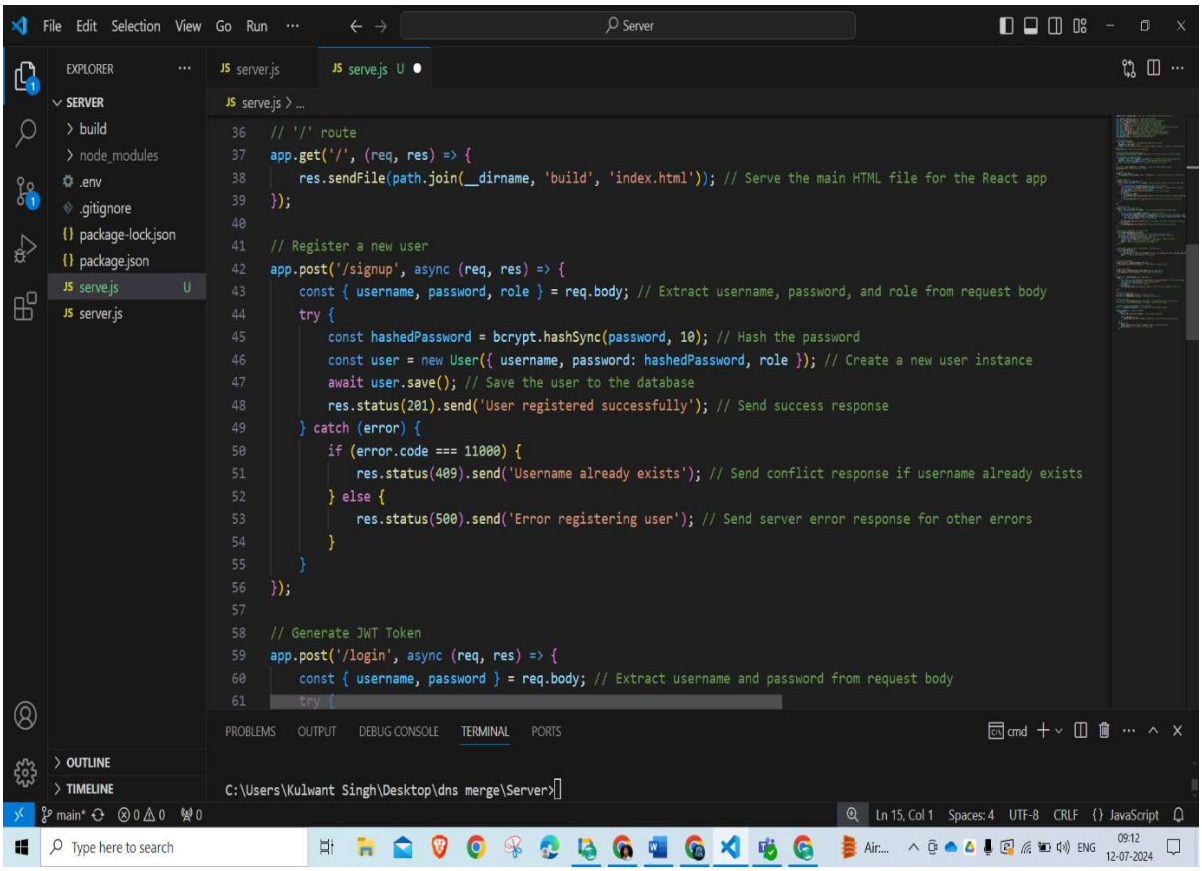
- **Backend:** The backend of a web application refers to the server-side components that manage the application's logic, database interactions, and server configuration. It is responsible for processing requests from the frontend, performing operations like data retrieval, storage, and manipulation, and sending responses back to the client. In this project, the backend is built using Node.js with Express, providing endpoints for user registration, authentication, and role-based access control. It employs JWT for secure token-based authentication and MongoDB for persistent data storage, ensuring secure and efficient handling of user data and application logic.
- **NODE JS :** Node.js is a runtime environment that allows developers to execute JavaScript code server-side. Built on Chrome's V8 JavaScript engine, it enables the creation of scalable and high-performance applications, especially web servers and networked applications. Node.js uses an event-driven, non-blocking I/O model, which makes it efficient and suitable for real-time applications and services that require a large number of concurrent connections.
- **Express :** Express is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications. It simplifies the process of developing server-side applications by offering built-in functionalities for handling HTTP requests, middleware integration, routing, and more. Express allows developers to create APIs and web applications quickly, with an emphasis on performance and ease of use.

SERVER.JS

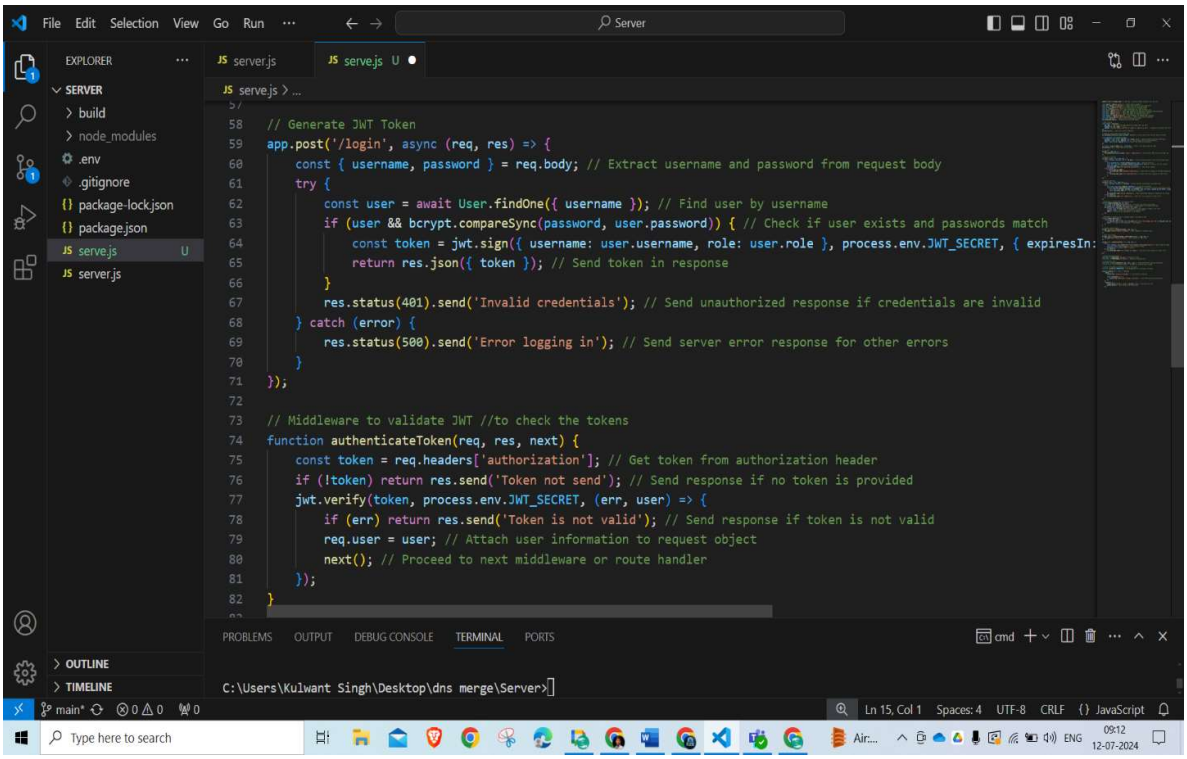


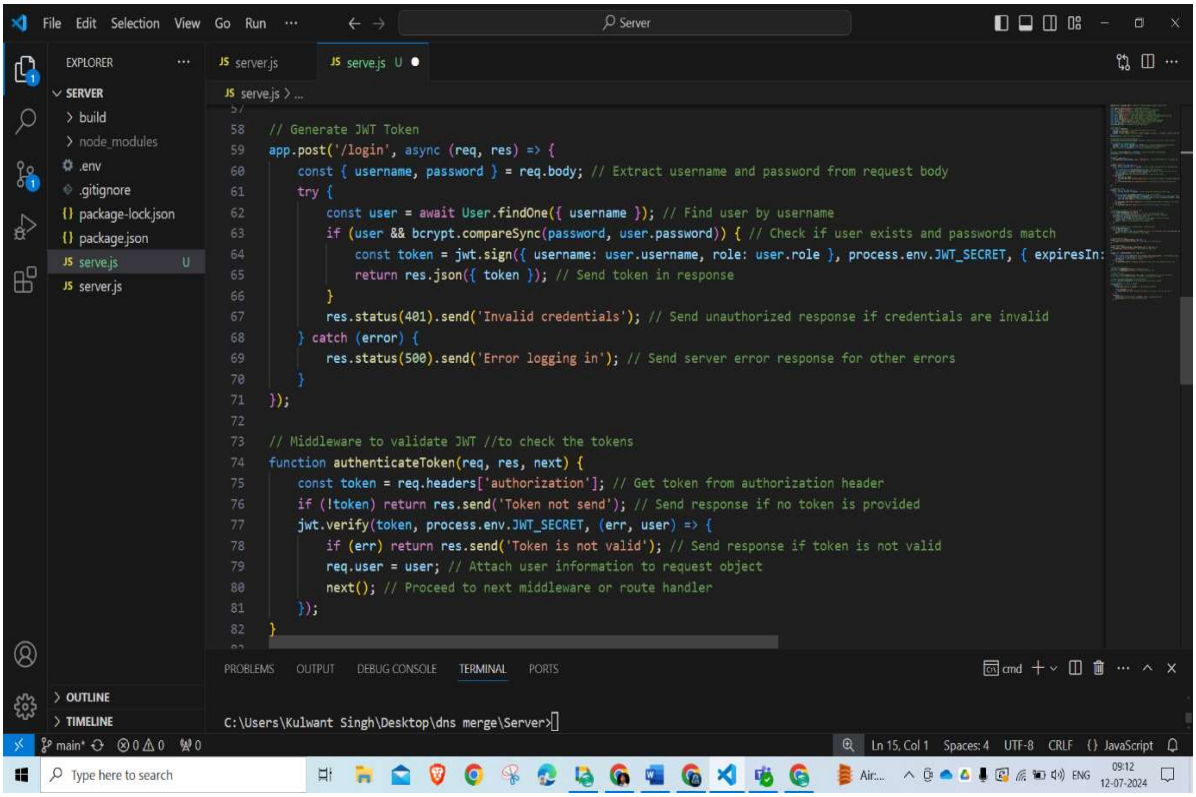


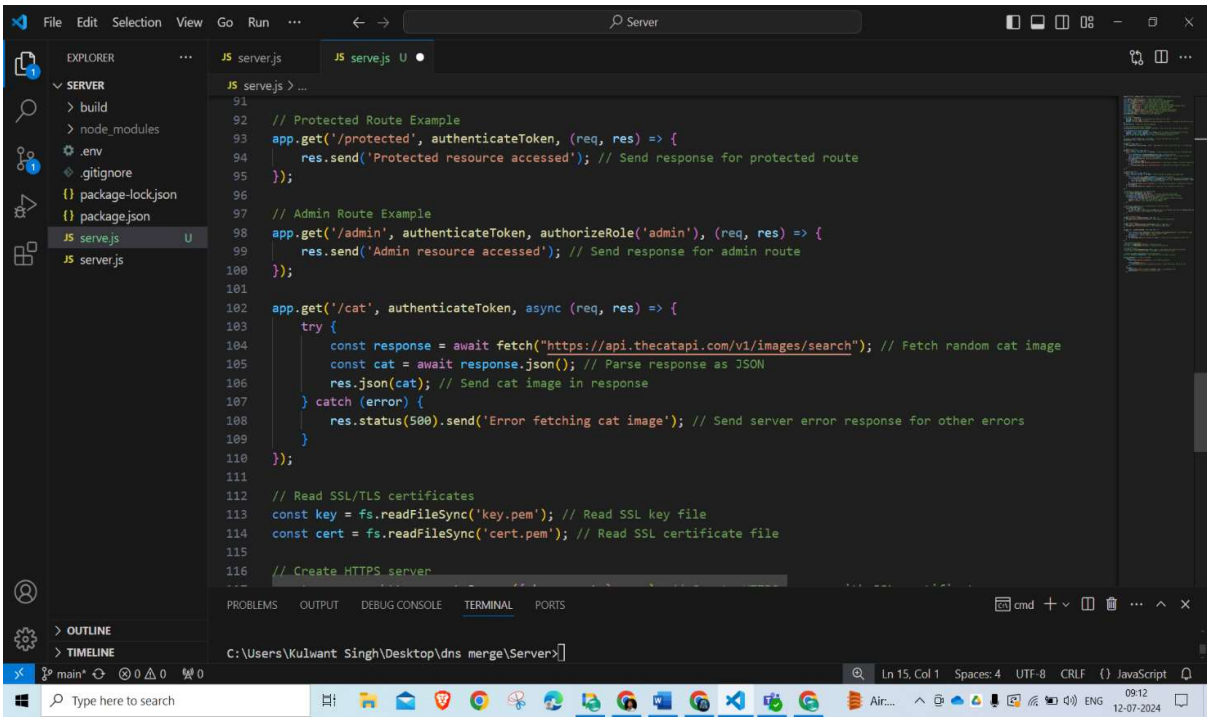
```
// Rate limiting
const limiter = rateLimit({
    max: 200, // Limit each IP to 200 requests per window (here, per hour)
    windowMs: 60 * 60 * 1000, // 1 hour
    message: "Too many requests from this IP, please try again in an hour!", // Message to send when rate limit is ex
});
app.use(limiter); // Apply rate limiting middleware

// Serve the static files from the React app
app.use(express.static(path.join(__dirname, 'build'))); // Serve static files from the 'build' directory

const userSchema = new mongoose.Schema({
    username: { type: String, required: true, unique: true }, // Define username field with constraints
    password: { type: String, required: true }, // Define password field with constraints
    role: { type: String, required: true } // Define role field with constraints
});

const User = mongoose.model('User', userSchema); // Create a User model using the schema
app.use(cors("*")); // Enable CORS for all routes

// '/' route
app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'build', 'index.html')); // Serve the main HTML file for the React app
});

// Register a new user
```



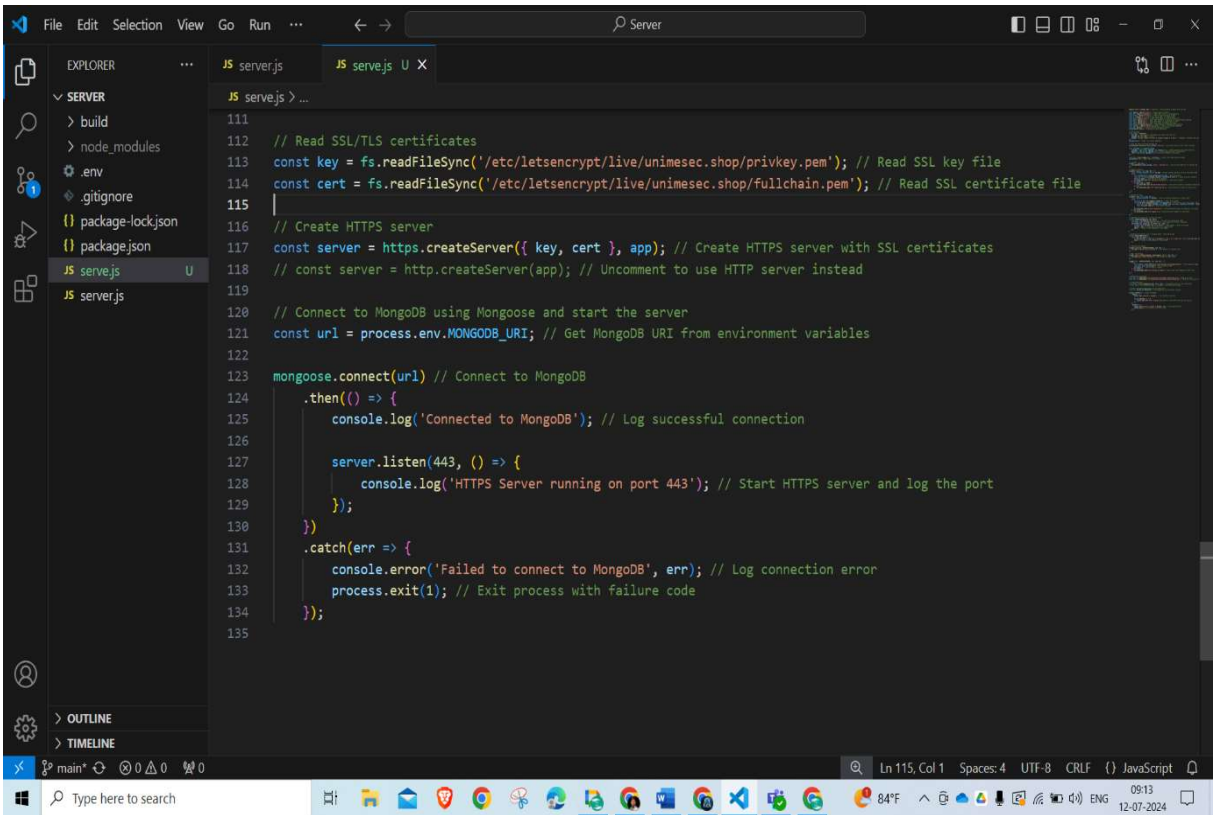
```
// '/' route
app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'build', 'index.html')); // Serve the main HTML file for the React app
});

// Register a new user
app.post('/signup', async (req, res) => {
    const { username, password, role } = req.body; // Extract username, password, and role from request body
    try {
        const hashedPassword = bcrypt.hashSync(password, 10); // Hash the password
        const user = new User({ username, password: hashedPassword, role }); // Create a new user instance
        await user.save(); // Save the user to the database
        res.status(201).send('User registered successfully'); // Send success response
    } catch (error) {
        if (error.code === 11000) {
            res.status(409).send('Username already exists'); // Send conflict response if username already exists
        } else {
            res.status(500).send('Error registering user'); // Send server error response for other errors
        }
    }
});

// Generate JWT Token
app.post('/login', async (req, res) => {
    const { username, password } = req.body; // Extract username and password from request body
    try {
```


File Edit Selection View Go Run
Server
EXPLORER
SERVER
build
node_modules
.env
.gitignore
package-lock.json
package.json
serve.js U
server.js
server.js
serve.js U
serve.js > ...
58 // Generate JWT Token
59 app.post('/login', async (req, res) => {
60 const { username, password } = req.body; // Extract username and password from request body
61 try {
62 const user = await User.findOne({ username }); // Find user by username
63 if (user && bcrypt.compareSync(password, user.password)) { // Check if user exists and passwords match
64 const token = jwt.sign({ username: user.username, role: user.role }, process.env.JWT_SECRET, { expiresIn:
65 return res.json({ token }); // Send token in response
66 }
67 res.status(401).send('Invalid credentials'); // Send unauthorized response if credentials are invalid
68 } catch (error) {
69 res.status(500).send('Error logging in'); // Send server error response for other errors
70 }
71 });
72
73 // Middleware to validate JWT //to check the tokens
74 function authenticateToken(req, res, next) {
75 const token = req.headers['authorization']; // Get token from authorization header
76 if (!token) return res.send('Token not send'); // Send response if no token is provided
77 jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
78 if (err) return res.send('Token is not valid'); // Send response if token is not valid
79 req.user = user; // Attach user information to request object
80 next(); // Proceed to next middleware or route handler
81 });
82 }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
cmd
C:\Users\Kulwant Singh\Desktop\dns merge\Server>
OUTLINE
TIMELINE
main*
Ln 15, Col 1 Spaces: 4 UTF-8 CRLF {} JavaScript
Type here to search
ENG
09:12
12-07-2024


File Edit Selection View Go Run
Server
EXPLORER
SERVER
build
node_modules
.env
.gitignore
package-lock.json
package.json
serve.js U
server.js
server.js
serve.js U
serve.js > ...
58 // Generate JWT Token
59 app.post('/login', async (req, res) => {
60 const { username, password } = req.body; // Extract username and password from request body
61 try {
62 const user = await User.findOne({ username }); // Find user by username
63 if (user && bcrypt.compareSync(password, user.password)) { // Check if user exists and passwords match
64 const token = jwt.sign({ username: user.username, role: user.role }, process.env.JWT_SECRET, { expiresIn:
65 return res.json({ token }); // Send token in response
66 }
67 res.status(401).send('Invalid credentials'); // Send unauthorized response if credentials are invalid
68 } catch (error) {
69 res.status(500).send('Error logging in'); // Send server error response for other errors
70 }
71 });
72
73 // Middleware to validate JWT //to check the tokens
74 function authenticateToken(req, res, next) {
75 const token = req.headers['authorization']; // Get token from authorization header
76 if (!token) return res.send('Token not send'); // Send response if no token is provided
77 jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
78 if (err) return res.send('Token is not valid'); // Send response if token is not valid
79 req.user = user; // Attach user information to request object
80 next(); // Proceed to next middleware or route handler
81 });
82 }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
cmd
C:\Users\Kulwant Singh\Desktop\dns merge\Server>
OUTLINE
TIMELINE
main*
Ln 15, Col 1 Spaces: 4 UTF-8 CRLF {} JavaScript
Type here to search
ENG
09:12
12-07-2024



```
// Protected Route Example
app.get('/protected', authenticateToken, (req, res) => {
    res.send('Protected resource accessed'); // Send response for protected route
});

// Admin Route Example
app.get('/admin', authenticateToken, authorizeRole('admin'), (req, res) => {
    res.send('Admin resource accessed'); // Send response for admin route
});

app.get('/cat', authenticateToken, async (req, res) => {
    try {
        const response = await fetch("https://api.thecatapi.com/v1/images/search"); // Fetch random cat image
        const cat = await response.json(); // Parse response as JSON
        res.json(cat); // Send cat image in response
    } catch (error) {
        res.status(500).send('Error fetching cat image'); // Send server error response for other errors
    }
});

// Read SSL/TLS certificates
const key = fs.readFileSync('key.pem'); // Read SSL key file
const cert = fs.readFileSync('cert.pem'); // Read SSL certificate file

// Create HTTPS server
```



```
// Read SSL/TLS certificates
const key = fs.readFileSync('/etc/letsencrypt/live/unimesec.shop/privkey.pem'); // Read SSL key file
const cert = fs.readFileSync('/etc/letsencrypt/live/unimesec.shop/fullchain.pem'); // Read SSL certificate file

// Create HTTPS server
const server = https.createServer({ key, cert }, app); // Create HTTPS server with SSL certificates
// const server = http.createServer(app); // Uncomment to use HTTP server instead

// Connect to MongoDB using Mongoose and start the server
const url = process.env.MONGODB_URI; // Get MongoDB URI from environment variables

mongoose.connect(url) // Connect to MongoDB
    .then(() => {
        console.log('Connected to MongoDB'); // Log successful connection

        server.listen(443, () => {
            console.log('HTTPS Server running on port 443'); // Start HTTPS server and log the port
        });
    })
    .catch(err => {
        console.error('Failed to connect to MongoDB', err); // Log connection error
        process.exit(1); // Exit process with failure code
    });
```

ENV:

JWT_SECRET = SHYAMU
MONGODB_URI =
mongodb+srv://amita:dpDbI0HzylvFPQPM@secure.fwadhjg.mongodb.net/?retryWrites=true&w=majority&appName=secure

DEPENDENCIES:

1.npm install cors: Cross-Origin Resource Sharing (CORS) is used in web development to allow web applications running at one origin (domain) to access resources from another origin.

2.npm install mongoose: Mongoose is a popular Node.js library that provides a schematic based solution for modeling application data and interacting with MongoDB databases.

3.npm install express: Express.js is a popular web framework for Node.js that simplifies the process of building web applications and APIs.

4.npm install bcrypt: Password Hashing bcrypt is a library used to hash passwords securely. Hashing is a one-way process that converts plain text (like a password) into a fixed-size string of characters, making it computationally difficult to reverse the process and obtain the original password. This enhances security by protecting user passwords even if the database is compromised.

5 npm install express-rate-limit: It is a middleware for Express.js that helps us limit repeated requests to out server from the same client or IP address. This is useful for preventing abuse, such as brute-force attacks on authentication endpoints, spamming of requests, or simply to manage server resources more efficiently.

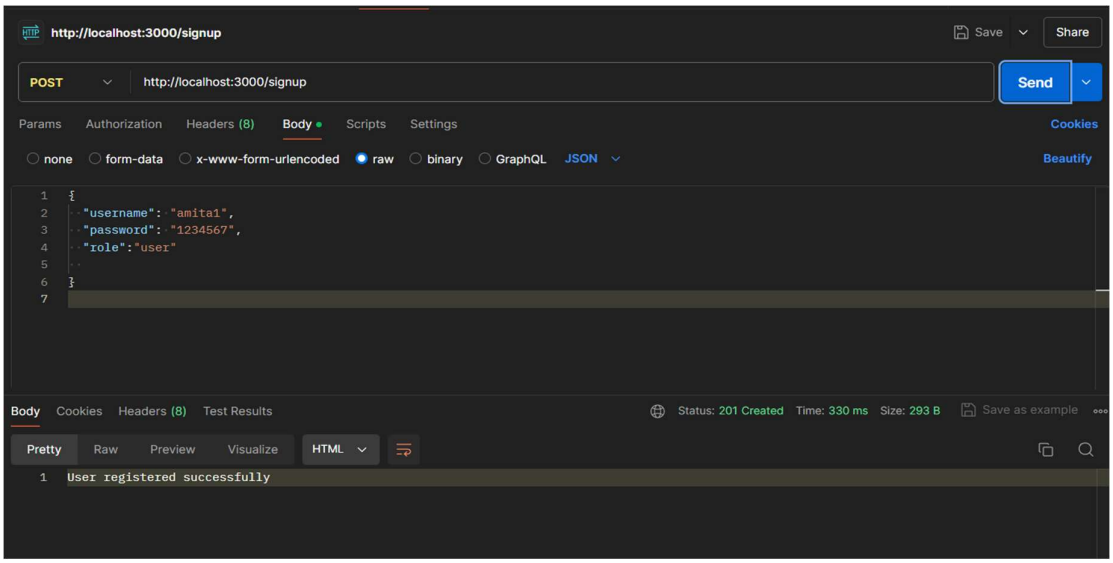
6 npm install axios: It is a promise-based HTTP client for the browser and Node.js. It allows us to make HTTP requests to fetch or save data to a web server. It is widely used for its simplicity and ease of use, supporting all modern browsers and Node.js.

POSTMAN:

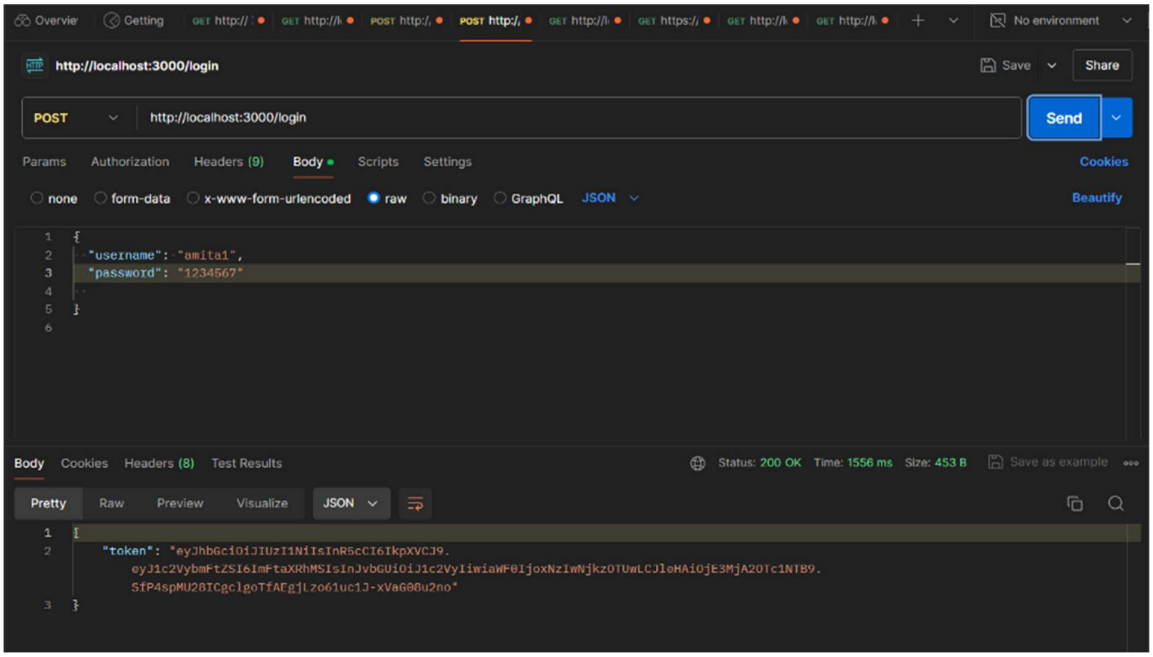
Postman is a widely used API development tool that simplifies designing, testing, documenting, and monitoring APIs. With its user-friendly interface, it allows developers to easily send and analyse HTTP requests. Postman supports automated testing through scripting, enables environment management for different stages of development, and facilitates team collaboration by sharing collections and documentation. It also features mock servers to simulate API responses and monitoring tools to ensure API performance and reliability. Overall, Postman enhances the efficiency and quality of API development and testing.

TESTING OF THE ENDPOINTS

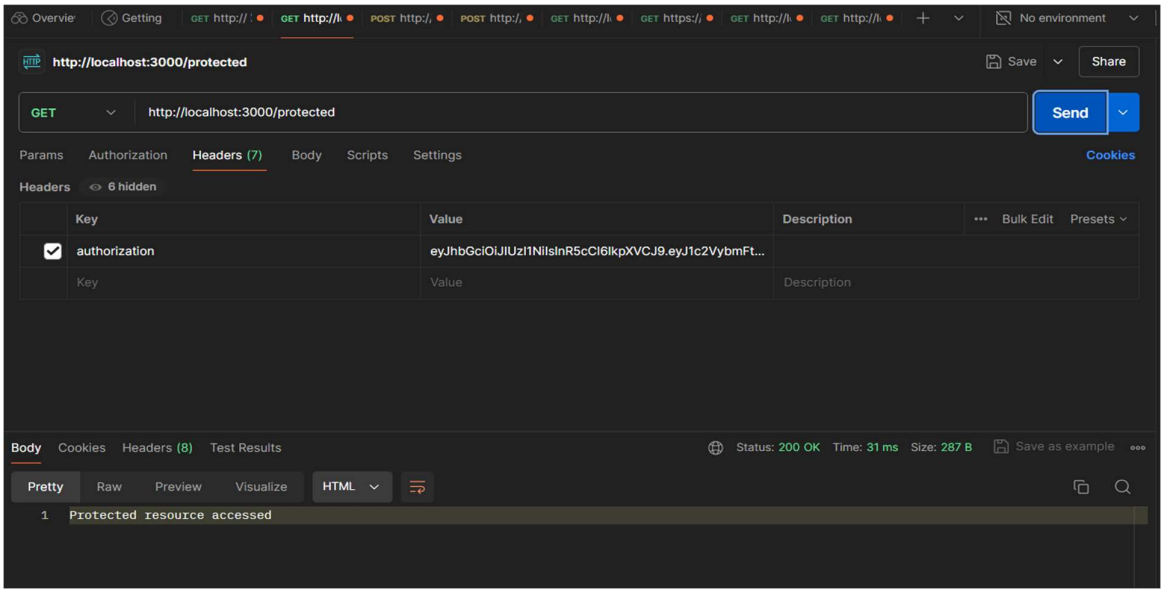
1./SIGNUP



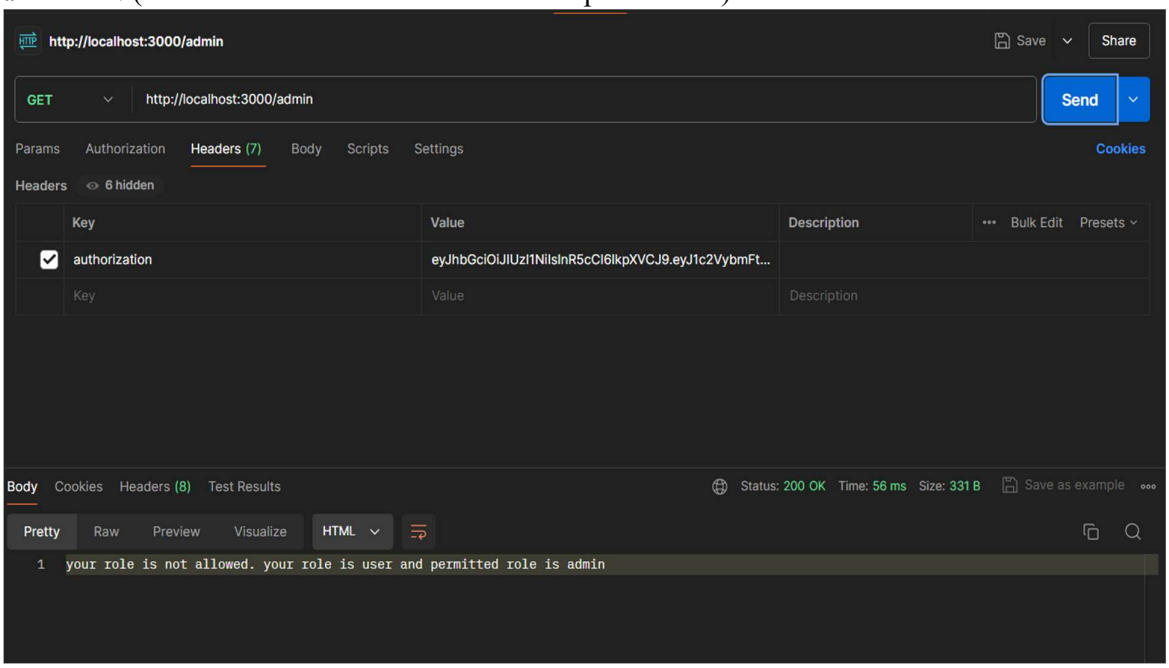
2. /LOGIN



3./PROTECTED



./**ADMIN** (Authentication and Authorization as per the role)



(JSON WEB TOKEN):

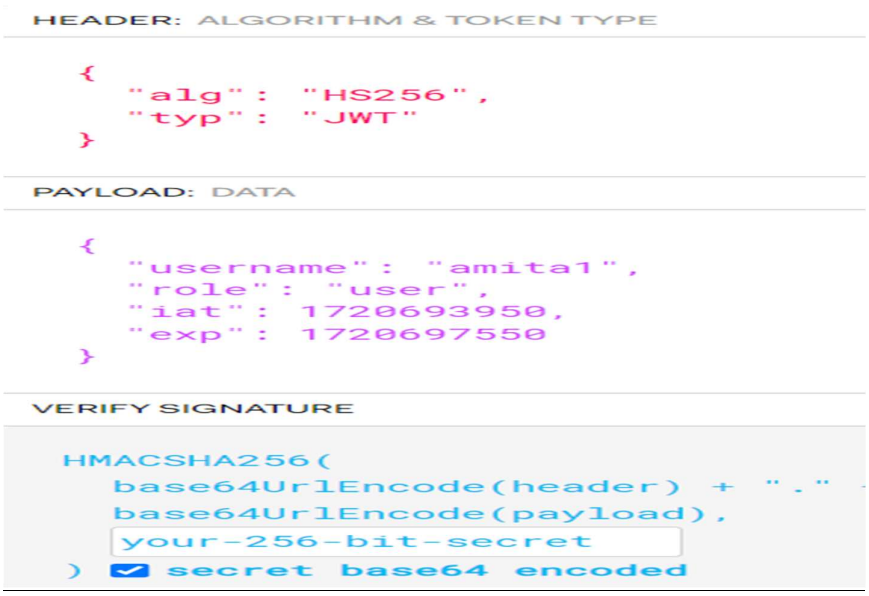
When we create JSON Web Tokens, they are signed. Signing the token allows its recipient to validate that the content of the token wasn't changed and verify the original issuer of the token created signature.
Heads up! Signatures are not encryptions!
Signing JWTs doesn't make their data unreadable. Signatures only verify that the content of the JWT was not changed.
The signature is the part of a JWT that verifies the content of the JWT hasn't changed since the moment it has been issued (as it would happen, for example, in bucket brigade attacks formerly known as "man or person in the middle" attacks).

ENCODED

"token":

"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFtaXRhMSIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzIwNjkzOTUwLCJleHAiOjE3MjA2OTc1NTB9.SfP4spMU28ICgclgoTfAEgjLzo61uc1J-xVaG08u2no"

DECODED



HS256 SIGNING ALGORITHM

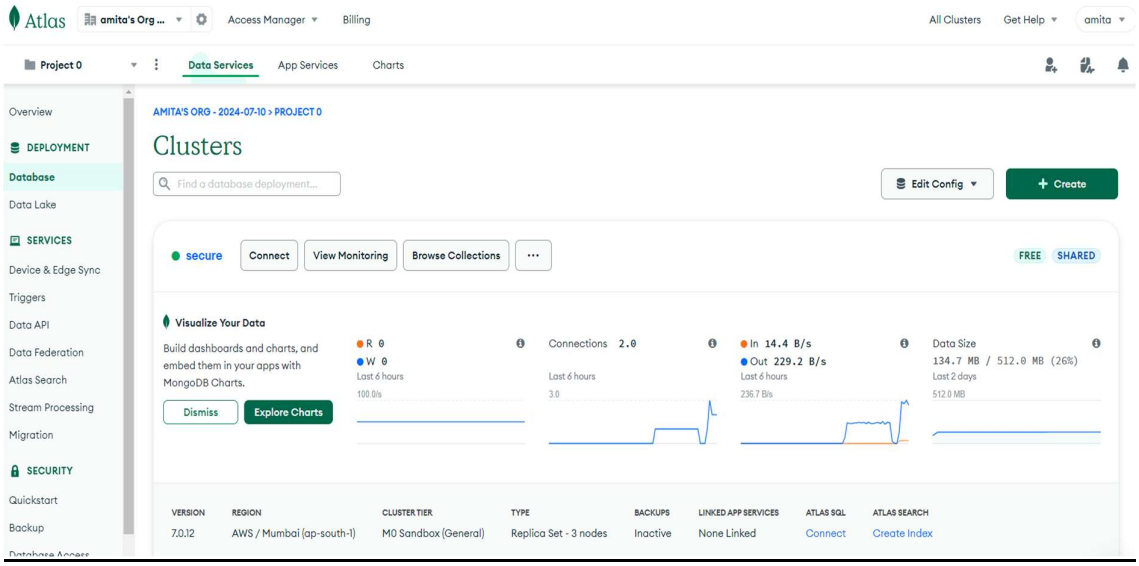
HS256 (HMAC with SHA-256) is a symmetric keyed hashing algorithm that uses one secret key. Symmetric means two parties share the secret key. The key is used for both generating the signature and validating it. Be mindful when using a shared key; it can open potential vulnerabilities if the verifiers (multiple applications) are not appropriately secured.

DATABASE CREATION

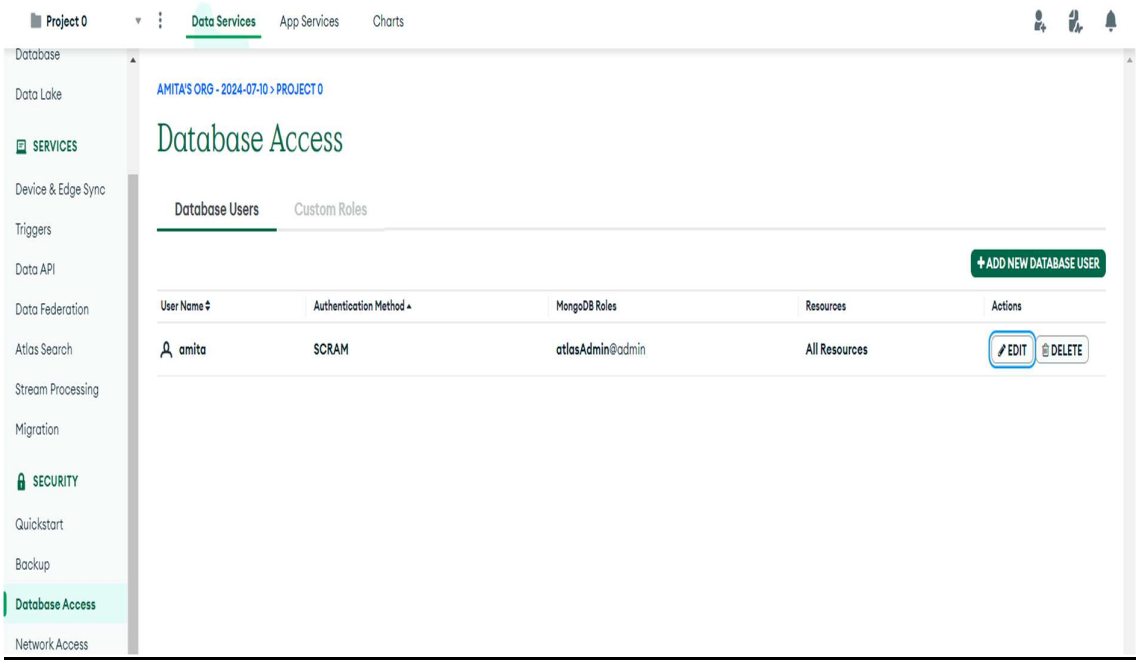
Mongodb Atlas is used for creating the database: It is a fully managed cloud database service provided by MongoDB. It is designed to handle deployment, management, and scalability of MongoDB databases with ease, offering a variety of features that help developers like us focus on building applications rather than managing infrastructure.

SETUP

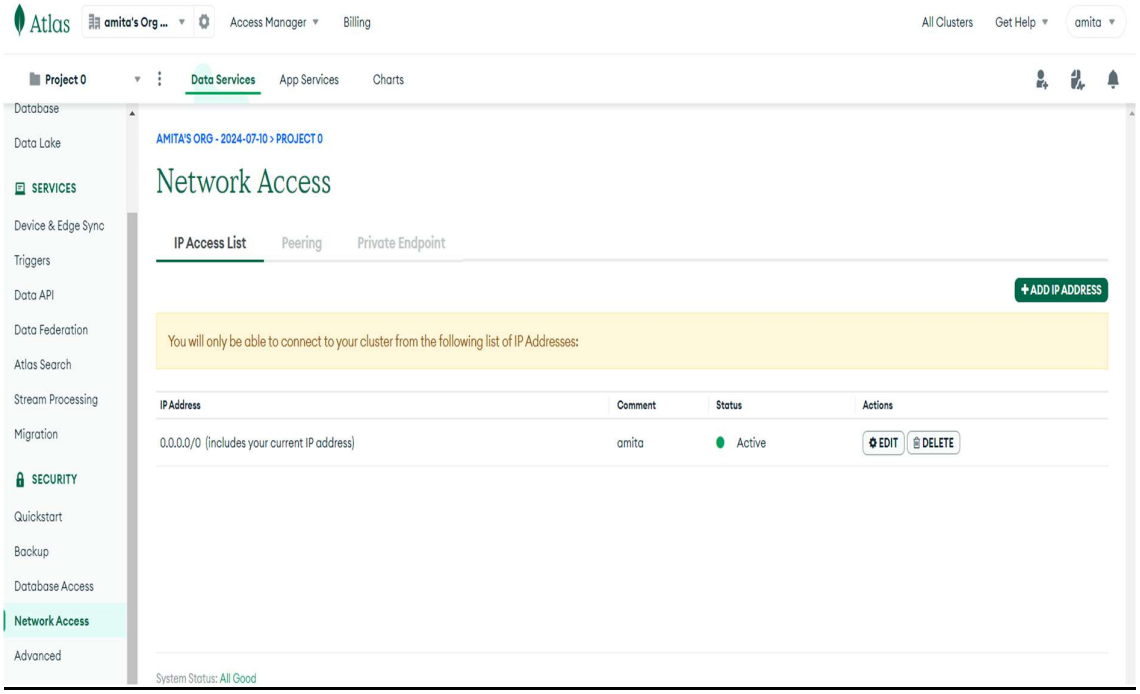
Database



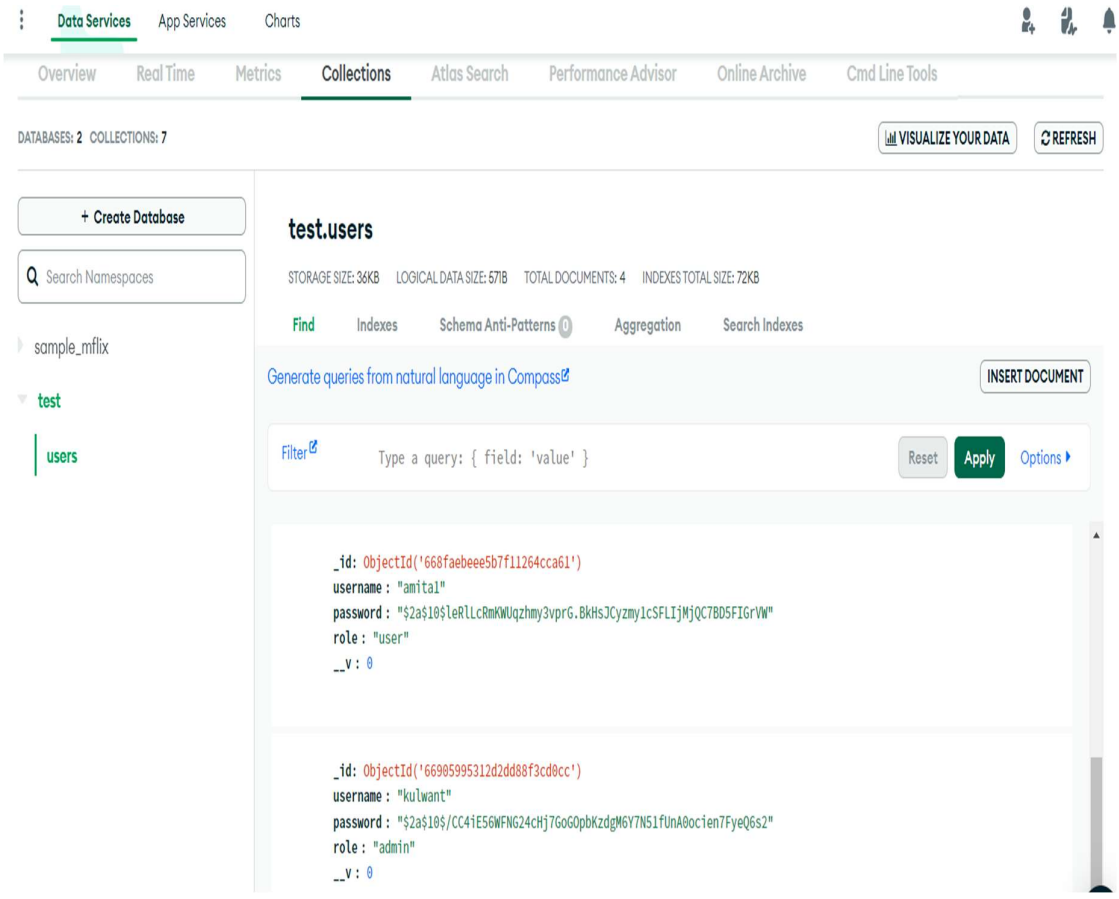
Database Access



Network Access



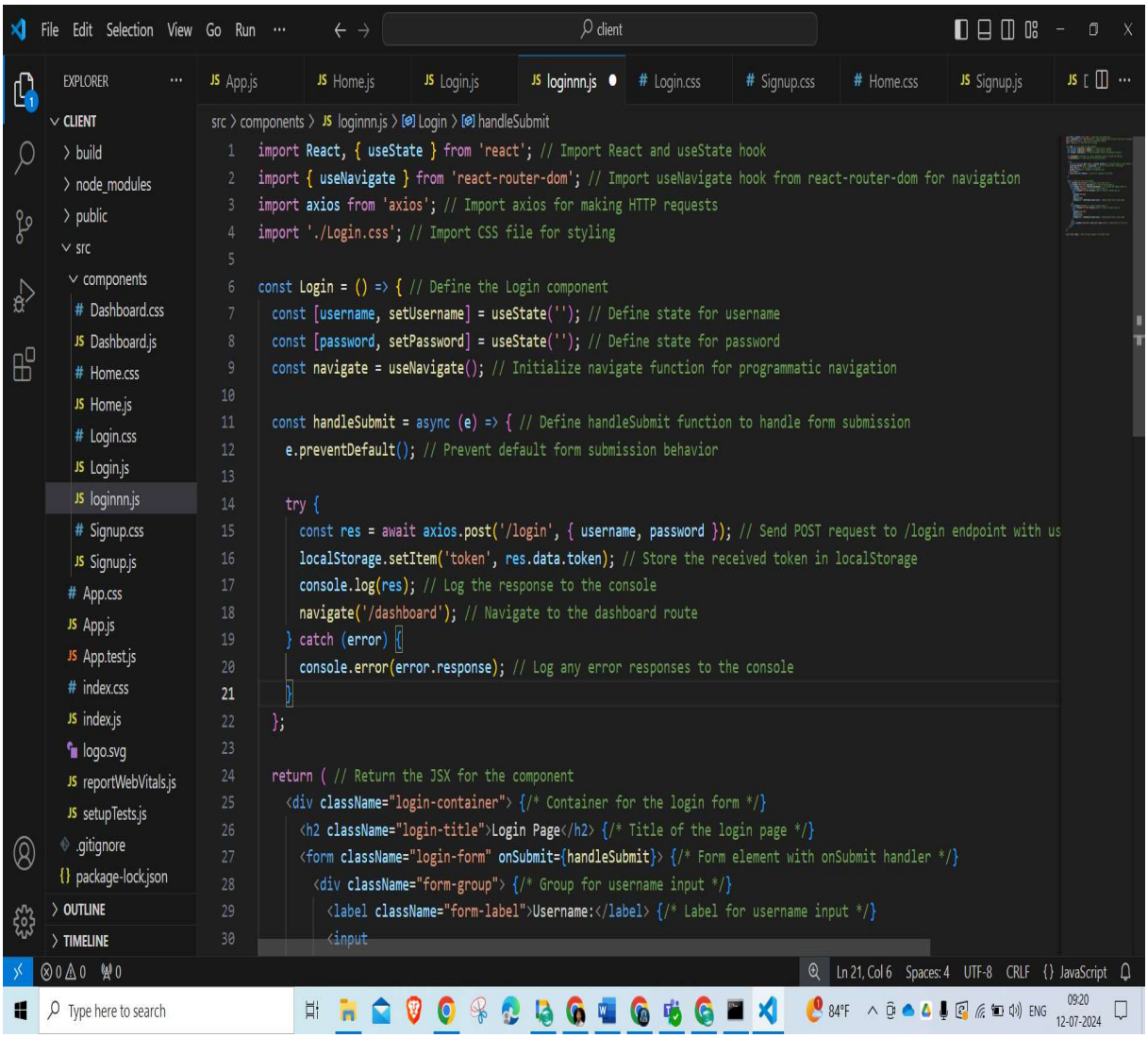
Data stored in the database

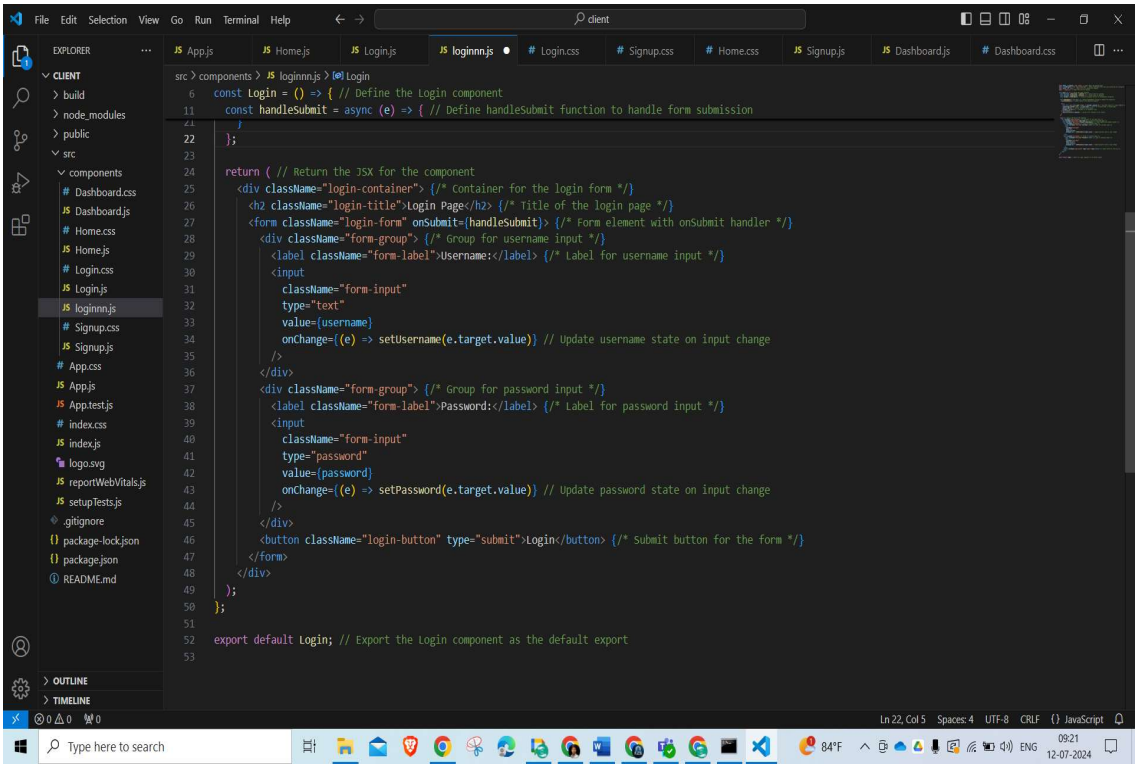


Building a Responsive Single Page Application (SPA) with React (Frontend)

It involves creating a modern web application that loads a single HTML page and dynamically updates content as users interact with it, providing a smooth user experience without full page reloads. Using React, a popular JavaScript library for building UIs, developers construct reusable components, manage application state efficiently, implement routing for navigation between different views, handle user input through forms with validation, integrate APIs for data fetching, and optimize performance with lazy loading and code splitting.

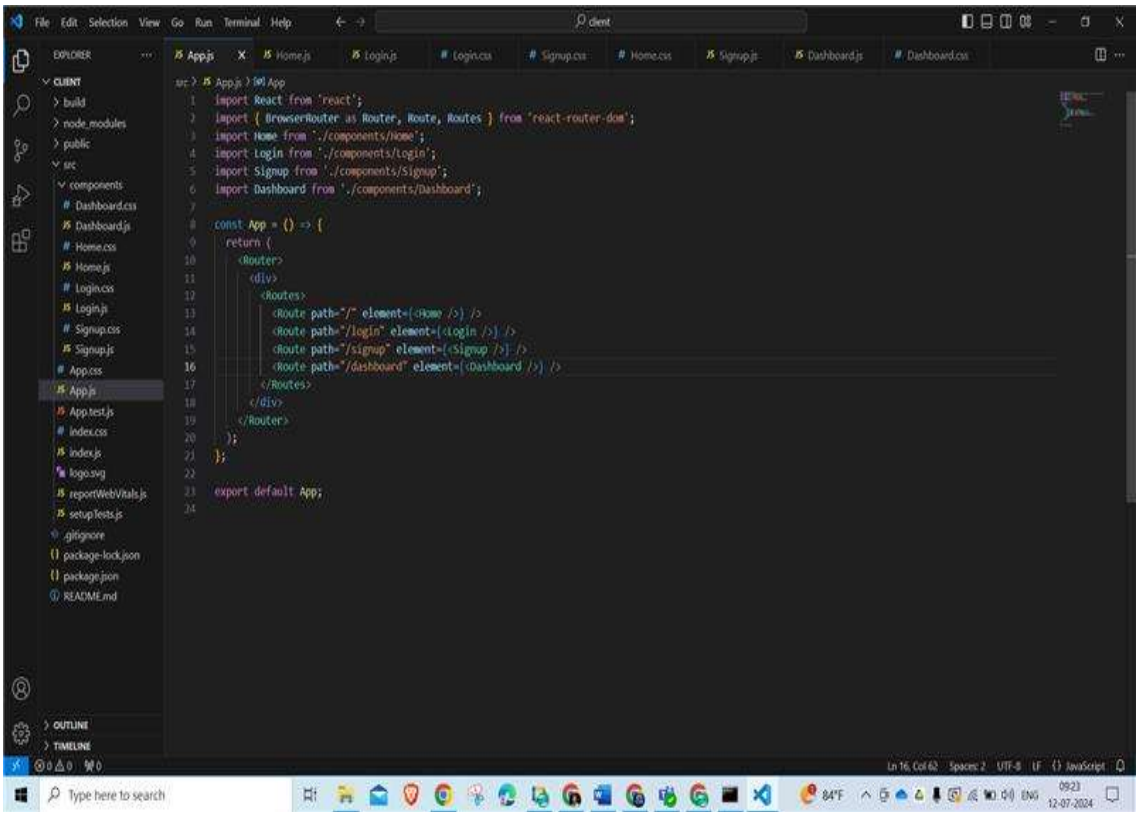
LOGIN.js





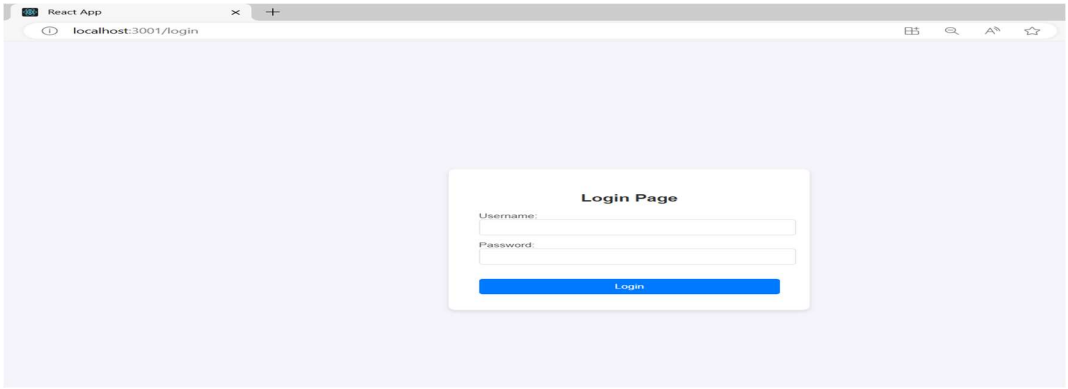
```
const Login = () => { // Define the Login component
  const handleSubmit = async (e) => { // Define handleSubmit function to handle form submission
    //
  };

  return ( // Return the JSX for the component
    <div className="login-container"> {/* Container for the login form */}
      <h2 className="login-title">Login Page</h2> {/* Title of the login page */}
      <form className="login-form" onSubmit={handleSubmit}> {/* Form element with onSubmit handler */}
        <div className="form-group"> {/* Group for username input */}
          <label className="form-label">Username:</label> {/* Label for username input */}
          <input
            className="form-input"
            type="text"
            value={username}
            onChange={(e) => setUsername(e.target.value)} // Update username state on input change
          />
        </div>
        <div className="form-group"> {/* Group for password input */}
          <label className="form-label">Password:</label> {/* Label for password input */}
          <input
            className="form-input"
            type="password"
            value={password}
            onChange={(e) => setPassword(e.target.value)} // Update password state on input change
          />
        </div>
        <button className="login-button" type="submit">Login</button> {/* Submit button for the form */}
      </form>
    </div>
  );
};

export default Login; // Export the Login component as the default export
```

APP.JS



```
import React from 'react';
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';
import Home from './components/Home';
import Login from './components/Login';
import Signup from './components/Signup';
import Dashboard from './components/Dashboard';

const App = () => {
  return (
    <Router>
      <div>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/login" element={<Login />} />
          <Route path="/signup" element={<Signup />} />
          <Route path="/dashboard" element={<Dashboard />} />
        </Routes>
      </div>
    </Router>
  );
};

export default App;
```

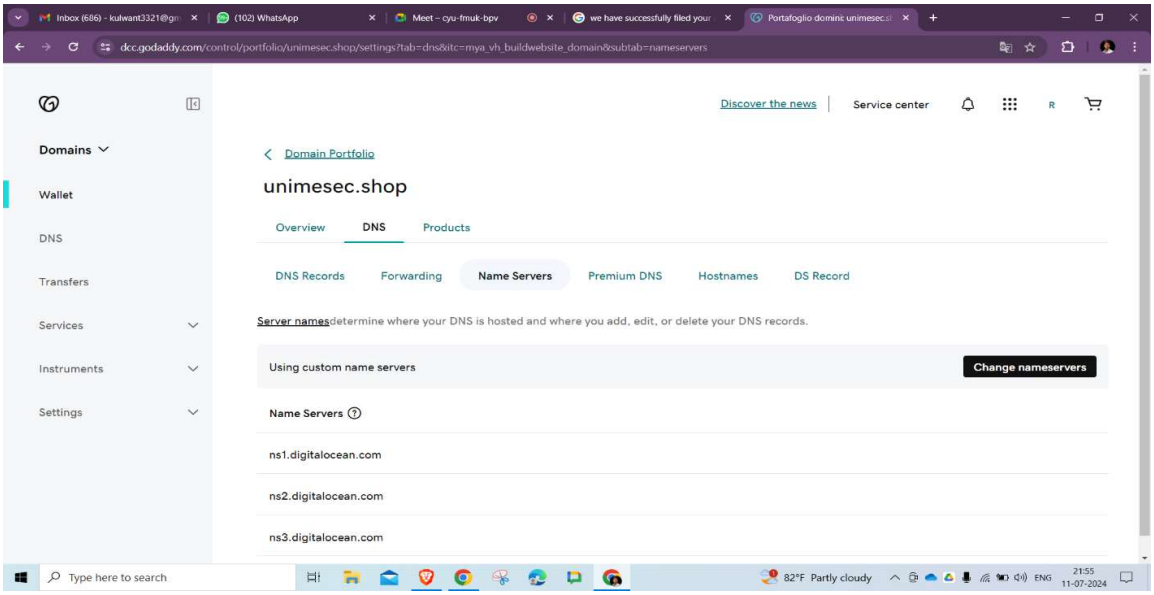
Visualisation



DEPLOYMENT AND HOSTING

Hosting is necessary for making our website accessible on the internet. When we create a website, the files (like HTML, CSS, JavaScript, images, etc.) need to be stored on a server that is connected to the internet 24/7. Hosting companies provide this service by renting out server space and infrastructure, allowing our website to be available to users worldwide.

DOMAIN SET-UP: GODADDY IS THE PROVIDER OF THE PAID DOMAIN REGISTRATION SERVICES

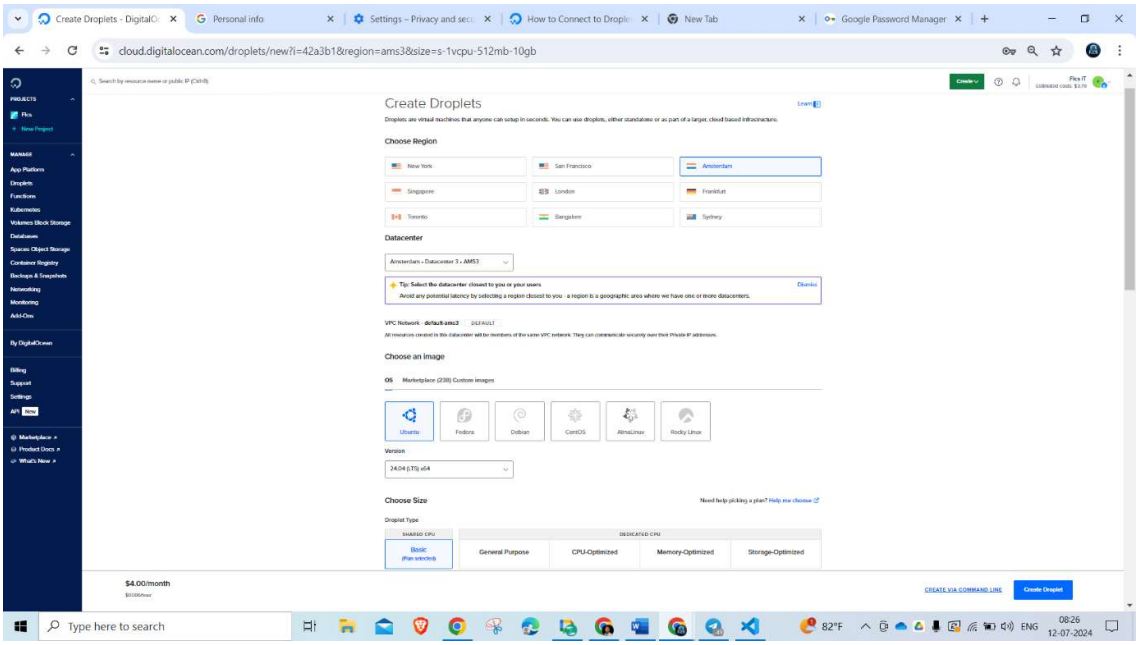


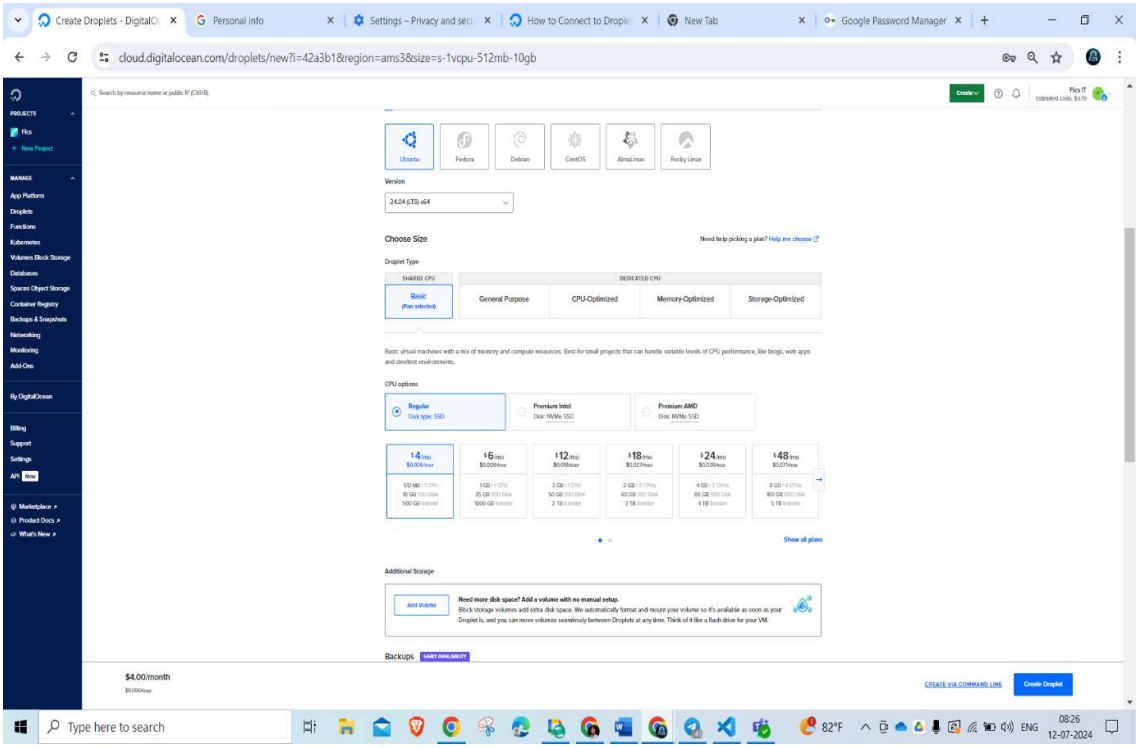
DOMAIN NAME: unimesec.shop

<u>SERVER SET-UP:</u> DIGITALOCEAN IS THE PROVIDER OF PAID SERVER HOSTING SERVICES CALLED DROPLETS USING UBUNTU DISTRIBUTION.

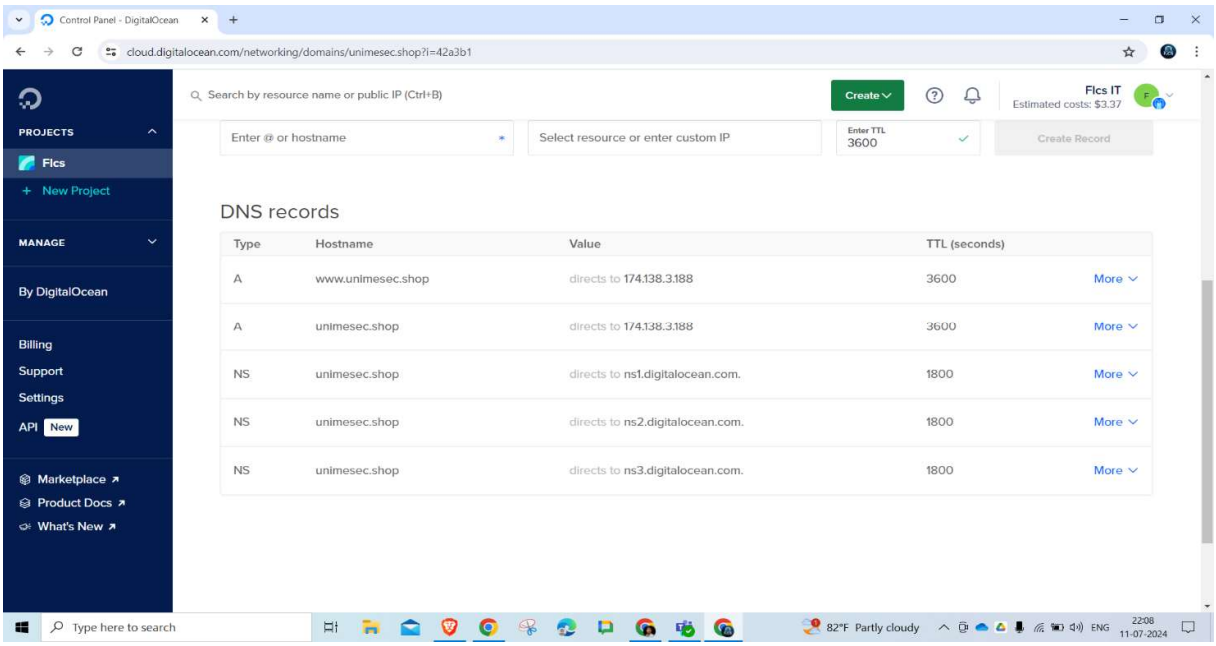
ServerName: unimeSEC

CREATING DROPLETS:

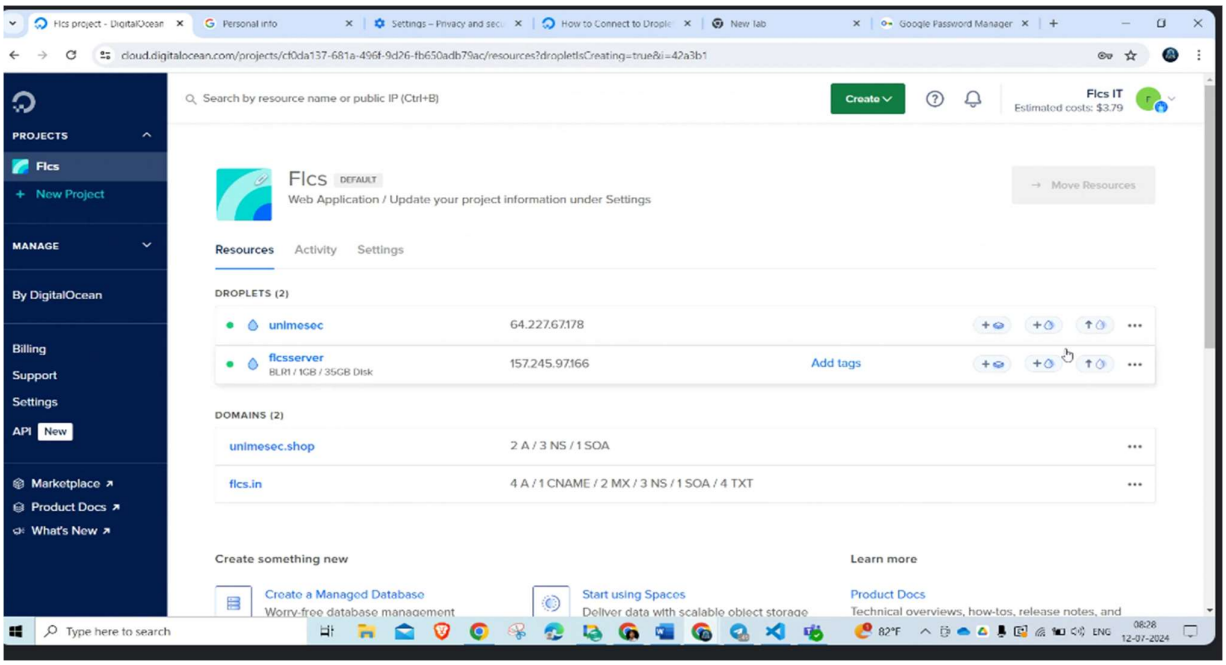




DNS RECORDS:



IP ADDRESS OF THE SERVER (unimesec):



CODE AT SERVER CONSOLE:

1. $ sudo apt-get install git-Installs Git, a version control system.

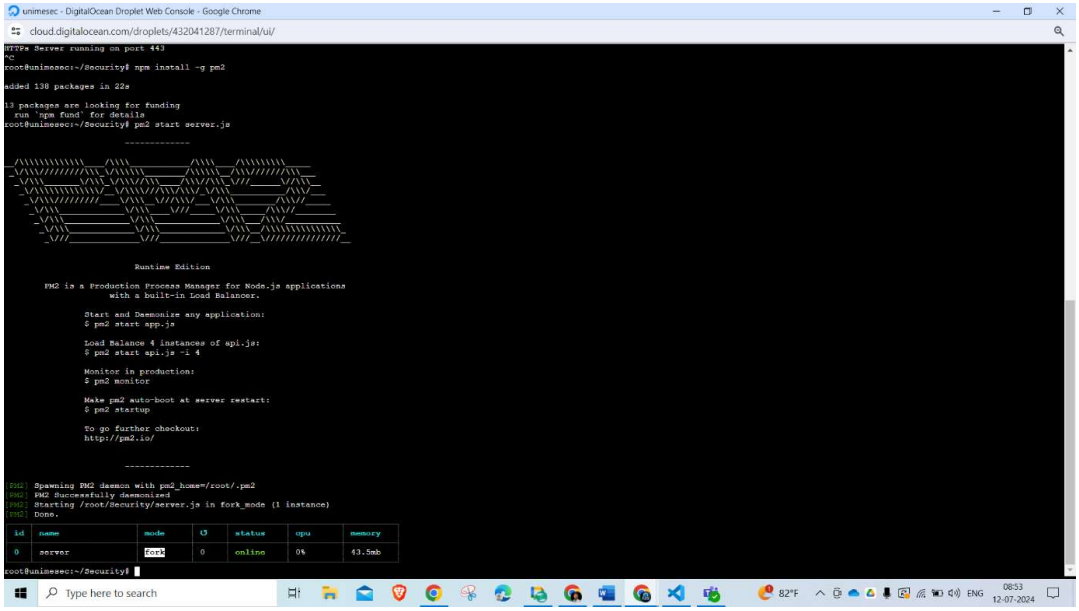
2. $ git –version: Verifies the installation by displaying the installed version of Git.

3. $ sudo apt-get install nodejs : Installs Node.js, a JavaScript runtime.
4. $ node –version: Verifies the installation by displaying the installed version of Node.js.
5. $ git clone https://github.com/Kulwant2024/Security.git: Clones the repository from GitHub to your local machine.

6.$ cd Security: Change the current directory to the Security directory.

7.$ apt-get install npm: Installing npm(node package manager), which is used to manage Node.js packages.

8.$ npm install: Installing the dependencies listed in the package.json file of the project.
9.$ node server: Starting the server using Node.js.
10.$ nano server.js: Opens the server.js file in the nano text editor for editing.
11.$ sudo apt-get install certbot: Installing Certbot, a tool to obtain SSL certificates from Let's Encrypt.
12.$ sudo certbot certonly --standalone -d unimesec.shop -d www.unimesec.shop.: Uses Certbot to obtain SSL certificates for the specified domains.
Output: Certificate is saved at: /etc/letsencrypt/live/unimesec.shop/fullchain.pem
Key is saved at: /etc/letsencrypt/live/unimesec.shop/privkey.pem
13.$ npm install -g pm2: Installs PM2, a process manager for Node.js applications, globally on our system.
14.$ pm2 start server.js: Starts the server.js file as a process managed by PM2.
15.$ pm2 restart server.js: Restarts the server.js process managed by PM2.



CODE TO BLOCK ALL OTHER PORTS EXCEPT 443 AND 22 IN UBUNTU SERVER

1. $ sudo apt-get update

2.$ sudo apt-get install ufw: Installing UFW (uncomplicated firewall), a user-friendly interface for managing firewall rules.

3.$ sudo ufw enable: Enableing UFW, activating the firewall with default settings. By default, it will block all incoming connections and allow all outgoing connections.

4.$ sudo ufw allow 443: Configuring the firewall to allow incoming traffic on port 443, which is used for HTTPS.

5.$ sudo ufw allow 22: Configures the firewall to allow incoming traffic on port 22, which is used for HTTPS.

WHY WE USE UFW (UNCOMPLICATED FIREWALL)

UFW (Uncomplicated Firewall) is a user-friendly frontend for managing a firewall in Linux, specifically designed to simplify the complexities of configuring iptables. Key features include:

- **Ease of Use**: Simple command-line interface for straightforward configuration.
- **Default Deny Policy**: Blocks all incoming connections by default, allowing only specified exceptions.
- **Predefined Rules**: Comes with preset rules for common applications like SSH, HTTP, and HTTPS.
- **Logging**: Offers logging options to monitor firewall activity.
- **IPv6 Support**: Handles both IPv4 and IPv6 traffic.

TO ACCESS REMOTELY THE SERVER:

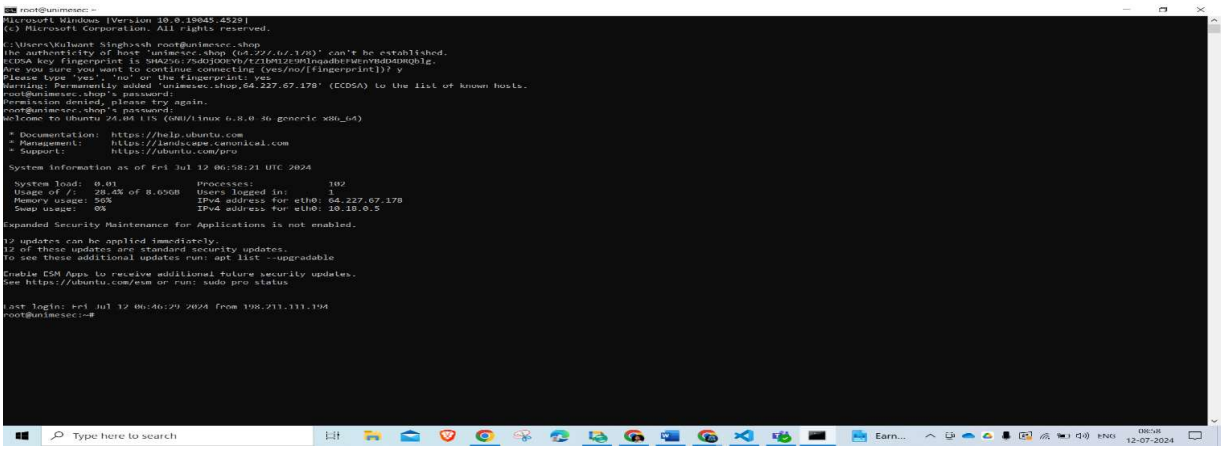
ssh root@unimesec.shop: root password: Italy@#2024SEC

This a command to remotely access the server unimesec.shop with the root user account via the Secure Shell (SSH) protocol. It's uses:

1. **Remote Access**: SSH allows secure remote login to the server, enabling management and configuration from anywhere.
2. **Root Privileges**: Logging in as root provides administrative privileges, allowing full control over the server.
3. **Security**: SSH encrypts the connection, protecting data from being intercepted or tampered with during transmission.

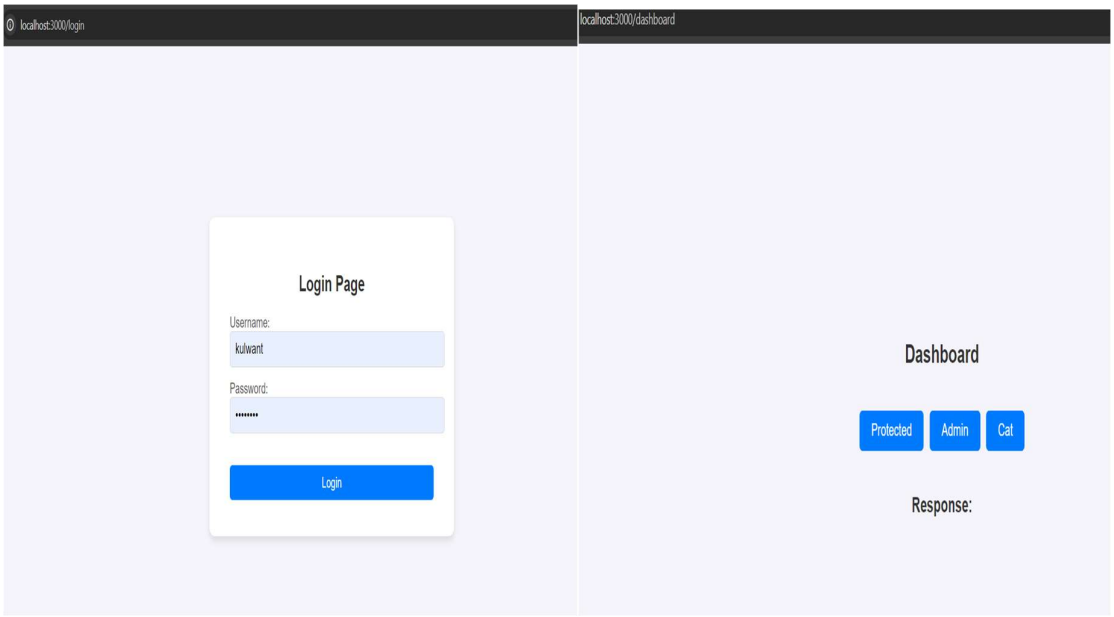
4. **Management**: Ideal for server maintenance, software installation, and performing system updates or troubleshooting.

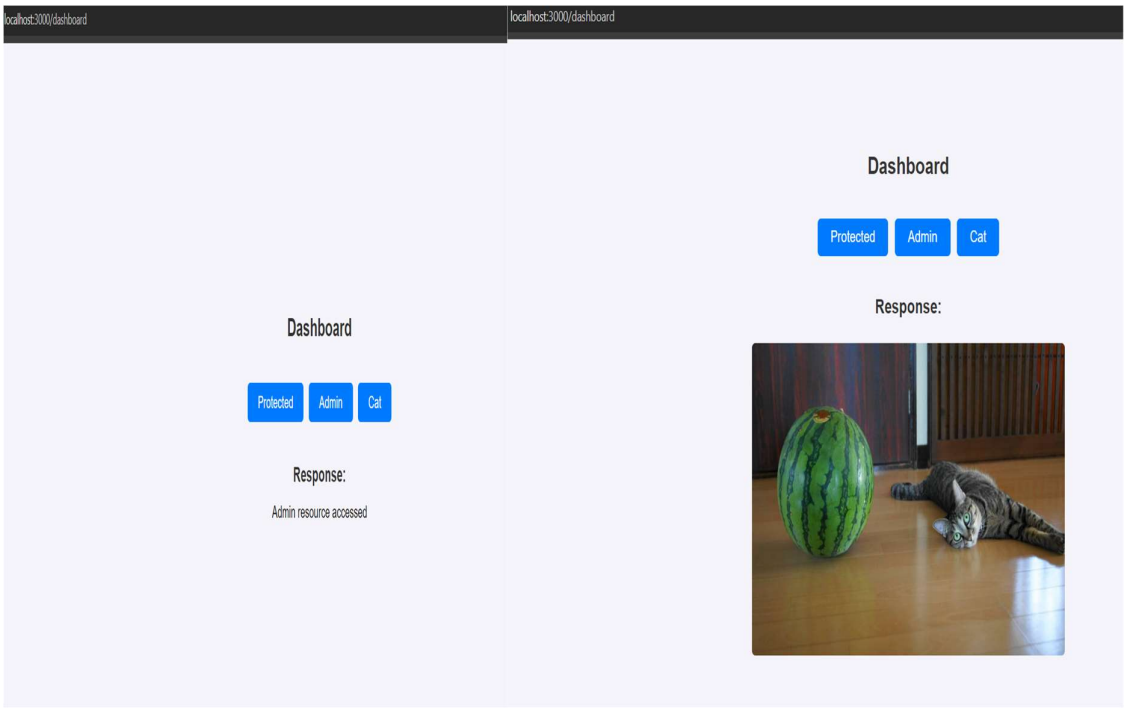
SSH PORT 22 IS WORKING:

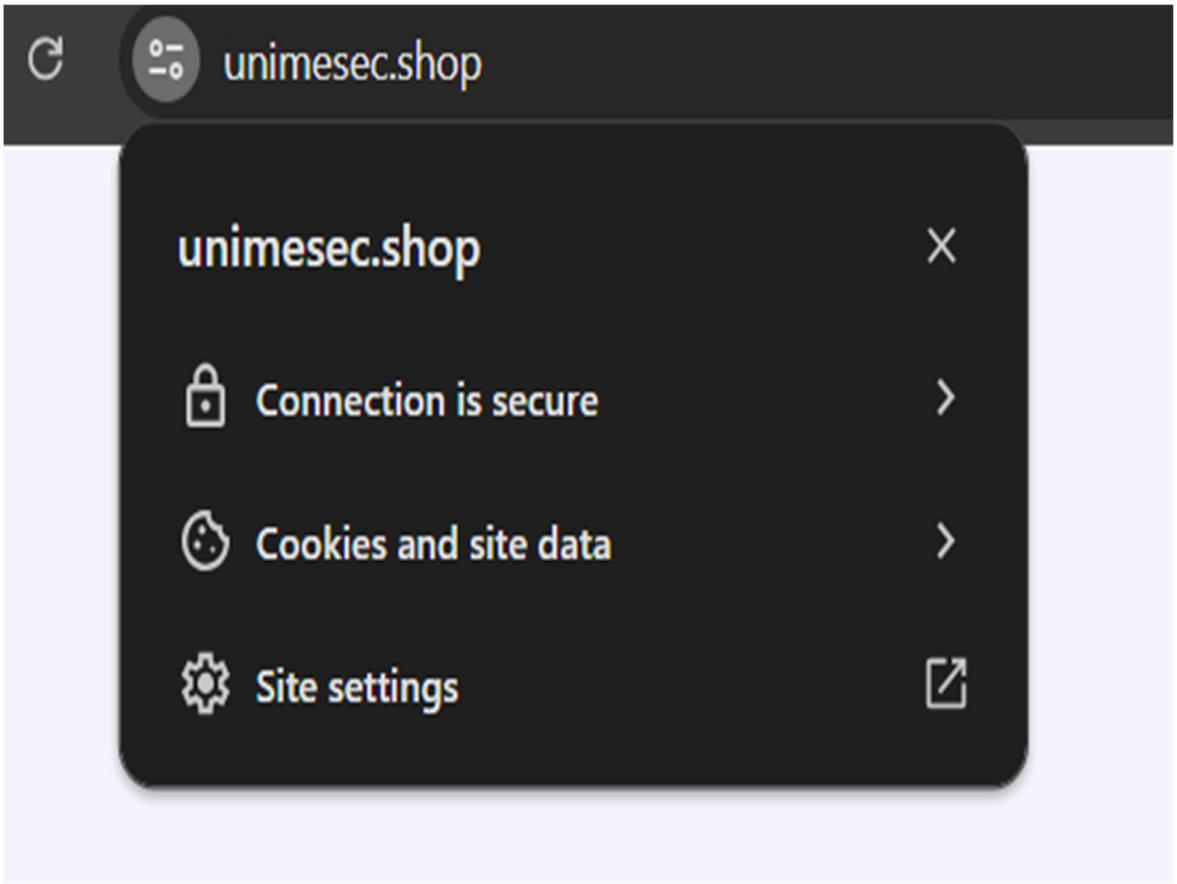


OUTPUTS

WHILE RUNNING THE SERVER AT LOCAL MACHINE USING PORT 3000




localhost:3000/dashboard
Dashboard
Protected
Admin
Cat
Response:
Admin resource accessed
localhost:3000/dashboard
Dashboard
Protected
Admin
Cat
Response:

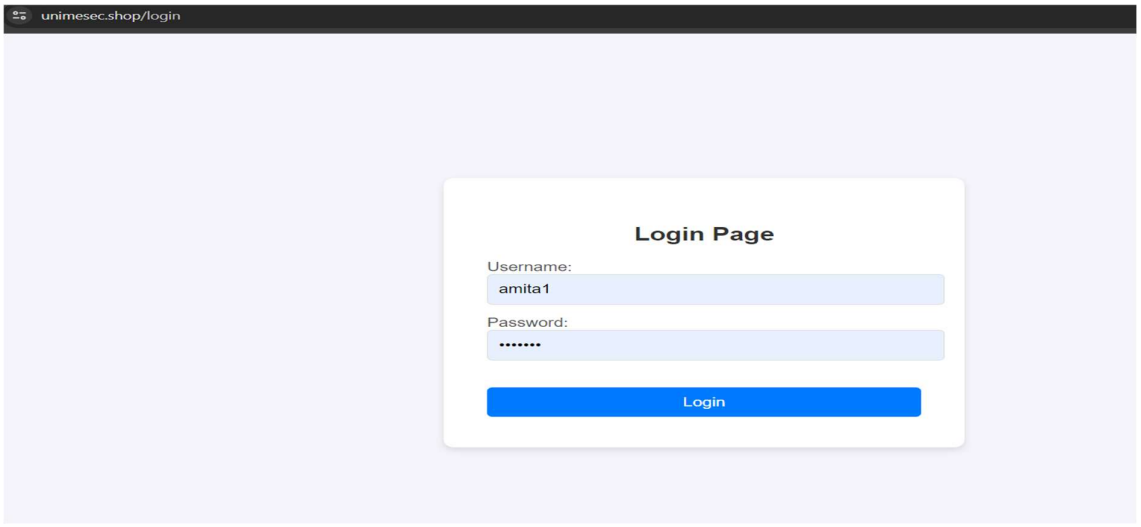

unimesec.shop
unimesec.shop
Connection is secure
Cookies and site data
Site settings

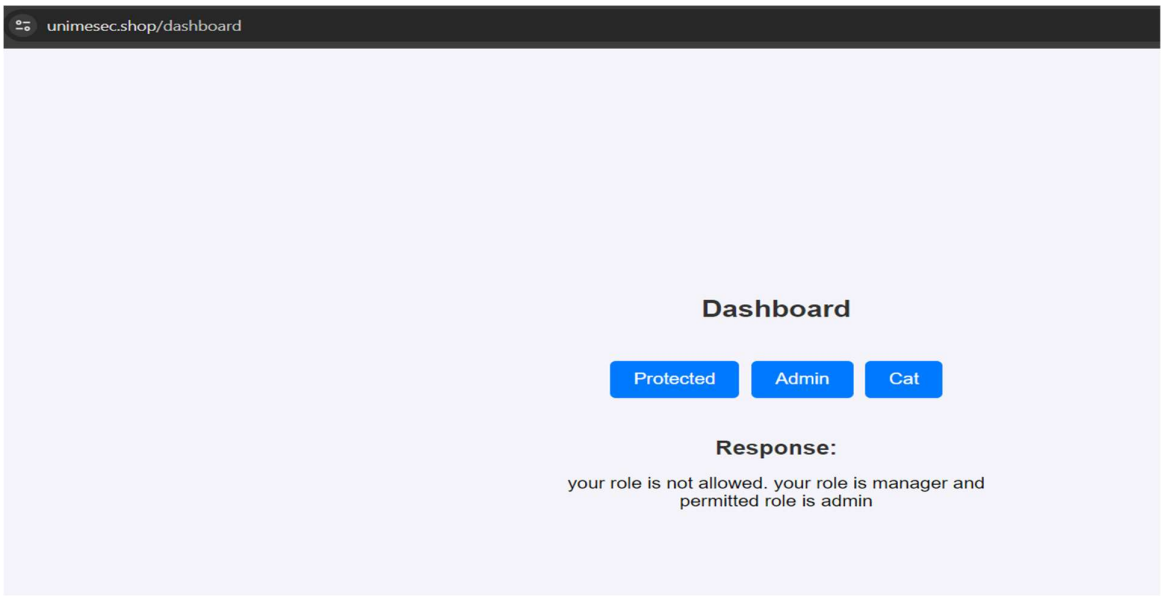
WHILE RUNNING THE SERVER AT UNIMESEC WITH IP (64.227.67.178) WITH OUR DOMAIN WWW.UNIMESEC.SHOP AT PORT 443

1.USING SSL CERTIFICATES MAKING OUR DOMAIN SECURE

2.APPLICATION IS RUNNING SECURELY AT OUR DOMAIN.

https://www.unimesec.shop/

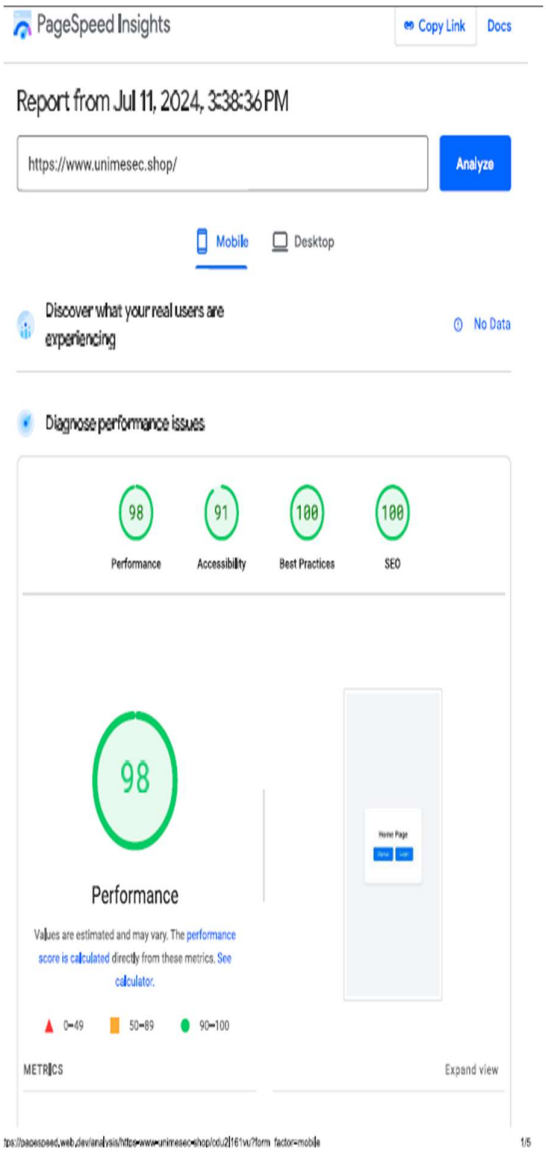


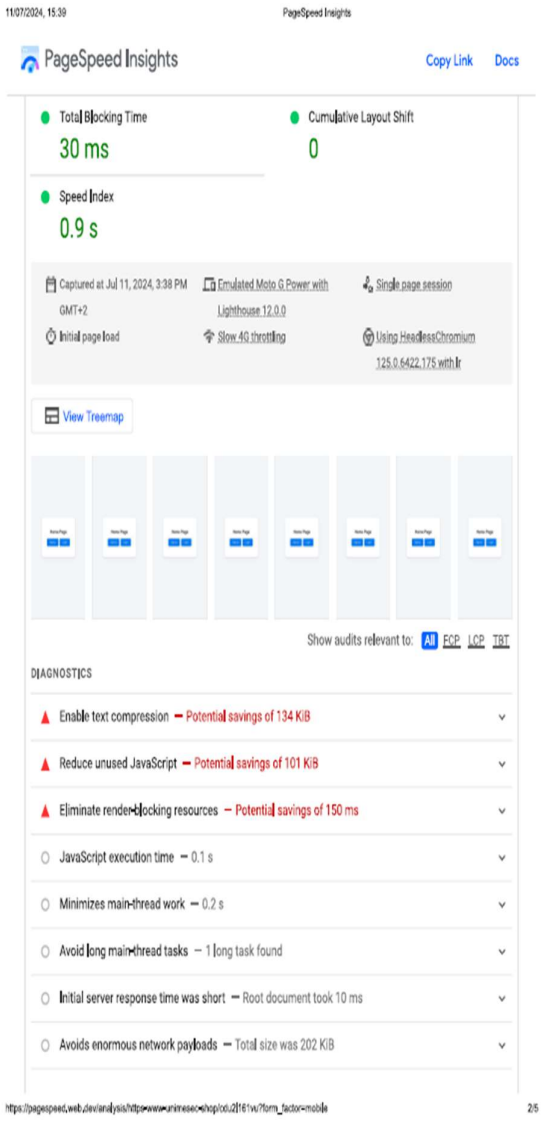


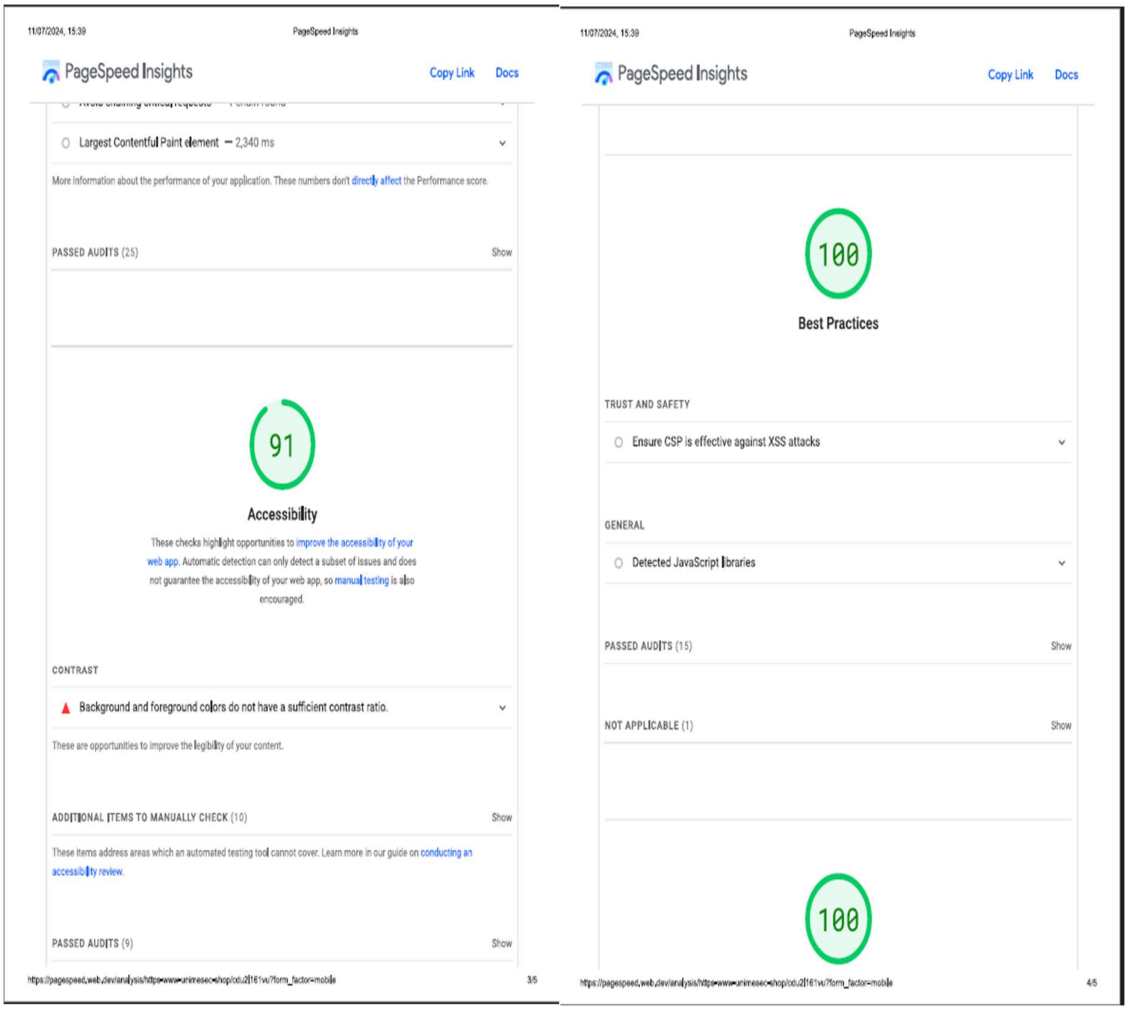
PERFORMANCE/EVALUTION CHECK

https://www.unimesec.shop/dashboard WITH THE HELP OF

https://pagespeed.web.dev/analysis






11/07/2024, 15:39
PageSpeed Insights
PageSpeed Insights
Copy Link
Docs
Largest Contentful Paint element — 2,340 ms
More information about the performance of your application. These numbers don't directly affect the Performance score.
PASSED AUDITS (25)
Show
91
Accessibility
These checks highlight opportunities to improve the accessibility of your web app. Automatic detection can only detect a subset of issues and does not guarantee the accessibility of your web app, so manual testing is also encouraged.
CONTRAST
Background and foreground colors do not have a sufficient contrast ratio.
These are opportunities to improve the legibility of your content.
ADDITIONAL ITEMS TO MANUALLY CHECK (10)
Show
These items address areas which an automated testing tool cannot cover. Learn more in our guide on conducting an accessibility review.
PASSED AUDITS (9)
Show
https://pagespeed.web.dev/analysis/http-www-uninesec-shop-cdi2f161w/?form_factor=mobile
3/5
11/07/2024, 15:39
PageSpeed Insights
PageSpeed Insights
Copy Link
Docs
100
Best Practices
TRUST AND SAFETY
Ensure CSP is effective against XSS attacks
GENERAL
Detected JavaScript libraries
PASSED AUDITS (15)
Show
NOT APPLICABLE (1)
Show
100
https://pagespeed.web.dev/analysis/http-www-uninesec-shop-cdi2f161w/?form_factor=mobile
4/5

Conclusion:

The SecureNet API Shield project has successfully integrated Secure DNS (DNSSEC) and JWT (JSON Web Token) authentication to create a comprehensive, secure, and scalable server-side application. This strategic merger addresses critical security concerns by enhancing both network and application-level defenses, ensuring robust protection against a multitude of cyber threats.

Achievements and Benefits:

1. **Enhanced Security**:
    - **DNSSEC Integration**: DNSSEC (Domain Name System Security Extensions) implementation has fortified the DNS infrastructure. It ensures the authenticity and integrity of DNS responses, protecting against DNS spoofing, cache poisoning, and related attacks. This guarantees that users are always directed to the legitimate servers, reducing the risk of man-in-the-middle attacks and other malicious activities.
    - **JWT Authentication**: JWT (JSON Web Token) authentication provides a secure and efficient way to handle user authentication and authorization. JWTs enable stateless authentication, eliminating the need for server-side session storage, which simplifies the architecture and improves scalability. Each token is signed and can be easily verified, ensuring that only authenticated and authorized users can access protected resources.
2. **Improved Data Integrity and Confidentiality**:
    - **Encrypted Traffic**: By leveraging TLS (Transport Layer Security), all data transmitted between clients and the server is encrypted. This encryption ensures that data remains confidential and protected from eavesdropping and tampering during transmission.
    - **Secure Access Control**: JWT allows for granular access control, enabling the application to grant or restrict access based on user roles and permissions. This ensures that sensitive information and critical operations are only accessible to authorized individuals, thereby maintaining data integrity and confidentiality.
3. **Scalability and Efficiency**:
    - **Stateless Authentication**: JWT's stateless nature significantly reduces the overhead associated with managing user sessions on the server. This enhances scalability as the server can handle more concurrent users without the need to manage session states.
    - **Modular Architecture**: The use of Node.js and Express.js, coupled with a modular approach to managing dependencies and routes, ensures that the application is both flexible and scalable. This architecture facilitates easy updates, maintenance, and the addition of new features.
4. **Comprehensive Testing and Validation**:
    - **API Testing with Postman**: Extensive use of Postman for API testing has ensured that all endpoints function as expected. This includes validating JWT authentication and checking DNSSEC configurations, providing confidence in the system's reliability and security.
    - **Performance Monitoring**: Regular performance checks and optimizations have ensured that the application remains responsive and efficient under varying load conditions. This ongoing monitoring helps in identifying and addressing performance bottlenecks proactively.

Future Scope and Applications

1. **Advanced Threat Detection:**

Integrating machine learning algorithms can further enhance security by detecting and responding to emerging threats in real-time. This will enable proactive defence mechanisms against sophisticated attacks targeting both DNS and API layers.

2. **IoT Device Security:**

Extending Secure DNS and JWT authentication to Internet of Things (IoT) devices will ensure secure communication and access control within IoT ecosystems. This will protect against unauthorized access and data breaches in IoT environments.

3. **Zero Trust Architecture:**

Implementing a Zero Trust security model, where every access request is thoroughly authenticated, authorized, and encrypted, regardless of its origin, will further strengthen security. This model ensures continuous verification of every user and device accessing the network.

4. **Decentralized Applications:**

Combining blockchain technology with Secure DNS and JWT authentication can enhance the security of decentralized applications (DApps). This integration will provide immutable and verifiable transaction records, ensuring trust and security in decentralized environments

5. **Comprehensive Compliance:**

Meeting stringent regulatory requirements (e.g., GDPR, HIPAA) will be facilitated by maintaining robust data protection and secure access mechanisms throughout the application. Ensuring compliance with these regulations will enhance trust and reliability among users and stakeholders.

6. **Scalable Microservices Security:**

Secure microservices architectures by ensuring that each service communicates securely with others using encrypted DNS and authenticated API calls.

Final Thoughts:

The Secure Net API Shield project represents a significant advancement in the field of web application security. By merging the strengths of Secure DNS and JWT authentication, the project delivers a holistic security solution that addresses both current and emerging threats. This integration not only enhances the security of web services but also lays a solid foundation for future innovations in secure communication and data protection.

The careful planning, execution, and continuous performance evaluation adopted in this project ensure that the application is well-equipped to handle the demands of a rapidly evolving digital landscape. Developers can leverage this comprehensive guide to build secure, scalable, and efficient applications that are resilient to various cyber threats.

In conclusion, the Secure Net API Shield project exemplifies how strategic integration of advanced security technologies can create a robust defence framework, ensuring the safety, integrity, and reliability of web services for users and businesses alike.