Authentication and Authorization in Express

1. Problem Statement

Modern web applications must safeguard sensitive data and control user access to various resources. As your application grows, ensuring only authenticated and authorized users can perform certain actions becomes a core security requirement.

The challenge:
How do you implement robust authentication and authorization in an Express.js application to verify user identities, manage permissions, and protect resources from unauthorized access?

---

2. Learning Objectives

By the end of this lesson, you will be able to:

- Understand the concepts of authentication and authorization in Express.js.
- Differentiate between authentication and authorization.
- Recognize common mechanisms and best practices for implementing both.
- Apply authentication and authorization using JWTs and middleware in an Express.js app.
- Identify typical pitfalls and how to avoid them.

---

3. Concept Introduction with Analogy

Analogy: The Coffee Shop

Think of your web application as a coffee shop:

- **Authentication** is like checking a customer's ID at the door to confirm who they are.
- **Authorization** is like checking if that customer is allowed behind the counter or only permitted to sit in the public area.

Just as a coffee shop grants different access to baristas and managers, your application must distinguish between users and their permissions.

---

4. Technical Deep Dive

What is Authentication?

Authentication is the process of verifying the identity of a user or entity. In web apps, this typically involves checking credentials (like username and password) or verifying a token. Common authentication methods include:

- Username and password
- Token-based (e.g., JWT)

- Single Sign-On (SSO)

Authentication process:

1. User submits credentials.
2. Server verifies credentials.
3. If valid, server issues a session or token.
4. Client stores the token/session for subsequent requests.
5. Client includes the token/session in future requests for access2.

What is Authorization?

Authorization determines what actions or resources an authenticated user can access. It enforces access control based on roles, permissions, or policies.

Common authorization models:

- Role-Based Access Control (RBAC)
- Attribute-Based Access Control (ABAC)
- Policy-Based Access Control

Authorization process:

1. Server receives a request from an authenticated user.
2. Server checks the user's roles/permissions.
3. Server allows or denies access to the requested resource2.

---

5. Step-by-Step Data Modeling & Code Walkthrough

Example: Implementing JWT Authentication & Authorization in Express

Project Setup

```
npm init -y
npm install express jsonwebtoken bcryptjs mongoose
```

Basic Server Setup

```
const express = require('express');
const app = express();
app.use(express.json());

const authRoute = require('./routes/index');
app.use('/api', authRoute);

app.listen(3000, () => console.log("Server running on localhost:3000"));
```

Defining Routes (`routes/index.js`)

```
const router = require('express').Router();
const userController = require('../controllers/user');

// Register and Login
router.post('/register', userController.register);
router.post('/login', userController.login);

module.exports = router;
```

User Registration (`controllers/user.js`)

```
exports.register = async (req, res) => {
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(req.body.password, salt);

    let user = new User({
        email: req.body.email,
        name: req.body.name,
        password: hashedPassword,
        user_type_id: req.body.user_type_id
    });

    user.save((err, registeredUser) => {
        if (err) return res.status(500).send(err);
        let payload = { id: registeredUser._id, user_type_id: req.body.user_type_id || 0 };
        const token = jwt.sign(payload, config.TOKEN_SECRET);
        res.status(200).send({ token });
    });
};
```

User Login (`controllers/user.js`)

```
exports.login = async (req, res) => {
    User.findOne({ email: req.body.email }, async (err, user) => {
        if (err || !user) return res.status(401).send('Invalid email');
        const validPass = await bcrypt.compare(req.body.password, user.password);
        if (!validPass) return res.status(401).send("Email or Password is wrong");
        let payload = { id: user._id, user_type_id: user.user_type_id };
        const token = jwt.sign(payload, config.TOKEN_SECRET);
        res.status(200).header("auth-token", token).send({ token });
    });
};
```

JWT Middleware (`middleware/auth.js`)

```
exports.verifyUserToken = (req, res, next) => {
    let token = req.headers.authorization;
    if (!token) return res.status(401).send("Access Denied / Unauthorized request");
    try {
        token = token.split(' ')[1];
        if (!token) return res.status(401).send('Unauthorized request');
        let verifiedUser = jwt.verify(token, config.TOKEN_SECRET);
        if (!verifiedUser) return res.status(401).send('Unauthorized request');
        req.user = verifiedUser;
        next();
```

```
    } catch (error) {
        res.status(400).send("Invalid Token");
    }
};

exports.IsUser = (req, res, next) => {
    if (req.user.user_type_id === 0) return next();
    return res.status(401).send("Unauthorized!");
};

exports.IsAdmin = (req, res, next) => {
    if (req.user.user_type_id === 1) return next();
    return res.status(401).send("Unauthorized!");
};
```

Protecting Routes (`routes/index.js`)

```
const { verifyUserToken, IsUser, IsAdmin } = require('../middleware/auth');

// Only regular users
router.get('/events', verifyUserToken, IsUser, userController.userEvent);

// Only admins
router.get('/special', verifyUserToken, IsAdmin, userController.adminEvent);
```

---

6. Common Pitfalls & Best Practices

- **Never store sensitive data in JWT payloads**—JWTs are only base64-encoded, not encrypted.
- **Always validate and sanitize user input** to prevent injection attacks.
- **Use HTTPS** to protect tokens in transit.
- **Rotate and securely store secret keys** for signing JWTs.
- **Apply the principle of least privilege**—grant users only the permissions they need.
- **Log and monitor authentication/authorization failures** for security auditing.

---

7. Quick Recap & Key Takeaways

- **Authentication** verifies user identity; **authorization** controls user actions and access.
- Both are essential for securing Express.js apps2.
- JWTs provide a stateless, scalable way to handle authentication and authorization.
- Middleware allows modular, reusable security logic in Express.
- Proper implementation protects sensitive data, enforces business rules, and enhances compliance.

By mastering authentication and authorization in Express, you can build secure, robust web applications that scale confidently as your user base grows2.