Week 6 Lab: Password Hashing, JWT Authentication & Role-Based Access Control

Objective: Secure the e-hailing system with password hashing, JWT tokens, and role-based access. Test authentication flows using Postman to simulate client-side interactions.

Lab Overview

Key Topics

1. **Password Hashing**: Securely store passwords using `bcrypt`.
2. **JWT Authentication**: Generate tokens for authenticated users.
3. **Role-Based Access Control (RBAC)**: Restrict endpoints by role (customer, driver, admin).
4. **Postman Testing**: Simulate client requests with JWT tokens.

Deliverables

1. Updated authentication APIs with JWT and password hashing.
2. Postman collection with tests for registration, login, and role-restricted endpoints.
3. Lab report with answers to security questions and Postman test results.

Lab Procedures

Part 1: Password Hashing

1. **Install Dependencies**

   npm install bcrypt jsonwebtoken

2. **Modify Registration Endpoint**

   Update `POST /users` to hash passwords:

```
const bcrypt = require('bcrypt');
const saltRounds = 10;

app.post('/users', async (req, res) => {
  try {
    const hashedPassword = await bcrypt.hash(req.body.password, saltRounds);
    const user = { ...req.body, password: hashedPassword };
    await db.collection('users').insertOne(user);
    res.status(201).json({ message: "User created" });
  } catch (err) {
    res.status(400).json({ error: "Registration failed" });
  }
});
```

Part 2: JWT Authentication

1. **Create a file called `.env` and add the following content:**

   ```
   JWT_SECRET=your_secure_key_here
   JWT_EXPIRES_IN=1h
   ```

2. **Update Login Endpoint**

   Return a JWT token on successful login:

```
const jwt = require('jsonwebtoken');

app.post('/auth/login', async (req, res) => {
  const user = await db.collection('users').findOne({ email: req.body.email
 });
  if (!user || !(await bcrypt.compare(req.body.password, user.password))) {
    return res.status(401).json({ error: "Invalid credentials" });
  }
  const token = jwt.sign(
    { userId: user._id, role: user.role },
    process.env.JWT_SECRET,
    { expiresIn: process.env.JWT_EXPIRES_IN }
  );
  res.status(200).json({ token }); // Return token to client
});
```

Part 3: Role-Based Access Control (RBAC)

1. **Create Authentication Middleware**

```
const jwt = require('jsonwebtoken');

const authenticate = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];

  if (!token) return res.status(401).json({ error: "Unauthorized" });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).json({ error: "Invalid token" });
  }
};

const authorize = (roles) => (req, res, next) => {
  if (!roles.includes(req.user.role))
     return res.status(403).json({ error: "Forbidden" });
  next();
};
```

2. **Protect Admin Endpoint**

   Restrict `DELETE /admin/users/:id` to admins:

```
app.delete('/admin/users/:id', authenticate, authorize(['admin']), async (req,
 res) => {
   console.log("admin only");
   res.status(200).send("admin access");
});
```

Part 4: Client-Side Testing with Postman

Step 1: User Registration

1. Create a **POST** request to `http://localhost:3000/users`.

2. Set headers:
    - **Content-Type**: `application/json`
3. Add request body (raw JSON):
```
{
  "email": "admin@example.com",
  "password": "securePassword123",
  "role": "admin"
}
```
4. Send the request. A `201 Created` response confirms registration.

Step 2: User Login

1. Create a **POST** request to `http://localhost:3000/auth/login`.
2. Set headers:
    - **Content-Type**: `application/json`
3. Add request body (raw JSON):
```
{
  "email": "admin@example.com",
  "password": "securePassword123"
}
```
4. Send the request. Copy the JWT token from the response similar to follow:
```
{ "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..." }
```

Step 3: Access Protected Endpoint (Admin)

1. Create a **DELETE** request to `http://localhost:3000/admin/users/123`.
2. Set headers:
    - **Authorization**: `Bearer <paste-token-here>`
    - Example: `Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...`
3. Send the request:
    - If the token is valid and the user is an admin, you'll get a `204 No Content` response.
    - If unauthorized, you'll get `401 Unauthorized` or `403 Forbidden`.

Postman Testing Workflow

Step	Endpoint	Method	Headers	Body
1	`/users`	POST	`Content-Type: application/json`	User credentials (email, password)
2	`/auth/login`	POST	`Content-Type: application/json`	User credentials
3	`/admin/users/{id}`	DELETE	`Authorization: Bearer <token>`	None

Lab Questions

Answer by testing your API in Postman.

1. **Token Usage**:
   - What happens if you omit the `Authorization` header when accessing `/admin/users/{id}`?
   - What error occurs if you use an expired token?
   - Paste the token generated to https://jwt.io, and discuss the content
2. **Role Restrictions**:
   - If a `customer`-role user tries to access `/admin/users/{id}`, what status code is returned?
   - How would you modify the middleware to allow both `admin` and `driver` roles to access an endpoint?
3. **Security**:
   - Why is the JWT token sent in the `Authorization` header instead of the request body?
   - How does password hashing protect user data in a breach?
4. **Postman Testing**:

    - What is the purpose of the `Bearer` keyword in the `Authorization` header?
    - How would you test a scenario where a user enters an incorrect password?

Submission Requirements

1. **GitHub Repository**:
    - Code for JWT authentication, password hashing, and RBAC middleware.
2. **Postman Collection**:
    - Exported collection with:
        - Registration request
        - Login request (save token as a variable)
        - Admin endpoint request (using token)
3. **Lab Report**:
    - Answers to questions.
    - Screenshots of Postman tests (successful and failed auth).