### \*\* Introduction to Authentication and Authorization\*\*

### Authentication vs. Authorization

- \*\*Authentication\*\*: This is the process of confirming a user's identity, typically done through credentials like usernames and passwords. It answers the question, "Who are you?"

- \*\*Authorization\*\*: After authentication, this process determines the user's permissions and what they are allowed to do within the system. It answers the question, "What are you allowed to do?"

- \*\*Why They Matter\*\*: Authentication ensures that users are who they claim to be, while authorization ensures they have the necessary permissions, protecting sensitive data and maintaining a secure user experience.

### JWT (JSON Web Tokens)

- \*\*Introduction to JWT\*\*: JWT is a compact, URL-safe token used for securely transmitting information between parties as a JSON object, often used for authentication and information exchange.

- \*\*Structure of a JWT\*\*: A JWT consists of three parts: the \*\*Header\*\* (algorithm and token type), the \*\*Payload\*\* (claims or data), and the \*\*Signature\*\* (ensures the token hasn't been tampered with).

- \*\*How JWT is used in Authentication\*\*: After a user logs in, the server creates and signs a JWT, which is then sent to the client. The client stores the token and includes it in requests to authenticate and access protected resources.

- \*\*Installation and Setup\*\*

- \*\*Express.js Setup:\*\* Installing Express in a Node.js environment.

- \*\*Installing Required Packages:\*\*

- `jsonwebtoken` for handling JWTs.

### JSON Web Token (jsonwebtoken) for Handling JWTs

`jsonwebtoken` is a library used in Node.js to create, sign, and verify JSON Web Tokens (JWTs). JWTs are used for securely transmitting information between a client and a server as a JSON object. They are commonly used in authentication and authorization processes, where the token carries encoded user data that the server can verify without storing session data.

### bcrypt for Password Hashing

`bcrypt` is a password-hashing library that provides a way to securely hash passwords in Node.js applications. It uses a cryptographic algorithm that includes a salt (a random value) to protect against dictionary attacks and rainbow table attacks. By storing only the hashed password, `bcrypt` ensures that even if the database is compromised, the original passwords remain secure.

### Mongoose for MongoDB Connection

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a schema-based solution to model application data, enabling developers to define the structure, validation, and relationships of the data within MongoDB. Mongoose simplifies database interactions by offering built-in functions for CRUD operations, middleware, and query building, making it easier to work with MongoDB in a Node.js environment.

\*\*Example Code:\*\*

```bash

npm install express mongoose jsonwebtoken bcrypt

```

#### \*\*Implementing Authentication and Authorization\*\*

- \*\*Setting Up MongoDB Models\*\*

- Create a `User` model using Mongoose.

- Fields: `username`, `email`, `password`, `role`.

- Hashing Passwords: Use `bcrypt` to hash passwords before saving to the database.

\*\*Example Code:\*\*

```javascript

const mongoose = require('mongoose');

const bcrypt = require('bcrypt');

const userSchema = new mongoose.Schema({

username: { type: String, required: true },

email: { type: String, required: true, unique: true },

password: { type: String, required: true },

role: { type: String, default: 'user' },

});

userSchema.pre('save', async function (next) {

if (this.isModified('password')) {

this.password = await bcrypt.hash(this.password, 10);

}

next();

});

const User = mongoose.model('User', userSchema);

```

- \*\*User Registration and Login\*\*

- \*\*Registration Endpoint:\*\* Save new users to the database after hashing their passwords.

- \*\*Login Endpoint:\*\* Verify user credentials and issue a JWT upon successful login.

\*\*Example Code:\*\*

```javascript

const express = require('express');

const jwt = require('jsonwebtoken');

const User = require('./models/User');

const app = express();

app.use(express.json());

app.post('/register', async (req, res) => {

try {

const { username, email, password } = req.body;

const user = new User({ username, email, password });

await user.save();

res.status(201).json({ message: 'User registered successfully' });

} catch (error) {

res.status(400).json({ error: 'User registration failed' });

}

});

app.post('/login', async (req, res) => {

try {

const { email, password } = req.body;

const user = await User.findOne({ email });

if (!user) return res.status(400).json({ error: 'Invalid email or password' });

const isMatch = await bcrypt.compare(password, user.password);

if (!isMatch) return res.status(400).json({ error: 'Invalid email or password' });

const token = jwt.sign({ userId: user.\_id, role: user.role }, 'secret\_key', { expiresIn: '1h' });

res.json({ token });

} catch (error) {

res.status(500).json({ error: 'Login failed' });

}

});

app.listen(3000, () => console.log('Server running on port 3000'));

```

#### \*\* Handling Authorization and Exceptions\*\*

### Protecting Routes with Middleware

1. \*\*Create Middleware to Check for Valid JWT Tokens:\*\* Develop a middleware function that inspects request headers for a JWT token and verifies its validity, ensuring that only authenticated users can access protected routes.

2. \*\*Verify the User's Role to Restrict Access:\*\* Extend the middleware to check the decoded JWT for user roles or permissions, and restrict access based on the user's role, such as admin or regular user, to specific routes.

\*\*Example Code:\*\*

```javascript

const authMiddleware = (req, res, next) => {

const token = req.headers['authorization'];

if (!token) return res.status(401).json({ error: 'Access denied' });

try {

const decoded = jwt.verify(token, 'secret\_key');

req.user = decoded;

next();

} catch (error) {

res.status(400).json({ error: 'Invalid token' });

}

};

const adminMiddleware = (req, res, next) => {

if (req.user.role !== 'admin') {

return res.status(403).json({ error: 'Forbidden' });

}

next();

};

```

- \*\*Example Protected Routes\*\*

- Implement routes that only authenticated users can access.

- Create admin-only routes.

\*\*Example Code:\*\*

```javascript

app.get('/dashboard', authMiddleware, (req, res) => {

res.json({ message: 'Welcome to the dashboard' });

});

app.get('/admin', authMiddleware, adminMiddleware, (req, res) => {

res.json({ message: 'Welcome to the admin panel' });

});

```

### Handling Exceptions

1. \*\*Implement Proper Error Handling in Middleware and Routes:\*\* Incorporate try-catch blocks in your route handlers and middleware to catch and handle errors gracefully, providing meaningful error responses.

2. \*\*Use Express's Error-Handling Middleware to Manage Exceptions Globally:\*\* Utilize Express's built-in error-handling middleware to catch unhandled errors and exceptions throughout the application, allowing for centralized error management and logging.

\*\*Example Code:\*\*

```javascript

app.use((err, req, res, next) => {

console.error(err.stack);

res.status(500).json({ error: 'Something went wrong!' });

});

```