

FINAL PROJECT OF COMPUTER SYSTEM SECURITY



**Università
degli Studi
di Messina**



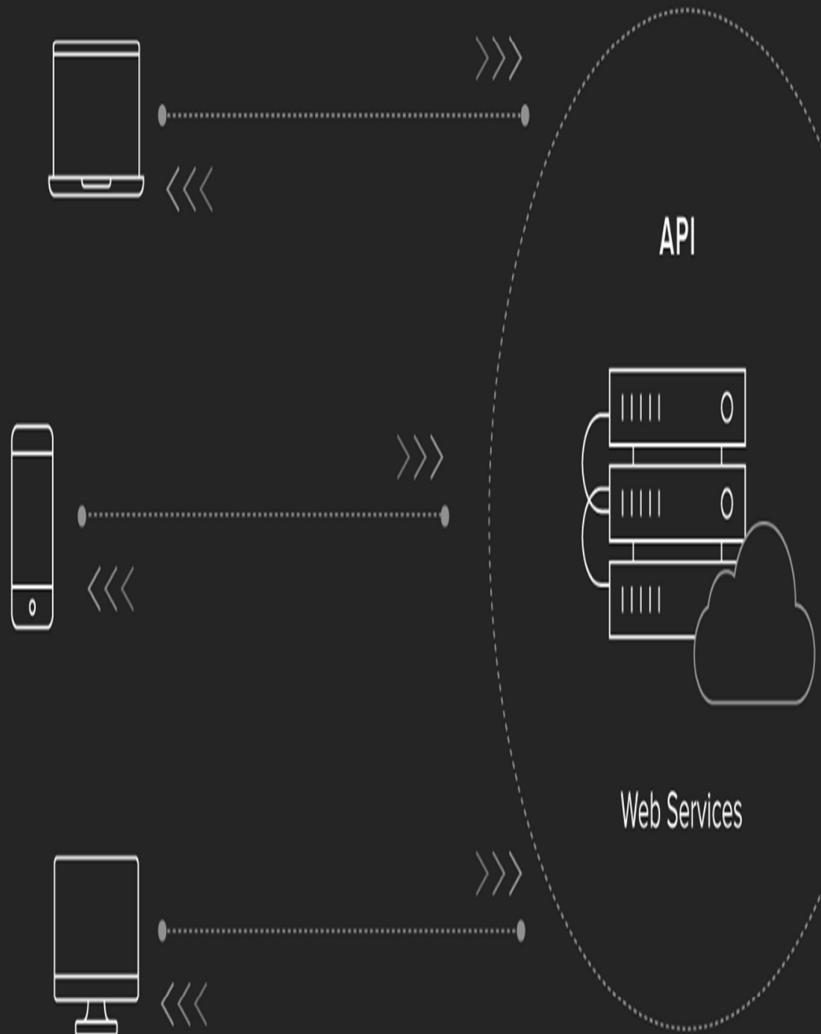
**Submitted by: Kulwant Singh Rathore (555368)
Amita Rajak (545238)**

Guided by: Prof. Massimo Villari

Course: Engineering and Computer Science

SecureNet API Shield: Unified DNS and JWT Security

REST API Security Best Practices



CONTENTS

1. ABSTRACT	4
A concise guide to developing, deploying, and securing a server-side application on Ubuntu 20.04.	
2. OVERVIEW	5
Covers server setup, dependency management, API testing, authentication, database creation, UI development, deployment, and security.	
3. SERVER-SIDE DEVELOPMENT	6
Create a backend with Node.js and Express.js.	
4. DEPENDENCIES	10
Manage libraries like Express.js, Mongoose, and JWT.	
5. POSTMAN	12
Use Postman for API testing.	
6. JSON WEB TOKEN (JWT)	13
Implement JWT for user authentication.	
7. DATABASE CREATION	16
Set up and manage MongoDB with Mongoose.	
8. USER INTERFACE	18
Develop a responsive UI using HTML, CSS, and React.	
9. DEPLOYMENT AND HOSTING	20
Deploy to AWS or Heroku and configure hosting.	
10. UBUNTU 20.04	21
Set up and configure Ubuntu 20.04 for hosting.	
11. UNCOMPLICATED FIREWALL (UFW)	24
Secure the server with UFW.	
12. REMOTE SERVER ACCESS	25
Access the server using SSH.	
13. OUTPUTS	26
Validate API responses, database entries, and UI.	
14. PERFORMANCE CHECK	27
Monitor and optimize performance.	
15. CONCLUSION	30
Effective server-side development involves careful planning, execution, regular performance checks, and security measures.	

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

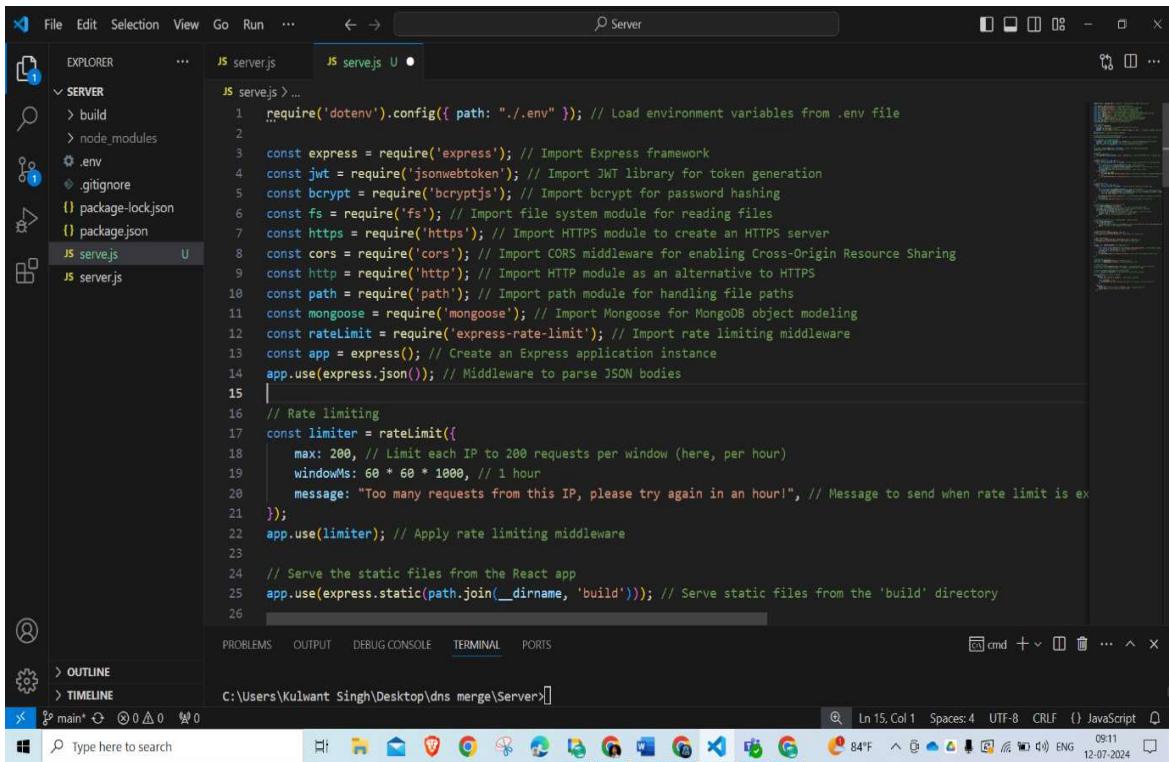
By integrating Secure DNS and JWT, the application benefits from enhanced security at both the network and application levels. DNSSEC protects against DNS attacks, ensuring the integrity and authenticity of DNS data, while JWT provides a robust mechanism for user authentication and authorization, controlling 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 INTEGRATION

- ❖ **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



The screenshot shows a code editor interface with the following details:

- File Path:** C:\Users\Kulwant Singh\Desktop\dns merge\Server>
- Code Content (server.js):**

```
1  require('dotenv').config({ path: './.env' }); // Load environment variables from .env file
2
3  const express = require('express'); // Import Express framework
4  const jwt = require('jsonwebtoken'); // Import JWT library for token generation
5  const bcrypt = require('bcryptjs'); // Import bcrypt for password hashing
6  const fs = require('fs'); // Import file system module for reading files
7  const https = require('https'); // Import HTTPS module to create an HTTPS server
8  const cors = require('cors'); // Import CORS middleware for enabling Cross-Origin Resource Sharing
9  const http = require('http'); // Import HTTP module as an alternative to HTTPS
10 const path = require('path'); // Import path module for handling file paths
11 const mongoose = require('mongoose'); // Import Mongoose for MongoDB object modeling
12 const rateLimit = require('express-rate-limit'); // Import rate limiting middleware
13 const app = express(); // Create an Express application instance
14 app.use(express.json()); // Middleware to parse JSON bodies
15
16 // Rate limiting
17 const limiter = rateLimit({
18   max: 200, // Limit each IP to 200 requests per window (here, per hour)
19   windowMs: 60 * 60 * 1000, // 1 hour
20   message: "Too many requests from this IP, please try again in an hour!", // Message to send when rate limit is exceeded
21 });
22 app.use(limiter); // Apply rate limiting middleware
23
24 // Serve the static files from the React app
25 app.use(express.static(path.join(__dirname, 'build'))); // Serve static files from the 'build' directory
```

- Toolbars and Status Bar:** The status bar at the bottom shows the current file is a JavaScript file (JavaScript), the line and column numbers (Ln 15, Col 1), and the file encoding (UTF-8). The system tray indicates the date and time (12-07-2024) and battery level (84%).

```
File Edit Selection View Go Run ... ← → ⌂ Server
EXPLORER JS server.js JS servejs U ●
JS server.js > ...
16 // Rate limiting
17 const limiter = rateLimit({
18   max: 200, // Limit each IP to 200 requests per window (here, per hour)
19   windowMs: 60 * 60 * 1000, // 1 hour
20   message: "Too many requests from this IP, please try again in an hour!", // Message to send when rate limit is exceeded
21 });
22 app.use(limiter); // Apply rate limiting middleware
23
24 // Serve the static files from the React app
25 app.use(express.static(path.join(__dirname, 'build'))); // Serve static files from the 'build' directory
26
27 const userSchema = new mongoose.Schema({
28   username: { type: String, required: true, unique: true }, // Define username field with constraints
29   password: { type: String, required: true }, // Define password field with constraints
30   role: { type: String, required: true } // Define role field with constraints
31 });
32
33 const User = mongoose.model('User', userSchema); // Create a User model using the schema
34 app.use(cors('*')); // Enable CORS for all routes
35
36 // '/' route
37 app.get('/', (req, res) => {
38   res.sendFile(path.join(__dirname, 'build', 'index.html')); // Serve the main HTML file for the React app
39 });
40
41 // Register a new user
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
C:\Users\Kulwant Singh\Desktop\dns merge\Server>
cmd + v ⌂ ... ^ x
Type here to search ⌂ ln 15, Col 1 Spaces:4 UTF-8 CRLF {} JavaScript 09:11
Air... ⌂ 12-07-2024
```

```
File Edit Selection View Go Run ... ← → ⌂ Server
EXPLORER JS server.js JS servejs U ●
JS server.js > ...
36 // '/' route
37 app.get('/', (req, res) => {
38   res.sendFile(path.join(__dirname, 'build', 'index.html')); // Serve the main HTML file for the React app
39 });
40
41 // Register a new user
42 app.post('/signup', async (req, res) => {
43   const { username, password, role } = req.body; // Extract username, password, and role from request body
44   try {
45     const hashedPassword = bcrypt.hashSync(password, 10); // Hash the password
46     const user = new User({ username, password: hashedPassword, role }); // Create a new user instance
47     await user.save(); // Save the user to the database
48     res.status(201).send('User registered successfully'); // Send success response
49   } catch (error) {
50     if (error.code === 11000) {
51       res.status(409).send('Username already exists'); // Send conflict response if username already exists
52     } else {
53       res.status(500).send('Error registering user'); // Send server error response for other errors
54     }
55   }
56 });
57
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 {
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
C:\Users\Kulwant Singh\Desktop\dns merge\Server>
cmd + v ⌂ ... ^ x
Type here to search ⌂ ln 15, Col 1 Spaces:4 UTF-8 CRLF {} JavaScript 09:12
Air... ⌂ 12-07-2024
```

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a project structure with files like `build`, `node_modules`, `.env`, `.gitignore`, `package-lock.json`, `package.json`, `JS serve.js`, and `JS server.js`.
- Code Editor:** Displays the `server.js` file content. The code handles user authentication using JWT tokens.
- Terminal:** Shows the command line interface with various icons and status information.
- Status Bar:** Provides file path (`C:\Users\Kulwant Singh\Desktop\dns merge\Server>`), line count (Ln 15, Col 1), and encoding (UTF-8).

```
JS server.js > ...
57 // Generate JWT Token
58 app.post('/login', async (req, res) => {
59   const { username, password } = req.body; // Extract username and password from request body
60   try {
61     const user = await User.findOne({ username }); // Find user by username
62     if (user && bcrypt.compareSync(password, user.password)) { // Check if user exists and passwords match
63       const token = jwt.sign({ username: user.username, role: user.role }, process.env.JWT_SECRET, { expiresIn: '1h' });
64       return res.json({ token }); // Send token in response
65     }
66     res.status(401).send('Invalid credentials'); // Send unauthorized response if credentials are invalid
67   } catch (error) {
68     res.status(500).send('Error logging in'); // Send server error response for other errors
69   }
70 });
71 // Middleware to validate JWT //to check the tokens
72 function authenticateToken(req, res, next) {
73   const token = req.headers['authorization']; // Get token from authorization header
74   if (!token) return res.send('Token not send'); // Send response if no token is provided
75   jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
76     if (err) return res.send('Token is not valid'); // Send response if token is not valid
77     req.user = user; // Attach user information to request object
78     next(); // Proceed to next middleware or route handler
79   });
80 }
81
82 }
```

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a project structure with files like `build`, `node_modules`, `.env`, `.gitignore`, `package-lock.json`, `package.json`, `JS serve.js`, and `JS server.js`.
- Code Editor:** Displays the `server.js` file content. The code handles user authentication using JWT tokens.
- Terminal:** Shows the command line interface with various icons and status information.
- Status Bar:** Provides file path (`C:\Users\Kulwant Singh\Desktop\dns merge\Server>`), line count (Ln 15, Col 1), and encoding (UTF-8).

```
JS server.js > ...
57 // Generate JWT Token
58 app.post('/login', async (req, res) => {
59   const { username, password } = req.body; // Extract username and password from request body
60   try {
61     const user = await User.findOne({ username }); // Find user by username
62     if (user && bcrypt.compareSync(password, user.password)) { // Check if user exists and passwords match
63       const token = jwt.sign({ username: user.username, role: user.role }, process.env.JWT_SECRET, { expiresIn: '1h' });
64       return res.json({ token }); // Send token in response
65     }
66     res.status(401).send('Invalid credentials'); // Send unauthorized response if credentials are invalid
67   } catch (error) {
68     res.status(500).send('Error logging in'); // Send server error response for other errors
69   }
70 });
71 // Middleware to validate JWT //to check the tokens
72 function authenticateToken(req, res, next) {
73   const token = req.headers['authorization']; // Get token from authorization header
74   if (!token) return res.send('Token not send'); // Send response if no token is provided
75   jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
76     if (err) return res.send('Token is not valid'); // Send response if token is not valid
77     req.user = user; // Attach user information to request object
78     next(); // Proceed to next middleware or route handler
79   });
80 }
81
82 }
```

A screenshot of the Visual Studio Code (VS Code) interface. The title bar shows 'File Edit Selection View Go Run ...' and a search bar with 'Server'. The Explorer sidebar on the left shows a project structure with files like 'build', '.env', '.gitignore', 'package-lock.json', 'package.json', 'serve.js', and 'server.js'. The main editor area displays 'server.js' with code for a Node.js application. The code includes routes for protected and admin resources, fetching cat images from an API, and creating an HTTPS server using SSL/TLS certificates. The bottom status bar shows the file path 'C:\Users\Kulwant Singh\Desktop\dns merge\Server\server.js', line 'Ln 15, Col 1', and other settings.

```
91 // Protected Route Example
92 app.get('/protected', authenticateToken, (req, res) => {
93   res.send('Protected resource accessed'); // Send response for protected route
94 });
95
96
97 // Admin Route Example
98 app.get('/admin', authenticateToken, authorizeRole('admin'), (req, res) => {
99   res.send('Admin resource accessed'); // Send response for admin route
100 });
101
102 app.get('/cat', authenticateToken, async (req, res) => {
103   try {
104     const response = await fetch("https://api.thecatapi.com/v1/images/search"); // Fetch random cat image
105     const cat = await response.json(); // Parse response as JSON
106     res.json(cat); // Send cat image in response
107   } catch (error) {
108     res.status(500).send('Error fetching cat image'); // Send server error response for other errors
109   }
110 });
111
112 // Read SSL/TLS certificates
113 const key = fs.readFileSync('key.pem'); // Read SSL key file
114 const cert = fs.readFileSync('cert.pem'); // Read SSL certificate file
115
116 // Create HTTPS server
```

A screenshot of the Visual Studio Code (VS Code) interface. The title bar shows 'File Edit Selection View Go Run ...' and a search bar with 'Server'. The Explorer sidebar on the left shows a project structure with files like 'build', '.env', '.gitignore', 'package-lock.json', 'package.json', 'serve.js', and 'server.js'. The main editor area displays 'server.js' with code for a Node.js application. The code includes logic for reading SSL/TLS certificates, creating an HTTPS server, connecting to MongoDB using Mongoose, and starting the server. It also handles connection errors and logs successful connections. The bottom status bar shows the file path 'C:\Users\Kulwant Singh\Desktop\dns merge\Server\server.js', line 'Ln 115, Col 1', and other settings.

```
111
112 // Read SSL/TLS certificates
113 const key = fs.readFileSync('/etc/letsencrypt/live/unimesec.shop/privkey.pem'); // Read SSL key file
114 const cert = fs.readFileSync('/etc/letsencrypt/live/unimesec.shop/fullchain.pem'); // Read SSL certificate file
115
116 // Create HTTPS server
117 const server = https.createServer({ key, cert }, app); // Create HTTPS server with SSL certificates
118 // const server = http.createServer(app); // Uncomment to use HTTP server instead
119
120 // Connect to MongoDB using Mongoose and start the server
121 const url = process.env.MONGODB_URI; // Get MongoDB URI from environment variables
122
123 mongoose.connect(url) // Connect to MongoDB
124   .then(() => {
125     console.log('Connected to MongoDB'); // Log successful connection
126
127     server.listen(443, () => {
128       console.log('HTTPS Server running on port 443'); // Start HTTPS server and log the port
129     });
130   })
131   .catch(err => {
132     console.error('Failed to connect to MongoDB', err); // Log connection error
133     process.exit(1); // Exit process with failure code
134   });
135
```

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 our 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.

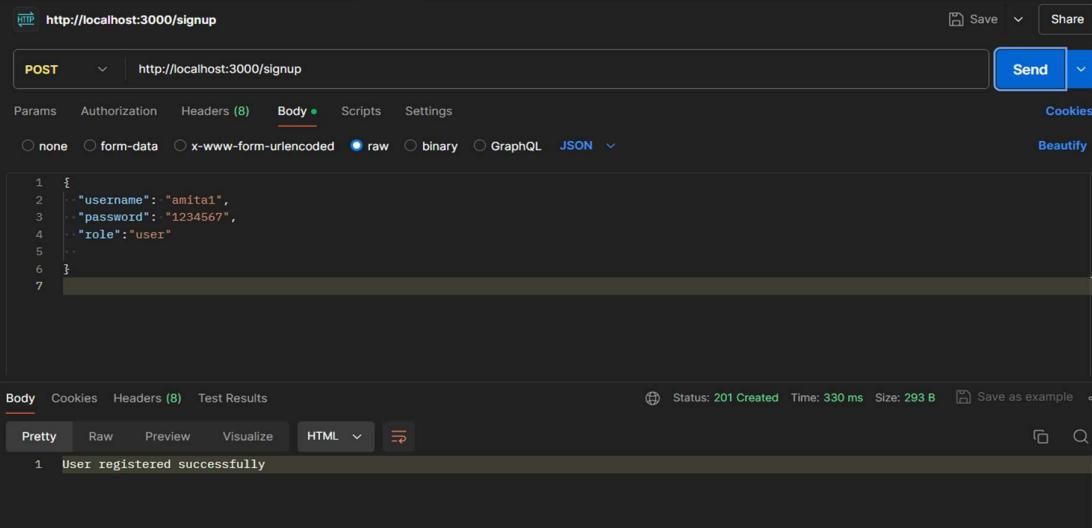
6 npm install axios: It is a promise-based HTTP client for the browser and Node.js. It allows us to make HTTP requests to fetch or save data to a web server. It is widely used for its simplicity and ease of use, supporting all modern browsers and Node.js.

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

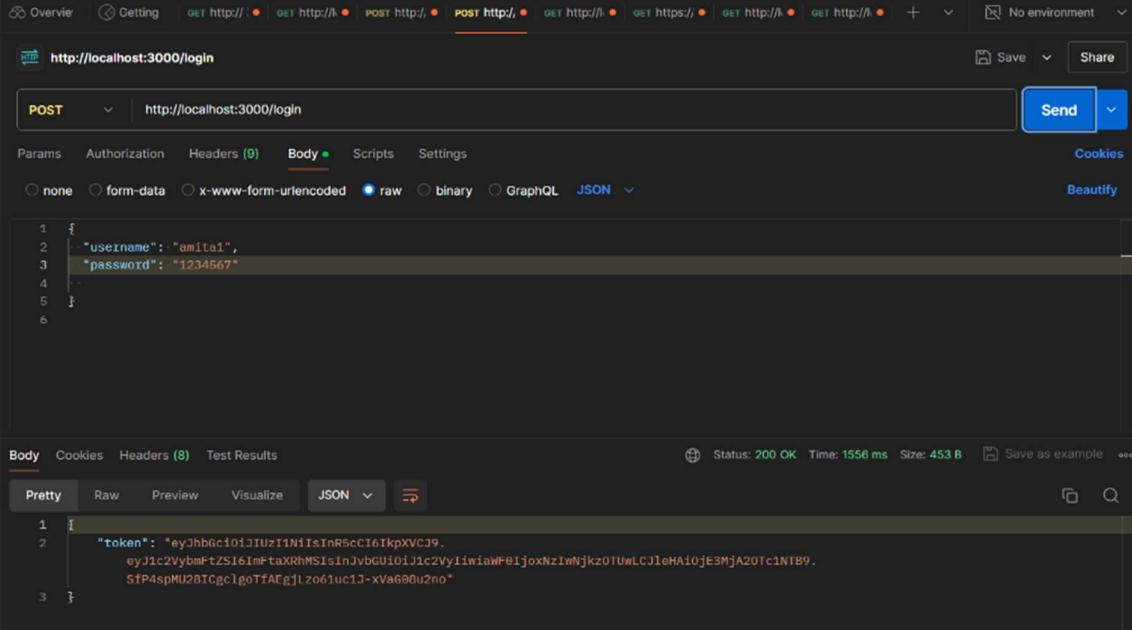


The screenshot shows a Postman request to `http://localhost:3000/signup`. The body is set to raw JSON with the following content:

```
1 {
2   "username": "amita1",
3   "password": "1234567",
4   "role": "user"
5 }
6
7 }
```

The response status is `201 Created`, and the body contains the message `User registered successfully`.

2. /LOGIN



The screenshot shows a Postman request to `http://localhost:3000/login`. The body is set to raw JSON with the following content:

```
1 {
2   "username": "amita1",
3   "password": "1234567"
4 }
5
6 }
```

The response status is `200 OK`, and the body contains a JSON object with a token key:

```
1 {
2   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
3   eyJlc2VybmtZS16ImFtaXRhMSIstnJvbGUiOiJlc2VyliwiawF0ijoxNzIwNjkzOTUwLCJlcl9hAiojE3MjA20Tc1NTB9.
4   SfP4spMU20TCgclgoTfAEgjlzog61uc1J-xVaG00U2no"
5 }
```

3./PROTECTED

The screenshot shows the Postman interface with a successful API call to `http://localhost:3000/protected`. The response details indicate a `200 OK` status, a response time of `31 ms`, and a response size of `287 B`. The response body is displayed as `Pretty` JSON, showing the message `Protected resource accessed`.

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

The screenshot shows the Postman interface with an unauthorized API call to `http://localhost:3000/admin`. The response details indicate a `200 OK` status, a response time of `56 ms`, and a response size of `331 B`. The response body is displayed as `Pretty` JSON, showing the message `your role is not allowed. your role is user and permitted role is admin`.

(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":

```
"eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9.eyJ1c2VybmcFtZSI6ImFtaXRhMSIsInJvbGUiOiJ1c2Vyl  
i  
wiaWF0IjoxNzlwNjkzOTUwLCIleHAiOjE3MjA2OTc1NTB9.SfP4spMU28ICgclgoTfAEgjLzo61uc1  
J-xVaG08u2no"
```

DECODED

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYLOAD: DATA

```
{  
  "username": "amitai1",  
  "role": "user",  
  "iat": 1720693950,  
  "exp": 1720697550  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + ". "  
  base64UrlEncode(payload),  
  your-256-bit-secret  
)  secret base64 encoded
```

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

The screenshot shows the MongoDB Atlas Data Services interface. The left sidebar has a 'Project 0' dropdown, followed by sections for Database, Services, Security, and Database Access. The 'Data Services' tab is selected. The main area shows 'AMITA'S ORG - 2024-07-10 > PROJECT 0' and the 'Clusters' section. It includes a search bar, a 'secure' button, and 'Edit Config' and '+ Create' buttons. Below is a 'Visualize Your Data' section with charts for R/W connections, network traffic (In 14.4 B/s, Out 229.2 B/s), and Data Size (134.7 MB / 512.0 MB). At the bottom, there's a table with columns: VERSION, REGION, CLUSTER TIER, TYPE, BACKUPS, LINKED APP SERVICES, ATLAS SQL, and ATLAS SEARCH. The table shows values: 7.0.12, AWS / Mumbai (ap-south-1), M0 Sandbox (General), Replica Set - 3 nodes, Inactive, None Linked, Connect, and Create Index.

Database Access

The screenshot shows the MongoDB Atlas Data Services interface. The left sidebar has a 'Project 0' dropdown, followed by sections for Database, Services, Security, and Database Access. The 'Data Services' tab is selected. The main area shows 'AMITA'S ORG - 2024-07-10 > PROJECT 0' and the 'Database Access' section. It includes tabs for 'Database Users' (selected) and 'Custom Roles'. A '+ ADD NEW DATABASE USER' button is at the top right. Below is a table with columns: User Name, Authentication Method, MongoDB Roles, Resources, and Actions. One row is shown: amita, SCRAM, atlasAdmin@admin, All Resources, with 'EDIT' and 'DELETE' buttons. At the bottom, there are 'Network Access' and 'Database Access' buttons.

Network Access

The screenshot shows the MongoDB Atlas Network Access interface. The left sidebar has a 'Project 0' section with 'Data Lake' selected, followed by 'SERVICES' (Device & Edge Sync, Triggers, Data API, Data Federation, Atlas Search, Stream Processing, Migration), 'SECURITY' (Quickstart, Backup, Database Access), and 'Network Access' (selected) with 'Advanced' below it. The main area is titled 'AMITA'S ORG - 2024-07-10 > PROJECT 0' and 'Network Access'. It has tabs for 'IP Access List' (selected), 'Peering', and 'Private Endpoint'. A button '+ ADD IP ADDRESS' is at the top right. A yellow box says 'You will only be able to connect to your cluster from the following list of IP Addresses:'. A table lists one entry: 'IP Address' 0.0.0.0/0 (includes your current IP address), 'Comment' amita, 'Status' Active, with 'EDIT' and 'DELETE' buttons. At the bottom is 'System Status: All Good'.

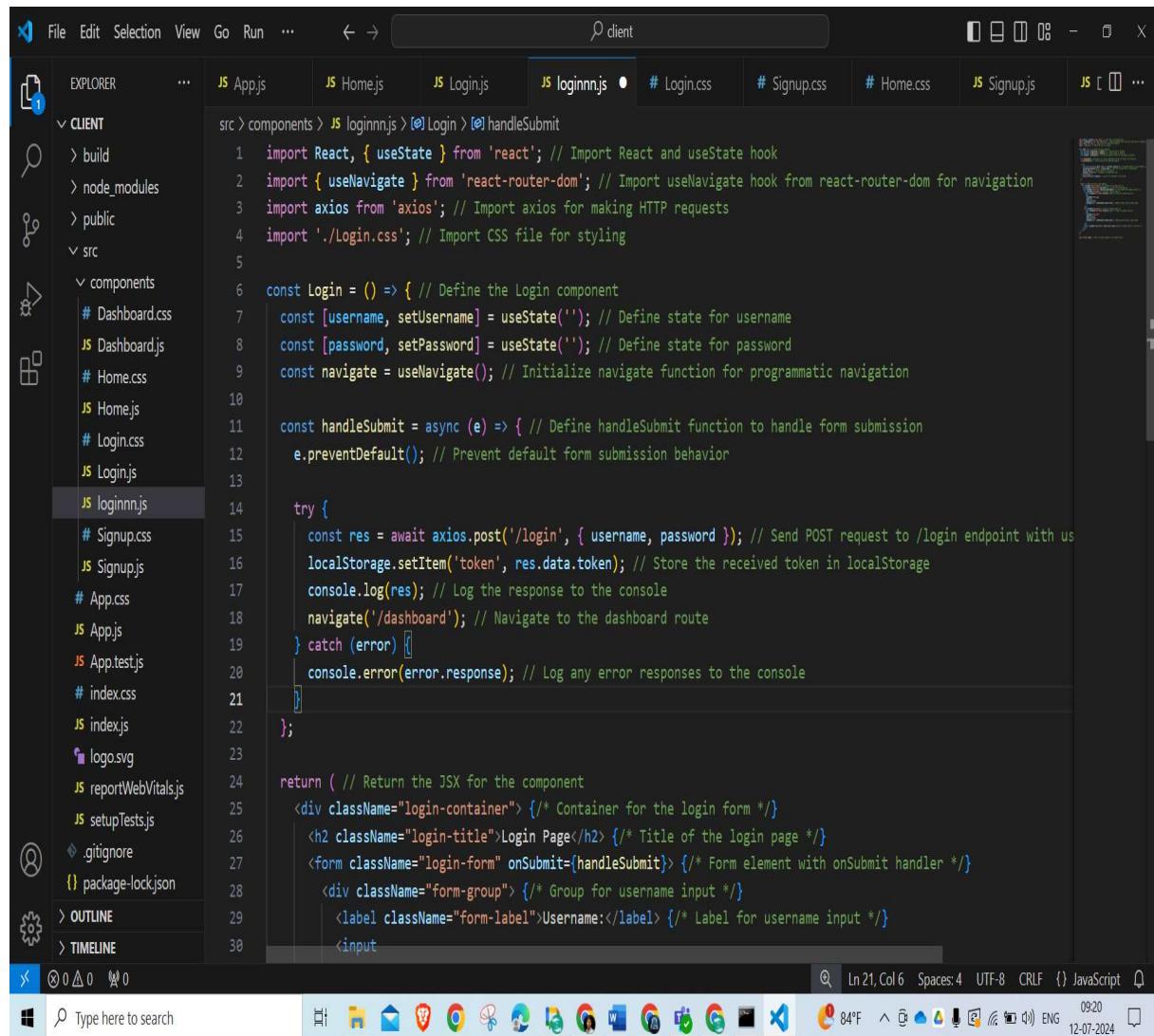
Data stored in the database

The screenshot shows the MongoDB Compass interface. The top navigation bar has 'Data Services' (selected), 'App Services', and 'Charts'. Below it are tabs for 'Overview', 'Real Time', 'Metrics', 'Collections' (selected), 'Atlas Search', 'Performance Advisor', 'Online Archive', and 'Cmd Line Tools'. A status bar shows 'DATABASES: 2 COLLECTIONS: 7'. Buttons for 'VISUALIZE YOUR DATA' and 'REFRESH' are on the right. The left sidebar shows databases 'sample_mflix' and 'test' with collection 'users' selected. The main area shows the 'test.users' collection with storage size 36KB, logical data size 57B, total documents 4, and index size 72KB. It has tabs for 'Find', 'Indexes', 'Schema Anti-Patterns', 'Aggregation', and 'Search Indexes'. A search bar 'Generate queries from natural language in Compass' has an 'INSERT DOCUMENT' button. A query builder 'Filter' shows '_id: ObjectId("669faeb05b7f11264cca61")', 'username: "amital"', 'password: "\$2a\$10\$leRllcRmKUqzhmy3vpr6.BkHsJCyzmy1cSFLijMjQC7BD5FIGrVW"', 'role: "user"', and '__v: 0'. Another document is partially visible below it.

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



The screenshot shows a code editor interface with a dark theme. On the left is the Explorer sidebar showing project files like App.js, Home.js, Login.js, and loginnn.js. The main editor area displays the content of the loginnn.js file. The file contains code for a Login component using React hooks (useState, useNavigate), axios for API calls, and CSS modules for styling. It handles form submission by sending a POST request to '/login' and navigating to the dashboard if successful. The status bar at the bottom shows file statistics (Ln 21, Col 6, Spaces: 4, CRLF) and a language setting for JavaScript.

```
File Edit Selection View Go Run ... ⏪ ⏩ client
EXPLORER ... JS App.js JS Home.js JS Login.js JS loginnn.js # Login.css # Signup.css # Home.css JS Signup.js JS ...
CLIENT
> build
> node_modules
> public
src
  < components
    # Dashboard.css
    JS Dashboard.js
    # Home.css
    JS Home.js
    # Login.css
    JS Login.js
    JS loginnn.js
    # Signup.css
    JS Signup.js
    # App.css
    JS App.js
    JS App.test.js
    # index.css
    JS index.js
    logo.svg
    JS reportWebVitals.js
    JS setupTests.js
    .gitignore
    package-lock.json
  < OUTLINE
  < TIMELINE
JS loginnn.js
# Login.css
# Signup.css
# Home.css
JS Signup.js
JS ...
src > components > JS loginnn.js > JS Login > JS handleSubmit
1 import React, { useState } from 'react'; // Import React and useState hook
2 import { useNavigate } from 'react-router-dom'; // Import useNavigate hook from react-router-dom for navigation
3 import axios from 'axios'; // Import axios for making HTTP requests
4 import './Login.css'; // Import CSS file for styling
5
6 const Login = () => { // Define the Login component
7   const [username, setUsername] = useState(''); // Define state for username
8   const [password, setPassword] = useState(''); // Define state for password
9   const navigate = useNavigate(); // Initialize navigate function for programmatic navigation
10
11   const handleSubmit = async (e) => { // Define handleSubmit function to handle form submission
12     e.preventDefault(); // Prevent default form submission behavior
13
14     try {
15       const res = await axios.post('/login', { username, password }); // Send POST request to /login endpoint with user credentials
16       localStorage.setItem('token', res.data.token); // Store the received token in localStorage
17       console.log(res); // Log the response to the console
18       navigate('/dashboard'); // Navigate to the dashboard route
19     } catch (error) {
20       console.error(error.response); // Log any error responses to the console
21     }
22   };
23
24   return ( // Return the JSX for the component
25     <div className="login-container"> /* Container for the login form */
26       <h2 className="login-title">Login Page</h2> /* Title of the login page */
27       <form className="login-form" onSubmit={handleSubmit}> /* Form element with onSubmit handler */
28         <div className="form-group"> /* Group for username input */
29           <label className="form-label">Username:</label> /* Label for username input */
30           <input
```

```
File Edit Selection View Go Run Terminal Help < > client
EXPLORER ... JS App.js JS Home.js JS Login.js JS loginnn.js • # Login.css # Signup.css # Home.css JS Signup.js JS Dashboard.js # Dashboard.css
CLIENT
> build
> node_modules
> public
src
> components
# Dashboard.css
JS Dashboard.js
# Home.css
JS Home.js
# Login.css
JS Login.js
JS loginnn.js
# Signup.css
JS Signup.js
# App.css
JS App.js
JS App.test.js
# index.css
JS index.js
logo.svg
JS reportWebVitals.js
JS setupTests.js
.githignore
package-lock.json
package.json
README.md
OUTLINE
TIMELINE
Type here to search
```

```
src > components > loginnn.js > Login
  6 const Login = () => { /* Define the Login component */
 11   const handleSubmit = async (e) => { // Define handleSubmit function to handle form submission
 12     e.preventDefault();
 13   }
 14   return ( // Return the JSX for the component
 15     <div className="login-container"> /* Container for the login form */
 16       <h2 className="login-title">Login Page</h2> /* Title of the login page */
 17       <form className="login-form" onSubmit={handleSubmit}> /* Form element with onSubmit handler */
 18         <div className="form-group"> /* Group for username input */
 19           <label className="form-label">Username:</label> /* Label for username input */
 20           <input
 21             className="form-input"
 22             type="text"
 23             value={username}
 24             onChange={(e) => setUsername(e.target.value)} // Update username state on input change
 25           />
 26         </div>
 27         <div className="form-group"> /* Group for password input */
 28           <label className="form-label">Password:</label> /* Label for password input */
 29           <input
 30             className="form-input"
 31             type="password"
 32             value={password}
 33             onChange={(e) => setPassword(e.target.value)} // Update password state on input change
 34           />
 35         </div>
 36         <button className="login-button" type="submit">Login</button> /* Submit button for the form */
 37       </form>
 38     </div>
 39   );
 40 }
 41
 42 export default Login; // Export the Login component as the default export
 43
 44
 45
 46
 47
 48
 49
 50
 51
 52
 53
```

APP.JS

The screenshot shows a code editor interface with a dark theme. On the left is a sidebar containing a tree view of the project structure:

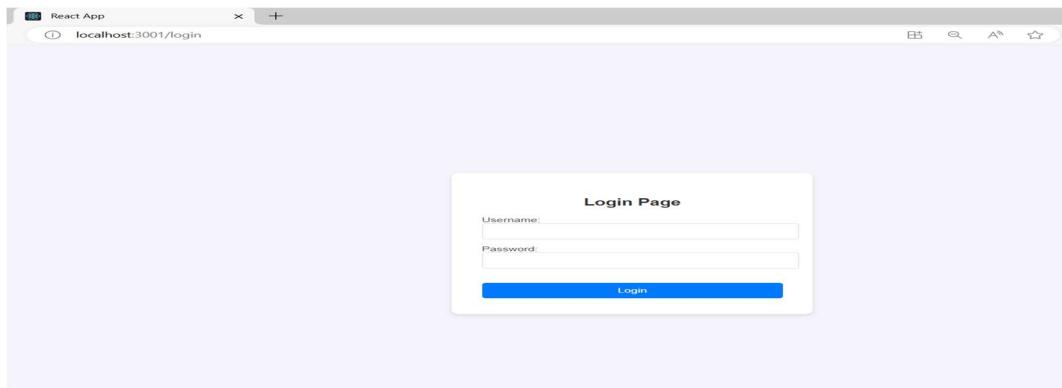
- CLIENT
- > build
- > node_modules
- > public
- > pt
- components
 - # Dashboard.css
 - # Dashboard.js
 - # Home.css
 - # Home.js
 - # Login.css
 - # Login.js
 - # Signup.css
 - # Signup.js
 - # App.css
 - App.js
- App.test.js
- index.css
- index.js
- logo.svg
- reportWebVitals.js
- setupTests.js
- .gitignore
- package-lock.json
- package.json
- README.md

The main editor area displays the content of the `App.js` file:

```
src > B App.js > 00 App
  1 import React from 'react';
  2 import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';
  3 import Home from './components/home';
  4 import Login from './components/login';
  5 import Signup from './components/signup';
  6 import Dashboard from './components/dashboard';
  7
  8 const App = () => {
  9   return (
 10     <Router>
 11       <div>
 12         <Routes>
 13           <Route path="/" element={<Home />} />
 14           <Route path="/login" element={<Login />} />
 15           <Route path="/signup" element={<Signup />} />
 16           <Route path="/dashboard" element={<Dashboard />} />
 17         </Routes>
 18       </div>
 19     </Router>
 20   );
 21 }
 22
 23 export default App;
 24
```

The code uses React Router to define routes for different components: Home, Login, Signup, and Dashboard. The Home component is the default route at the root. The Login and Signup components are available via links, and the Dashboard component is also available via a link.

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

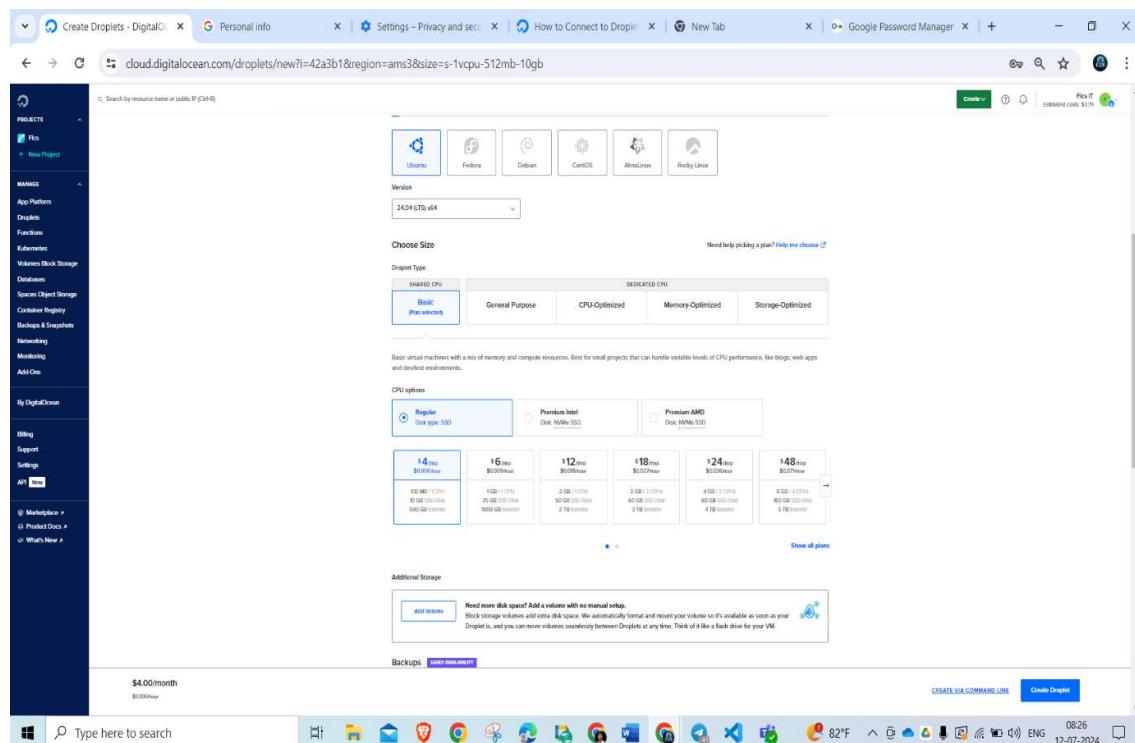
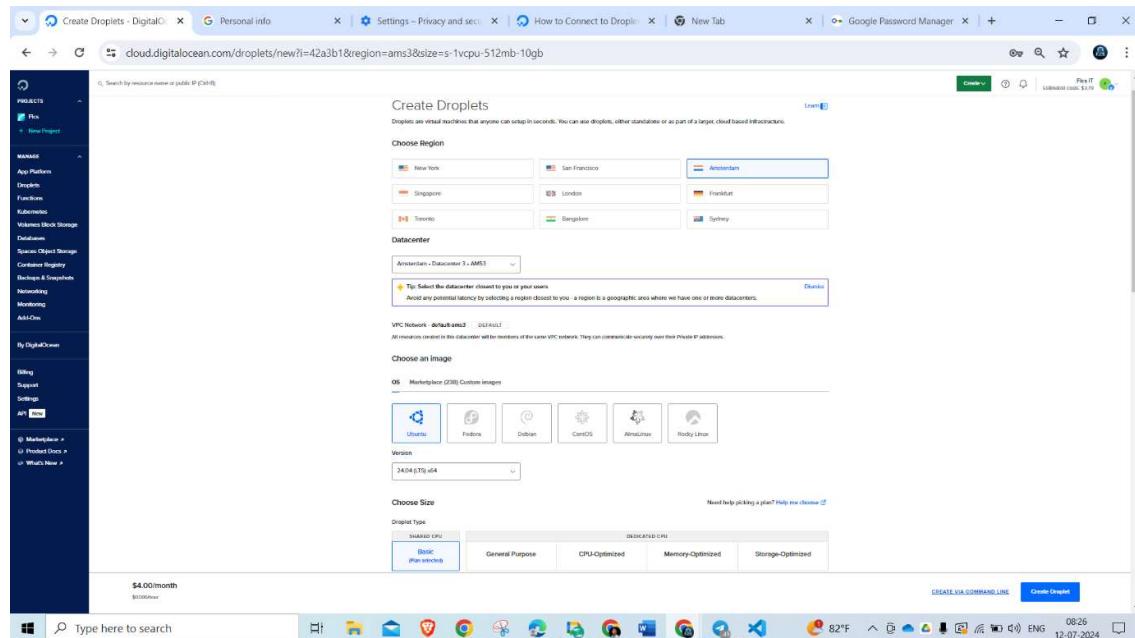
A screenshot of a web browser showing the GoDaddy domain control panel for the domain "unimesec.shop". The left sidebar has a "Domains" section with "unimesec.shop" selected. Under "DNS", the "DNS" tab is active, showing "DNS Records", "Forwarding", and "Name Servers". The "Name Servers" section lists "ns1.digitalocean.com", "ns2.digitalocean.com", and "ns3.digitalocean.com". A "Change nameservers" button is visible. The top navigation bar includes links for "Discover the news", "Service center", and other account options. The bottom taskbar shows system icons and the date/time.

DOMAIN NAME: unimesec.shop

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

ServerName: unimeSEC

CREATING DROPLETS:



DNS RECORDS:

The screenshot shows the DigitalOcean Control Panel interface. On the left, there's a sidebar with project management options like 'New Project', 'Manage', and 'API'. The main area is titled 'DNS records' and displays a table of records for the domain 'unimesec.shop'. The table columns are 'Type', 'Hostname', 'Value', and 'TTL (seconds)'. The records listed are:

Type	Hostname	Value	TTL (seconds)
A	www.unimesec.shop	directs to 174.138.3.188	3600
A	unimesec.shop	directs to 174.138.3.188	3600
NS	unimesec.shop	directs to ns1.digitalocean.com.	1800
NS	unimesec.shop	directs to ns2.digitalocean.com.	1800
NS	unimesec.shop	directs to ns3.digitalocean.com.	1800

IP ADDRESS OF THE SERVER (unimesec):

The screenshot shows the DigitalOcean Control Panel interface for the project 'flics'. The sidebar includes options like 'New Project', 'Manage', and 'API'. The main area displays the 'Resources' tab for the project. It shows two droplets: 'unimesec' (IP: 64.227.67.178) and 'flicserver' (IP: 157.245.97.166). There are also two domains listed: 'unimesec.shop' and 'flics.in'. At the bottom, there are links for 'Create something new' (Managed Database, Spaces), 'Learn more', and 'Product Docs'.

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.

The screenshot shows a terminal session on a DigitalOcean Droplet. The user has run several commands to set up a Node.js application and its deployment using PM2. The session includes:

- Initial command output showing Node.js and npm installations.
- A decorative ASCII art banner.
- PM2 configuration and usage instructions.
- PM2 spawning a process for the application.
- PM2 successfully demuxing the application.
- PM2 starting the application in fork mode.
- A table showing the running process details:

id	name	mode	status	cpu	memory
0	server	fork	0 online	0%	43.5mb

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:** Enabling 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:

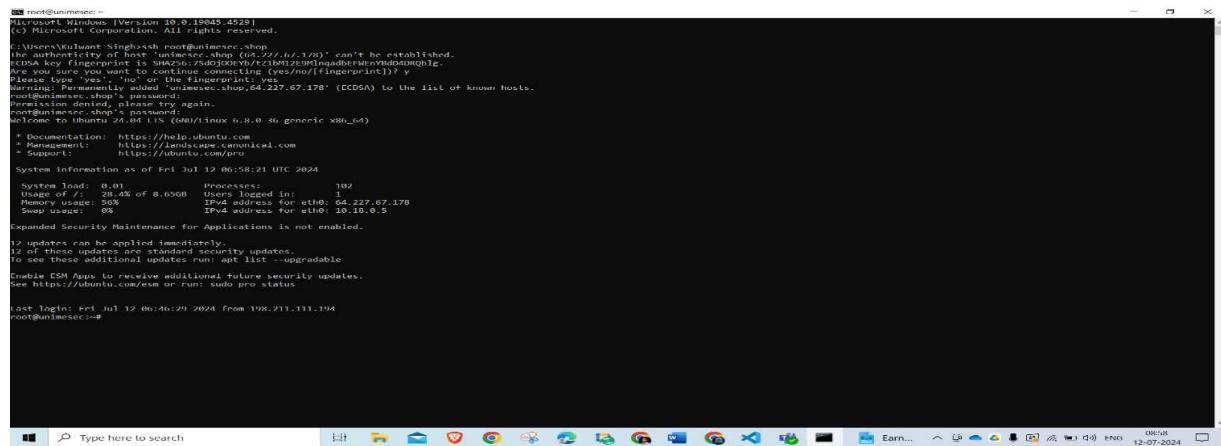
ssh root@unimesec.shop: root password: Italy@#2024SEC

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:



```
Microsoft Windows [Version 10.0.19045.4529]
(c) Microsoft Corporation. All rights reserved.

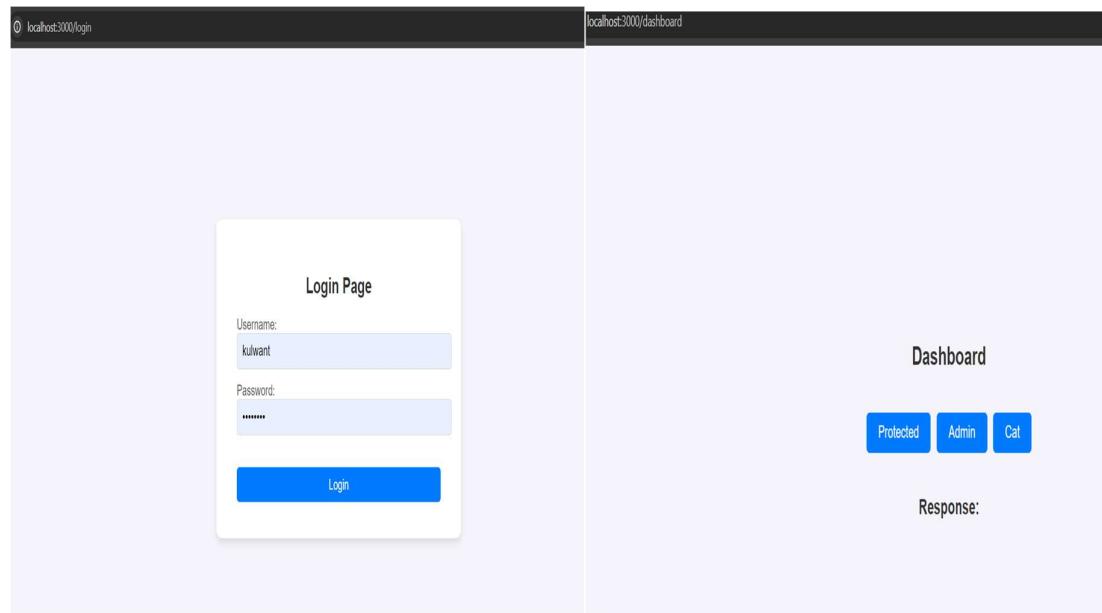
C:\Users\Kulwant Singh&ash root@unimsec:~shop
ssh: connect to host 64.227.67.178 port 22: Connection refused.
ECDSA key fingerprint is SHA256:5d0j00tY/tZjbm129mlqabherMny6d04mQblg.
Please type 'yes', 'no' or the fingerprint again to continue connecting (yes/no/[fingerprint])? y
Warning: Permanently added 'unimsec.shop,64.227.67.178' (ECDSA) to the list of known hosts.

Permission denied, please try again.

root@unimsec:~shop
Last login: Fri Jul 12 06:46:29 2024 from 198.211.111.194
root@unimsec:~
```

OUTPUTS

WHILE RUNNING THE SERVER AT LOCAL MACHINE USING PORT 3000



localhost:3000/login localhost:3000/dashboard

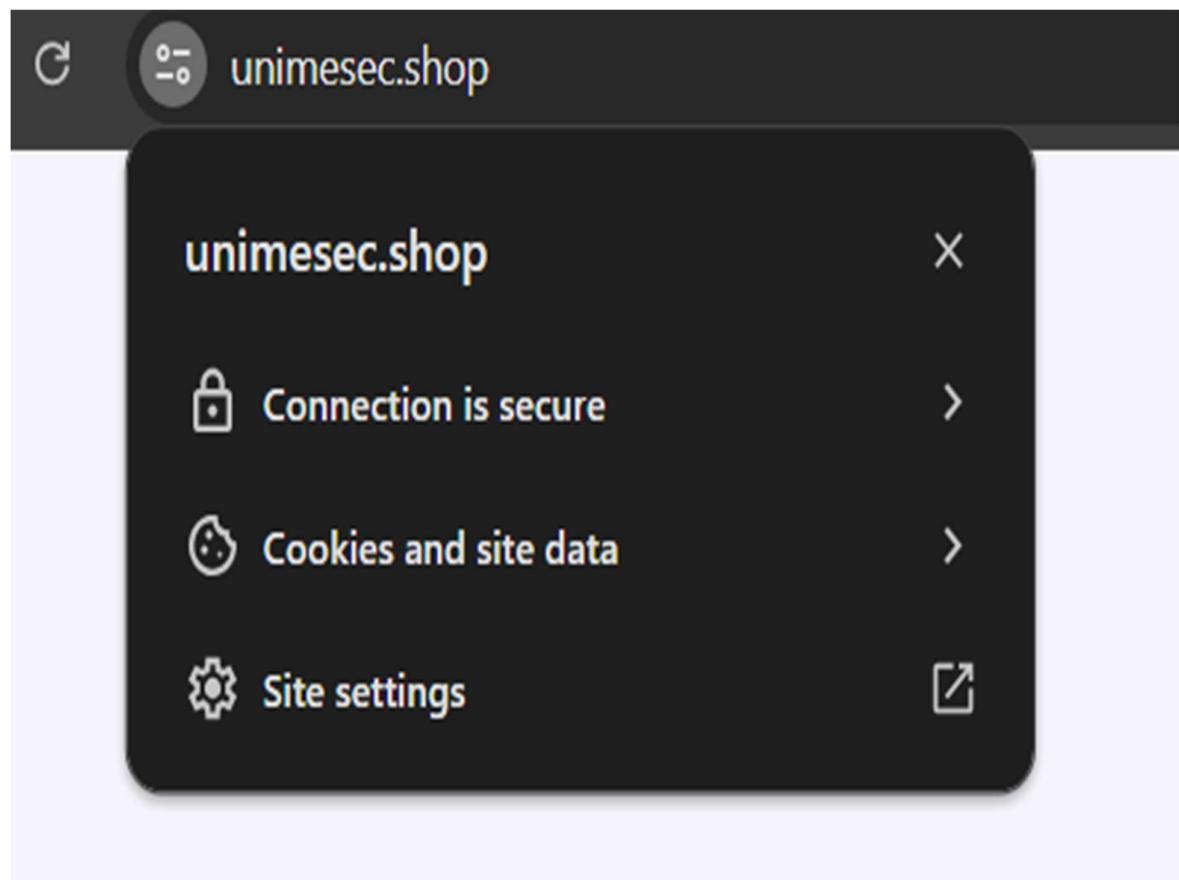
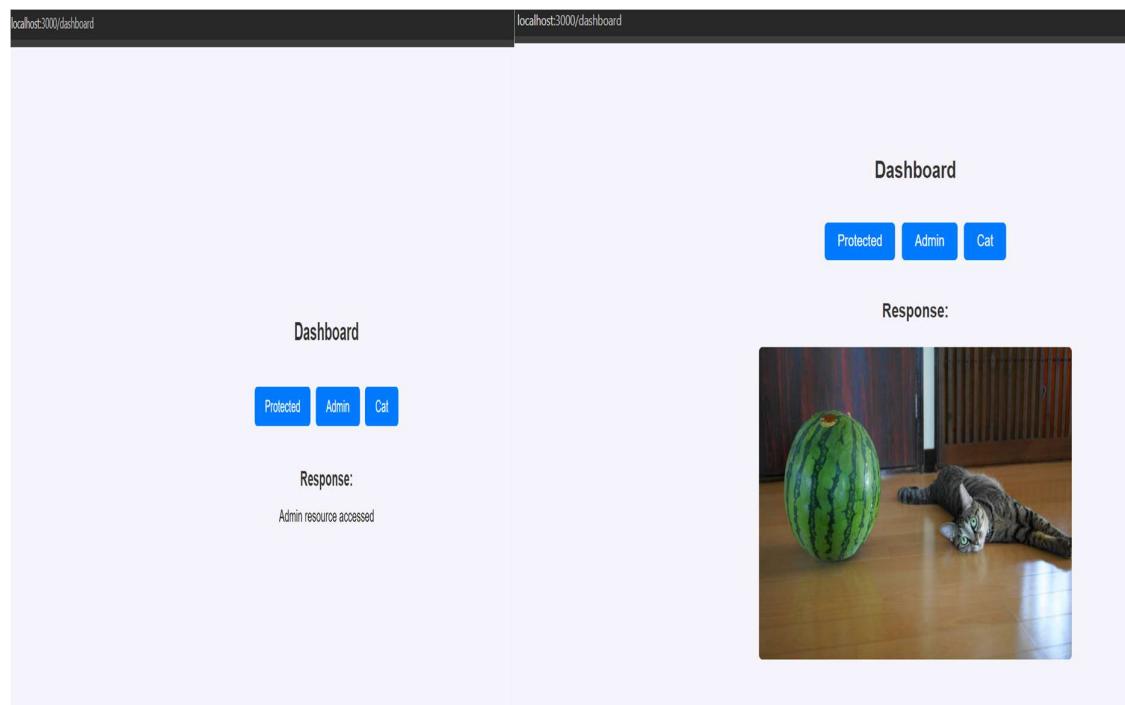
Login Page

Username:

Password:

Protected Admin Cat

Response:

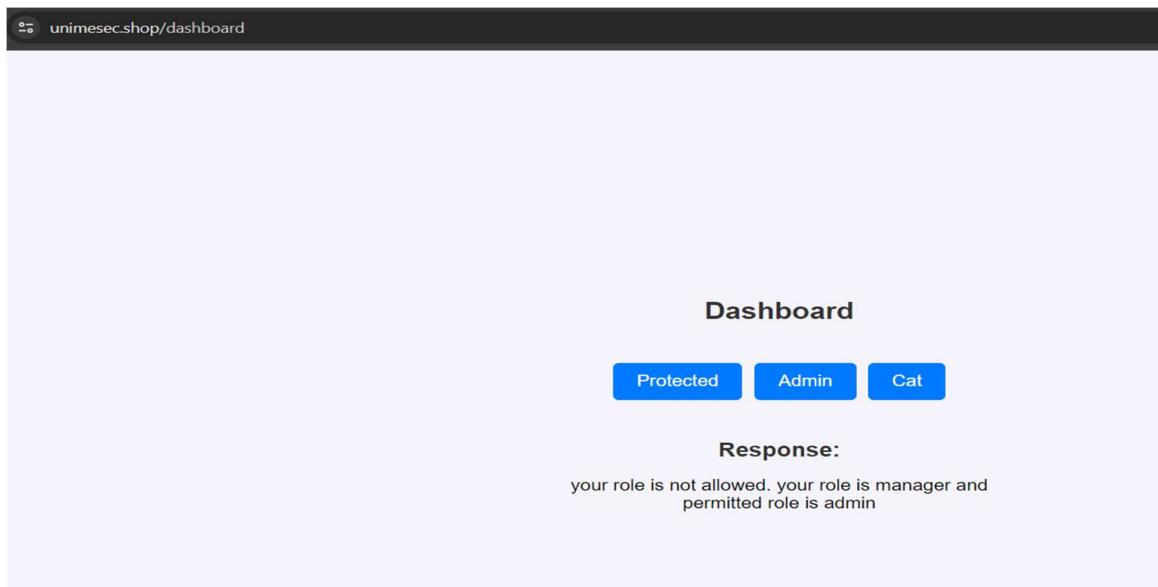
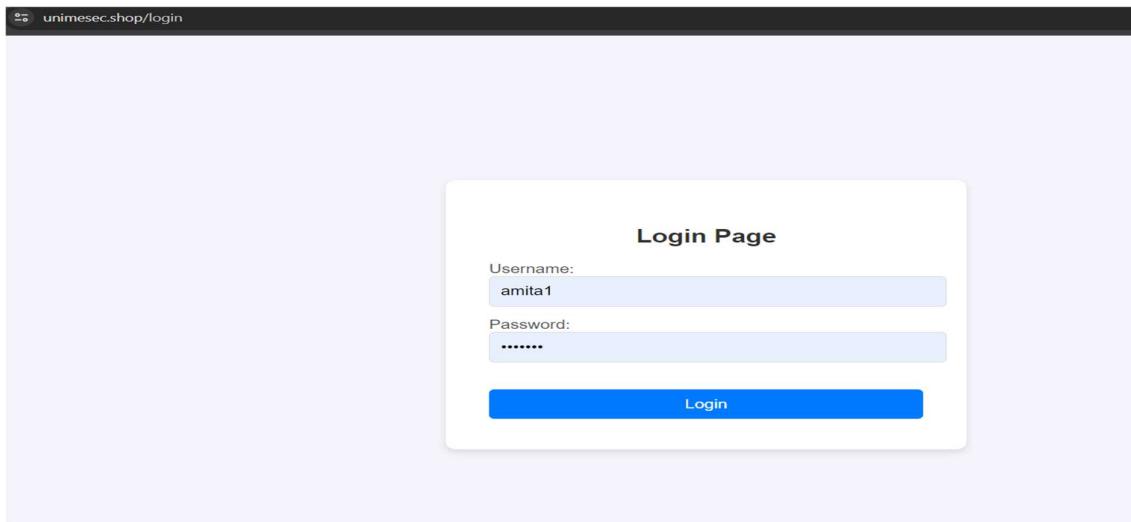


WHILE RUNNING THE SERVER AT UNIMESEC WITH IP (64.227.67.178) WITH OUR DOMAIN [WWW.UNIMESEC.SHOP](https://www.unimesec.shop) AT PORT 443

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>

The image displays two side-by-side screenshots of the PageSpeed Insights tool analyzing the website <https://www.unimesec.shop/>.

Left Screenshot (Mobile View):

- Performance Score:** 98
- Metrics:** Accessibility (91), Best Practices (100), SEO (100)
- Diagnose Performance Issues:** A large green circle indicates a high performance score.
- Details:** Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.
- Legend:** 0-49 (red), 50-89 (orange), 90-100 (green).
- METRICS:** Expand view.

Right Screenshot (Desktop View):

- Summary Metrics:** Total Blocking Time: 30 ms, Cumulative Layout Shift: 0, Speed Index: 0.9 s.
- Environment:** Captured at Jul 11, 2024, 3:38 PM, Emulated Moto G Power with Lighthouse 12.0.0, Single page session, Using Headless Chromium.
- Treemap:** Shows the size of various resources on the page.
- Diagnostics:**
 - Enable text compression - Potential savings of 134 KIB
 - Reduce unused JavaScript - Potential savings of 101 KIB
 - Eliminate render-blocking resources - Potential savings of 150 ms
 - JavaScript execution time - 0.1 s
 - Minimizes main-thread work - 0.2 s
 - Avoid long main-thread tasks - 1 long task found
 - Initial server response time was short - Root document took 10 ms
 - Avoids enormous network payloads - Total size was 202 KIB

11/07/2024, 15:39

PageSpeed Insights

[Copy Link](#) [Docs](#)

Largest Contentful Paint element — 2,340 ms

PASSED AUDITS (25)

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)

These items address areas which an automated testing tool cannot cover. Learn more in our guide on [conducting an accessibility review](#).

PASSED AUDITS (9)

11/07/2024, 15:39

PageSpeed Insights

[Copy Link](#) [Docs](#)

Best Practices

TRUST AND SAFETY

Ensure CSP is effective against XSS attacks

GENERAL

Detected JavaScript libraries

PASSED AUDITS (15)

NOT APPLICABLE (1)

100

https://pagespeed.web.dev/analysis/http://www.unresec.org/pic/216/w?utm_factor=mobile

35

https://pagespeed.web.dev/analysis/http://www.unresec.org/pic/216/w?utm_factor=mobile

45

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.