Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

AUTHENTICATION & SECURITY IN EXPRESS.JS (WEEKS 5-6)

USING JWT FOR TOKEN-BASED AUTHENTICATION IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO JWT (JSON WEB TOKEN)

1.1 What is JWT?

JSON Web Token (JWT) is a **compact, URL-safe token** used to **securely transfer information** between parties. JWT is widely used for **authentication and authorization** in web applications.

- **Why Use JWT?**

✓ Securely **transmits user identity** without storing session data.
✓ Works with **stateless authentication**, reducing server load.
✓ Used in **REST APIs, microservices, and authentication flows**.

- **JWT Structure (Three Parts Separated by Dots .):**

Part	Purpose	Example
Header	Specifies algorithm & token type	{ "alg": "HS256", "typ": "JWT" }

Payload	Contains user data (claims)	{ "userId": 123, "role": "admin" }
Signature	Ensures token integrity	HMACSHA256(header + payload, secret)

📌 **Example JWT Token:**

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEyMywicm9sZSI6ImFkbWluIno.sDfxIuy98kdfj9382JDJFJX

✓ **Securely encodes user information** that can be verified.

CHAPTER 2: SETTING UP JWT AUTHENTICATION IN EXPRESS.JS

2.1 Installing Required Packages

📌 **Install jsonwebtoken and bcrypt.js for authentication:**

npm install express jsonwebtoken bcryptjs dotenv

✓ jsonwebtoken ⟶ Generates and verifies JWT tokens.
✓ bcryptjs ⟶ Hashes and compares passwords.
✓ dotenv ⟶ Stores sensitive environment variables.

2.2 Creating an Express Server with JWT Authentication

📌 **Create server.js and Set Up Express.js:**

```
const express = require("express");

const jwt = require("jsonwebtoken");

const bcrypt = require("bcryptjs");

const dotenv = require("dotenv");
```

```
dotenv.config();

const app = express();

app.use(express.json());


const users = []; // Temporary in-memory user storage


app.listen(5000, () => console.log("Server running on port 5000"));
```

✓ Loads environment variables from .env file.
✓ Uses express.json() to **parse request bodies**.

CHAPTER 3: REGISTERING USERS WITH HASHED PASSWORDS

3.1 Hashing Passwords Before Storing

📌 **Modify server.js to Add a User Registration Endpoint:**

```
app.post("/register", async (req, res) => {
  try {
    const { username, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);
    users.push({ username, password: hashedPassword });
    res.json({ message: "User registered successfully" });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});
```

✓ **Hashes passwords** before storing them for security.
✓ Stores users **temporarily in an array (replace with DB in production)**.

📌 **Example Request (Using Postman or cURL):**

POST http://localhost:5000/register

Content-Type: application/json

```
{
  "username": "Alice",
  "password": "securepassword"
}
```

CHAPTER 4: IMPLEMENTING JWT-BASED LOGIN AUTHENTICATION

4.1 Generating a JWT Token for Logged-In Users

📌 **Modify server.js to Add a Login Route:**

```
app.post("/login", async (req, res) => {
  const { username, password } = req.body;
  const user = users.find(u => u.username === username);

  if (!user || !(await bcrypt.compare(password, user.password))) {
    return res.status(401).json({ message: "Invalid credentials" });
  }
```

```
  const token = jwt.sign({ username: user.username },
process.env.JWT_SECRET, { expiresIn: "1h" });

  res.json({ token });

});
```

✓ **Verifies password using bcrypt.compare()**.
✓ **Generates a JWT token** that expires in **1 hour**.

📌 **Example Request (Using Postman or cURL):**

```
POST http://localhost:5000/login

Content-Type: application/json

{

  "username": "Alice",

  "password": "securepassword"

}
```

📌 **Response:**

```
{

  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

}
```

✓ Returns a **JWT token** for authentication.

CHAPTER 5: PROTECTING ROUTES WITH JWT AUTHENTICATION

5.1 Creating a Middleware to Verify JWT Tokens

📌 **Add JWT Verification Middleware in server.js:**

```
const authenticateToken = (req, res, next) => {
```

```
  const token = req.header("Authorization")?.split(" ")[1]; // Extract Bearer Token

  if (!token) return res.status(403).json({ message: "Access Denied" });


  jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
    if (err) return res.status(403).json({ message: "Invalid Token" });
    req.user = user;
    next();
  });
};
```

✓ **Extracts and verifies JWT tokens from headers**.
✓ Calls next() to allow access to protected routes.

5.2 Protecting an API Endpoint

📌 **Modify server.js to Add a Protected Route:**

```
app.get("/profile", authenticateToken, (req, res) => {
  res.json({ message: `Welcome ${req.user.username}!` });
});
```

✓ Allows only **authenticated users with valid JWT tokens**.

📌 **Example Request (Using Postman or cURL):**

```
GET http://localhost:5000/profile
Authorization: Bearer YOUR_JWT_TOKEN
```

✓ Returns **403 Forbidden** if token is missing or invalid.

CHAPTER 6: BEST PRACTICES FOR JWT AUTHENTICATION

- **1. Store Tokens Securely**

✓ Store JWT tokens in **HTTP-only cookies** for security.
✓ Avoid saving tokens in localStorage (vulnerable to XSS attacks).

- **2. Set Token Expiry Time**

✓ Use **short-lived JWTs (expiresIn: "1h")** for better security.

- **3. Implement Token Refresh Mechanism**

✓ Generate **new tokens before expiration** using refresh tokens.

- **4. Use HTTPS in Production**

✓ Protects **JWTs from being intercepted (man-in-the-middle attacks)**.

Case Study: How Spotify Uses JWT for Secure Authentication

Challenges Faced by Spotify

✓ Handling **millions of user logins securely**.
✓ Ensuring **token-based authentication for API requests**.

Solutions Implemented

✓ **Implemented JWT tokens** for secure authentication.
✓ Used **refresh tokens** to generate new JWTs when expired.
✓ **Stored JWTs in secure HTTP-only cookies**.

- **Key Takeaways from Spotify's Authentication Strategy:**

✓ **JWT-based authentication is scalable & secure**.

✓ **Short-lived tokens improve security**.

✓ **Refresh tokens allow seamless user experience**.

📝 Exercise

✅ Implement **JWT authentication** in an Express.js API.

✅ Create a **protected route (/dashboard)** that requires a valid token.

✅ Store JWT tokens **securely in HTTP-only cookies**.

✅ Implement a **logout feature** that invalidates tokens.

Conclusion

✓ **JWT enables secure, token-based authentication in APIs.**

✓ **Password hashing prevents security breaches.**

✓ **Protecting routes ensures only authenticated users access sensitive data.**

✓ **Best practices like short-lived tokens and HTTPS improve security.**

IMPLEMENTING PASSWORD HASHING WITH BCRYPT.JS

CHAPTER 1: INTRODUCTION TO PASSWORD HASHING

1.1 What is Password Hashing?

Password hashing is the process of **converting plaintext passwords into a secure format** before storing them in a database. Instead of saving passwords directly, hashing ensures that even if the database is compromised, passwords remain unreadable.

- **Why is Password Hashing Important?**

✓ **Protects user credentials from leaks and data breaches.**
✓ **Ensures passwords cannot be easily reversed or decrypted.**
✓ **Adds security against brute force and rainbow table attacks.**

- **Comparison: Plaintext vs. Hashed Passwords**

Storage Method	Example	Security Level
Plaintext Password	"mypassword123"	❌ Very Weak
Hashed Password	$2b$10$...2Ebfh9Q9LzZJ	✅ Strong

- **Common Hashing Algorithms**

Algorithm	Security	Used In
MD5	Weak	Not Recommended
SHA-256	Moderate	Cryptographic applications
bcrypt	Strong	User authentication
Argon2	Strongest	Modern applications

CHAPTER 2: INTRODUCTION TO BCRYPT.JS

2.1 What is bcrypt.js?

bcrypt.js is a popular **password hashing library** used in Node.js applications. It provides:

✓ **Salting mechanism** to prevent rainbow table attacks.

✓ **Adaptive hashing** to make cracking attempts slower over time.

✓ **Secure password storage** for authentication systems.

📌 **Installing bcrypt.js in a Node.js Project**

npm install bcryptjs

CHAPTER 3: HASHING PASSWORDS WITH BCRYPT.JS

3.1 How Does bcrypt.js Work?

1. **Generates a salt** (random value added to the password).
2. **Hashes the password using the salt**.
3. **Stores the hashed password** in the database.

📌 **Example: Hashing a Password in Node.js**

```
const bcrypt = require("bcryptjs");


async function hashPassword(plainPassword) {
  const salt = await bcrypt.genSalt(10);  // Generate salt (10 rounds)
  const hashedPassword = await bcrypt.hash(plainPassword, salt);
  console.log("Hashed Password:", hashedPassword);
```

```
}
```

```
hashPassword("mypassword123");
```

✓ Uses bcrypt.genSalt(10) to **generate a salt**.
✓ Uses bcrypt.hash() to **convert password into a secure hash**.

📌 **Example Output (Hashed Password)**

```
$2a$10$V1CZGZ5J6fW.TP8NXXKfHe...2Ebfh9Q9LzZJ
```

✓ Even if the same password is hashed multiple times, the result is **different each time** due to the **random salt**.

CHAPTER 4: VERIFYING PASSWORDS WITH BCRYPT.JS

4.1 How Password Verification Works

To verify a user's password, bcrypt **compares** the plaintext password with the stored hash.

📌 **Example: Checking a Password Against a Hash**

```
async function verifyPassword(plainPassword, storedHash) {
  const isMatch = await bcrypt.compare(plainPassword, storedHash);
  if (isMatch) {
    console.log("Password is correct!");
  } else {
    console.log("Invalid password!");
  }
```

```
}


// Example usage
verifyPassword("mypassword123",
"$2a$10$V1CZGZ5J6fW.TP8NXXKfHe...2Ebfh9Q9LzZJ");
```

✓ Uses bcrypt.compare() to **match plaintext password with stored hash**.
✓ Returns **true** if the password is correct, otherwise **false**.

📌 **Example Output (Valid Password)**

```
Password is correct!
```

📌 **Example Output (Invalid Password)**

```
Invalid password!
```

CHAPTER 5: IMPLEMENTING BCRYPT.JS IN AN EXPRESS.JS AUTHENTICATION SYSTEM

5.1 Storing Hashed Passwords in MongoDB

📌 **Step 1: Create a Mongoose User Schema (models/User.js)**

```
const mongoose = require("mongoose");
const bcrypt = require("bcryptjs");


const UserSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
```

```
  password: { type: String, required: true }
});

// Hash password before saving user
UserSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
  next();
});

module.exports = mongoose.model("User", UserSchema);
```

✓ Uses pre("save") middleware to **automatically hash passwords before saving them**.

5.2 Creating User Registration Endpoint in Express.js

📌 Step 2: Create a User Signup API (routes/auth.js)

```
const express = require("express");
const bcrypt = require("bcryptjs");
const User = require("../models/User");

const router = express.Router();
```

```
router.post("/register", async (req, res) => {
  try {
    const { username, email, password } = req.body;
    const newUser = new User({ username, email, password });
    await newUser.save();
    res.status(201).json({ message: "User registered successfully!" });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

module.exports = router;
```

✓ Takes **username, email, and password**, hashes the password, and stores it securely.

5.3 Implementing User Login with Password Verification

📌 Step 3: Create a User Login API (routes/auth.js)

```
router.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ message: "Invalid credentials" });
```

```
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(400).json({ message: "Invalid credentials" });

    res.json({ message: "Login successful!" });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});
```

✓ **Checks if the user exists, then verifies the password using bcrypt.compare()**.

📌 **Example Request (Login User)**

```
POST http://localhost:5000/auth/login
Content-Type: application/json
{
  "email": "alice@example.com",
  "password": "mypassword123"
}
```

📌 **Response (Successful Login)**

```
{ "message": "Login successful!" }
```

✓ Verifies **password securely** before granting access.

CHAPTER 6: BEST PRACTICES FOR SECURE PASSWORD HANDLING

- **1. Always Hash Passwords Before Storing**

✓ Never store **plaintext passwords** in a database.

- **2. Use a Strong Salt Factor**

✓ Use at least **10 rounds of salt** (bcrypt.genSalt(10)).

- **3. Implement Rate Limiting for Login Attempts**

✓ Prevent brute-force attacks using express-rate-limit.

- **4. Securely Store User Sessions or Tokens**

✓ Use **JWT (JSON Web Tokens)** for authentication.

Case Study: How Facebook Uses Password Hashing

Challenges Faced by Facebook

✓ **Protecting billions of user accounts** from credential leaks.
✓ **Ensuring password security against brute-force attacks**.

Solutions Implemented

✓ **Used bcrypt hashing** to store passwords securely.
✓ **Implemented multi-factor authentication (MFA)** for added security.
✓ **Deployed rate-limiting mechanisms** to prevent brute-force attacks.

- **Key Takeaways from Facebook's Security Model:**

✓ **Hashed passwords prevent data leaks**.
✓ **Strong salt values slow down brute-force attacks**.

✓ **Layered security (MFA, CAPTCHA) improves authentication safety**.

📝 Exercise

✅ Implement a **password hashing function using bcrypt.js**.
✅ Modify an Express.js app to **secure user passwords in MongoDB**.
✅ Implement **password verification using bcrypt.compare()**.
✅ Add **error handling** for incorrect login attempts.

Conclusion

✓ **bcrypt.js securely hashes and verifies passwords**.
✓ **Salting prevents dictionary & rainbow table attacks**.
✓ **Express.js authentication systems use bcrypt for security**.
✓ **Following best practices ensures strong user authentication**.

SECURING ROUTES WITH AUTHENTICATION MIDDLEWARE IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO AUTHENTICATION MIDDLEWARE

1.1 What is Authentication Middleware?

Authentication middleware is a function that runs **before processing a request** to verify whether a user is **authenticated** and **authorized** to access a route.

- **Why Use Authentication Middleware?**

✓ **Protects private routes** from unauthorized access.
✓ **Validates user identity** using tokens (JWT, OAuth).
✓ **Enhances security** by preventing unauthorized data access.

- **Types of Authentication Middleware:**

Type	Description	Example
Token-based Authentication	Uses JWT for stateless authentication	REST APIs
Session-based Authentication	Uses sessions & cookies	Traditional web apps
OAuth Authentication	Uses third-party login (Google, GitHub)	Social logins

CHAPTER 2: IMPLEMENTING JWT AUTHENTICATION IN EXPRESS.JS

2.1 What is JWT (JSON Web Token)?

JWT (JSON Web Token) is a **secure token format** used for authentication. It consists of:

1. **Header** – Contains signing algorithm (HS256).
2. **Payload** – User data (ID, role, etc.).
3. **Signature** – Verifies token authenticity.

📌 **Example JWT Token (Base64 Encoded):**

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

.eyJ1c2VySWQiOiIxMjM0NTYiLCJyb2xlIjoiYWRtaW4ifQ

.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

✓ **Signature validation ensures the token is secure.**

2.2 Installing Required Packages

📌 **Install JWT & bcrypt for authentication:**

npm install jsonwebtoken bcryptjs express dotenv

✓ jsonwebtoken ⟶ Generates & verifies JWT tokens.
✓ bcryptjs ⟶ Hashes passwords securely.
✓ dotenv ⟶ Manages secret environment variables.

2.3 Setting Up User Authentication (Signup & Login)

📌 **Step 1: Create a middleware/authMiddleware.js File for Authentication Middleware**

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


module.exports = (req, res, next) => {
   const token = req.header("Authorization");
```

```
  if (!token) return res.status(401).json({ message: "Access Denied. No token provided." });

  try {
    const verified = jwt.verify(token.replace("Bearer ", ""), process.env.JWT_SECRET);
    req.user = verified;
    next();
  } catch (error) {
    res.status(403).json({ message: "Invalid or expired token" });
  }
};
```

✓ **Extracts JWT from request headers**.
✓ **Verifies token authenticity** using jwt.verify().
✓ Calls next() to allow access **only if valid**.

📌 Step 2: Creating a User Model (models/User.js)

```
const mongoose = require("mongoose");
const bcrypt = require("bcryptjs");

const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
```

```
  password: { type: String, required: true }
});

// Hash password before saving user
UserSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  this.password = await bcrypt.hash(this.password, 10);
  next();
});

module.exports = mongoose.model("User", UserSchema);
```

✓ **Encrypts passwords** before storing them in MongoDB.

📌 Step 3: Implementing Signup (routes/authRoutes.js)

```
const express = require("express");
const User = require("../models/User");
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");

const router = express.Router();

// User Registration
```

```
router.post("/register", async (req, res) => {
  try {
    const { name, email, password } = req.body;
    let user = await User.findOne({ email });
    if (user) return res.status(400).json({ message: "User already exists" });

    user = new User({ name, email, password });
    await user.save();

    res.status(201).json({ message: "User registered successfully" });
  } catch (error) {
    res.status(500).json({ message: "Error registering user" });
  }
});

module.exports = router;
```

✓ **Validates if the email is already registered**.
✓ **Hashes and stores password securely**.

📌 Step 4: Implementing Login (routes/authRoutes.js)

```
router.post("/login", async (req, res) => {
  try {
```

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

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

    if (!user) return res.status(400).json({ message: "Invalid credentials" });


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

    if (!isMatch) return res.status(400).json({ message: "Invalid credentials" });


    const token = jwt.sign({ userId: user._id }, process.env.JWT_SECRET, { expiresIn: "1h" });

    res.json({ token });

  } catch (error) {

    res.status(500).json({ message: "Error logging in" });

  }

});
```

✓ **Validates email & password before issuing JWT**.
✓ **Token expires after 1 hour** (expiresIn: "1h").

CHAPTER 3: SECURING ROUTES WITH AUTHENTICATION MIDDLEWARE

3.1 Applying Authentication Middleware to Protected Routes

📌 **Example: Protecting a Profile Route (routes/userRoutes.js)**

```
const express = require("express");
const authMiddleware = require("../middleware/authMiddleware");

const router = express.Router();

router.get("/profile", authMiddleware, async (req, res) => {
  res.json({ message: `Welcome, User ID: ${req.user.userId}` });
});

module.exports = router;
```

✓ Uses authMiddleware to **restrict access to authenticated users only**.

📌 **Request Example:**

```
GET http://localhost:5000/api/users/profile
Authorization: Bearer YOUR_JWT_TOKEN
```

✓ **403 Forbidden** if no token is provided.
✓ **200 OK** if valid token is sent.

CHAPTER 4: USING ROLE-BASED ACCESS CONTROL (RBAC)

4.1 Assigning User Roles in the Database

📌 **Modify the UserSchema to Include Roles (models/User.js)**

```
const UserSchema = new mongoose.Schema({
  name: String,
```

```
  email: String,
  password: String,
  role: { type: String, default: "user", enum: ["user", "admin"] }
});
```

✓ Assigns **default role as "user"**.

4.2 Creating Role-Based Middleware (middleware/roleMiddleware.js)

```
module.exports = (role) => {
  return (req, res, next) => {
    if (req.user.role !== role) return res.status(403).json({ message: "Access Denied" });
    next();
  };
};
```

✓ Checks **user role before allowing access**.

📌 **Apply to Protected Admin Route (routes/adminRoutes.js)**

```
const roleMiddleware = require("../middleware/roleMiddleware");

router.get("/admin", authMiddleware, roleMiddleware("admin"), (req, res) => {
  res.json({ message: "Welcome, Admin!" });
});
```

✓ Only **admins can access this route**.

Case Study: How Netflix Secures Routes in Express.js

Challenges Faced by Netflix

✓ **Securing millions of user accounts**.
✓ **Handling role-based permissions (Users vs. Admins)**.

Solutions Implemented

✓ Used **JWT authentication** for secure API access.
✓ Implemented **role-based access control (RBAC)** for admin routes.
✓ Enforced **token expiration** for session security.

🔹 **Key Takeaways from Netflix's Authentication Strategy:**

✓ **JWT ensures stateless & scalable authentication**.
✓ **RBAC enhances security for different user types**.
✓ **Session expiration prevents unauthorized long-term access**.

📝 Exercise

✅ Secure a **profile page** using authentication middleware.
✅ Implement **role-based access control** for admin routes.
✅ Set **JWT expiration to 30 minutes** for better security.

Conclusion

✓ **Authentication middleware protects Express.js routes**.
✓ **JWT ensures secure token-based authentication**.
✓ **Role-based access control (RBAC) restricts user permissions**.
✓ **Middleware improves API security and scalability**.

PREVENTING SQL INJECTION & XSS ATTACKS

CHAPTER 1: INTRODUCTION TO WEB SECURITY

1.1 Why is Web Security Important?

Web applications handle **sensitive user data** and must be protected from malicious attacks. Two of the most common security vulnerabilities are:

✓ **SQL Injection (SQLi)** – Attackers manipulate SQL queries to gain unauthorized database access.

✓ **Cross-Site Scripting (XSS)** – Malicious scripts are injected into web applications to steal user data.

- **Impacts of Security Vulnerabilities:**

Threat	Impact
SQL Injection	Exposes sensitive database information, modifies data, or deletes records.
XSS Attacks	Hijacks user sessions, steals credentials, or injects malicious scripts.

CHAPTER 2: UNDERSTANDING SQL INJECTION & PREVENTION TECHNIQUES

2.1 What is SQL Injection?

SQL Injection (SQLi) occurs when attackers **insert malicious SQL code** into an application's database query, allowing them to access or manipulate sensitive data.

📌 Example: Vulnerable SQL Query in Express.js

```
app.get("/user", (req, res) => {
  const username = req.query.username;
  const query = `SELECT * FROM users WHERE username = '${username}'`; // Unsafe query

  db.query(query, (err, result) => {
    if (err) throw err;
    res.json(result);
  });
});
```

✓ If an attacker enters "admin' --", the SQL query becomes:

```
SELECT * FROM users WHERE username = 'admin' --';
```

✓ This **comments out the password check**, allowing unauthorized access.

2.2 How to Prevent SQL Injection

✅ Use Prepared Statements & Parameterized Queries

📌 Example: Secure SQL Query using Prepared Statements

```
app.get("/user", (req, res) => {
  const query = "SELECT * FROM users WHERE username = ?";
  db.query(query, [req.query.username], (err, result) => {
    if (err) throw err;
```

```
    res.json(result);
  });
});
```

✓ **Prevents attackers from injecting SQL code** into input fields.

✅ **Use ORM like Sequelize to Sanitize Queries**

📌 **Example: Secure Query in Sequelize**

```
const user = await User.findOne({ where: { username: req.query.username } });
```

✓ Sequelize **automatically prevents SQL injection** by using **query parameterization**.

✅ **Escape User Inputs Before Using in Queries**

📌 **Example: Escaping Input in MySQL**

```
const username = mysql.escape(req.query.username);
const query = `SELECT * FROM users WHERE username = ${username}`;
```

✓ **Sanitizes input values** to prevent SQL Injection.

CHAPTER 3: UNDERSTANDING XSS ATTACKS & PREVENTION TECHNIQUES

3.1 What is Cross-Site Scripting (XSS)?

Cross-Site Scripting (XSS) occurs when attackers **inject malicious JavaScript** into web pages that are then executed in a user's browser.

📌 **Example: Vulnerable Code in Express.js**

```
app.get("/search", (req, res) => {
  const searchQuery = req.query.q;
  res.send(`<h1>Results for ${searchQuery}</h1>`); // Unsafe output
});
```

✓ If an attacker enters:

```
<script>alert("Hacked!")</script>
```

✓ This script runs in **every user's browser**, stealing sensitive information.

3.2 How to Prevent XSS Attacks

✅ Escape User Input Before Rendering HTML

📌 Example: Using escape-html to Prevent XSS

```
npm install escape-html
```

```
const escapeHtml = require("escape-html");

app.get("/search", (req, res) => {
  const searchQuery = escapeHtml(req.query.q);
  res.send(`<h1>Results for ${searchQuery}</h1>`); // Safe output
});
```

✓ Converts <script> tags into harmless text.

✅ Use Helmet Middleware to Set Security Headers

📌 **Example: Installing and Using Helmet.js**

```
npm install helmet
```

```
const helmet = require("helmet");
```

```
app.use(helmet());
```

✓ **Prevents common XSS attacks** by securing HTTP headers.

✅ **Use Content Security Policy (CSP)**

📌 **Example: Setting CSP Headers in Express.js**

```
app.use(helmet.contentSecurityPolicy({
  directives: {
    defaultSrc: ["'self'"],
    scriptSrc: ["'self'"]
  }
}));
```

✓ Blocks unauthorized **inline JavaScript execution**.

✅ **Sanitize User Input with express-validator**

📌 **Example: Validating & Sanitizing Input Fields**

```
npm install express-validator
```

```
const { body, validationResult } = require("express-validator");


app.post("/comment",
```

```
  body("comment").escape(),  // Sanitizes the input
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

    res.send("Comment posted safely!");
  }
);
```

✓ Prevents users from **injecting harmful JavaScript code** in inputs.

CHAPTER 4: TESTING FOR SQL INJECTION & XSS ATTACKS

4.1 How to Test SQL Injection Vulnerabilities

📌 Use SQL Injection Testing Tools

- **SQLMap** (Command-line tool for testing SQL injection)
- **Postman** (Manually input SQL injection attempts)

📌 Example SQL Injection Test Input

```
' OR 1=1 --
```

✓ If the database **returns all users,** it is vulnerable to SQL injection.

4.2 How to Test XSS Vulnerabilities

📌 Example Malicious XSS Input for Testing

<script>alert("Hacked!")</script>

✓ If an alert appears, the site **is vulnerable to XSS**.

📌 **Use Automated Security Testing Tools**

- **OWASP ZAP** – Detects XSS vulnerabilities.
- **Burp Suite** – Web security testing tool.

Case Study: How Facebook Prevents SQL Injection & XSS Attacks

Challenges Faced by Facebook

✓ Handling **millions of SQL queries daily**.
✓ Preventing **data breaches from XSS attacks**.

Solutions Implemented

✓ Used **ORMs like GraphQL & Sequelize** for query sanitization.
✓ Implemented **Content Security Policy (CSP)** to block XSS.
✓ Enforced **strict input validation and escaping**.

🔹 **Key Takeaways from Facebook's Security Strategy:**
✓ **Always validate and sanitize user input.**
✓ **Use ORM frameworks to prevent raw SQL injection.**
✓ **Implement security headers (CSP, Helmet.js) for XSS protection.**

📝 **Exercise**

✅ Implement **Sequelize ORM** to prevent raw SQL injection.
✅ Use **express-validator** to sanitize user input.
✅ Set up **CSP headers with Helmet.js** to block XSS.
✅ Perform a **SQL Injection test** on a sample API.

Conclusion

✓ SQL Injection is prevented using prepared statements & ORM frameworks.

✓ XSS attacks are mitigated with escaping, CSP, and Helmet.js.

✓ Security testing tools help identify vulnerabilities in applications.

✓ A secure Express.js application ensures data integrity and user safety.

IMPLEMENTING CORS & RATE LIMITING IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO CORS & RATE LIMITING

1.1 What is CORS (Cross-Origin Resource Sharing)?

CORS (Cross-Origin Resource Sharing) is a **security feature in web browsers** that **prevents unauthorized requests** between different domains. By default, browsers **block cross-origin requests** to protect users.

🔹 **Why is CORS Important?**

✓ Allows **secure API access** from different domains.

✓ Prevents **unauthorized data exposure**.

✓ Enables **public APIs to be consumed securely**.

🔹 **Examples of Cross-Origin Requests:**

Scenario	Cross-Origin?	Example
Same-Origin Request	❌ No	frontend.example.com → backend.example.com ✅
Cross-Origin Request	✅ Yes	frontend.com → api.backend.com ❌

📌 **Example of a Blocked CORS Request:**

Access to fetch at 'https://api.example.com/data' from origin 'https://frontend.com' has been blocked by CORS policy.

✓ Browsers **block the request unless CORS is enabled on the backend**.

CHAPTER 2: ENABLING CORS IN EXPRESS.JS

2.1 Installing CORS Middleware

📌 Step 1: Install the cors package in your Express.js app

```
npm install cors
```

📌 Step 2: Import and Use CORS in server.js

```
const express = require("express");
const cors = require("cors");

const app = express();

// Enable CORS for all requests
app.use(cors());

app.get("/data", (req, res) => {
  res.json({ message: "CORS is enabled!" });
});

app.listen(5000, () => console.log("Server running on port 5000"));
```

✓ Allows requests from **all origins** (*).

2.2 Restricting CORS to Specific Domains

📌 **Example: Allowing Requests Only from https://frontend.com**

```
const corsOptions = {
  origin: "https://frontend.com",
  methods: ["GET", "POST"],
  allowedHeaders: ["Content-Type", "Authorization"]
};

app.use(cors(corsOptions));
```

✓ Allows requests **only from the specified domain**.

2.3 Handling Preflight Requests (OPTIONS Method)

🔹 **Browsers send preflight requests (OPTIONS method) before POST/PUT/DELETE requests.**

📌 **Example: Enabling Preflight Requests in Express**

```
app.options("*", cors());
```

✓ Handles **preflight requests automatically**.

CHAPTER 3: IMPLEMENTING RATE LIMITING IN EXPRESS.JS

3.1 What is Rate Limiting?

Rate limiting **restricts the number of API requests** a user or IP can make in a specific timeframe. It prevents:

✓ **API abuse & spam** requests.

✓ **Denial of Service (DoS) attacks**.

✓ **Excessive resource consumption** on servers.

- **Example Scenarios Where Rate Limiting is Used:**

Use Case	Limit
User Login Requests	Max **5 attempts per minute**
Public API Requests	Max **100 requests per hour**
File Uploads	Max **10 uploads per day**

3.2 Installing & Using express-rate-limit

📌 Step 1: Install express-rate-limit

```
npm install express-rate-limit
```

📌 Step 2: Implement Rate Limiting in server.js

```
const rateLimit = require("express-rate-limit");


const limiter = rateLimit({
  windowMs: 1 * 60 * 1000, // 1 minute
  max: 10, // Limit each IP to 10 requests per windowMs
  message: { error: "Too many requests. Please try again later." }
});


app.use("/api", limiter);
```

✓ **Limits users to 10 requests per minute**.
✓ Returns **429 Too Many Requests** if exceeded.

3.3 Applying Rate Limits to Specific Routes

📌 Example: Rate Limiting Login Requests Only

```
const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5, // Allow 5 login attempts per 15 minutes
  message: { error: "Too many login attempts. Please try again later." }
});

app.post("/login", loginLimiter, (req, res) => {
  res.json({ message: "Login endpoint" });
});
```

✓ Prevents **brute-force attacks** on login systems.

CHAPTER 4: COMBINING CORS & RATE LIMITING IN EXPRESS.JS

4.1 Complete Express.js Setup with CORS & Rate Limiting

📌 server.js - Fully Secured API

```
const express = require("express");
const cors = require("cors");
const rateLimit = require("express-rate-limit");

const app = express();
```

```
// CORS Configuration
const corsOptions = {
  origin: ["https://trusted-site.com"],
  methods: ["GET", "POST"],
  allowedHeaders: ["Content-Type", "Authorization"]
};

app.use(cors(corsOptions));

// Rate Limiting Configuration
const apiLimiter = rateLimit({
  windowMs: 1 * 60 * 1000, // 1 minute
  max: 10, // 10 requests per minute
  message: { error: "Too many requests, please try again later." }
});

app.use("/api", apiLimiter);

// Sample API Route
app.get("/api/data", (req, res) => {
  res.json({ message: "Protected API Data" });
});
```

```
app.listen(5000, () => console.log("Server running on port 5000"));
```

✓ Implements **CORS for specific domains**.
✓ Applies **rate limiting to all API routes**.

Chapter 5: Best Practices for CORS & Rate Limiting

- **1. Set CORS Rules Based on Environment**

✓ Allow **all origins in development,** but restrict in **production**.

```
const allowedOrigins = process.env.NODE_ENV === "production" ? ["https://frontend.com"] : "*";

app.use(cors({ origin: allowedOrigins }));
```

- **2. Use Different Rate Limits for Different APIs**

✓ **Stricter limits for authentication endpoints** (login, signup).
✓ **Higher limits for public APIs** (/products, /articles).

- **3. Log & Monitor API Usage**

✓ Use **logging tools like Winston or Datadog** to track API abuse.
✓ Store **rate-limited requests in logs for analysis**.

Case Study: How GitHub Uses CORS & Rate Limiting

Challenges Faced by GitHub

✓ Handling **millions of API requests daily**.
✓ Protecting **public APIs from abuse**.
✓ Ensuring **secure access for frontend clients**.

Solutions Implemented

✓ Used **CORS headers** to allow requests from trusted domains only.
✓ Implemented **rate limits of 60 requests per hour per user**.
✓ Provided **authentication-based rate increases** for verified users.

🔹 **Key Takeaways from GitHub's API Strategy:**

✓ **CORS ensures only trusted applications access APIs.**
✓ **Rate limiting protects APIs from excessive traffic & attacks.**
✓ **Logging helps monitor & optimize API usage.**

📝 **Exercise**

✅ Implement **CORS in an Express.js API** and allow requests only from a specific domain.
✅ Apply **rate limiting** to prevent more than **5 login attempts per minute**.
✅ Test API security using **Postman or cURL** to verify blocked requests.
✅ Optimize API security by combining **CORS, rate limiting, and JWT authentication**.

Conclusion

✓ **CORS controls access to APIs, preventing unauthorized cross-origin requests.**
✓ **Rate limiting protects APIs from abuse, ensuring fair usage.**
✓ **Combining CORS & rate limiting enhances API security & performance.**
✓ **Best practices like logging & monitoring help track API usage trends.**

HANDLING SESSION AUTHENTICATION WITH EXPRESS SESSIONS

CHAPTER 1: INTRODUCTION TO SESSION AUTHENTICATION

1.1 What is Session Authentication?

Session authentication is a method of managing **user logins** by storing authentication details on the server. It allows users to stay **logged in across multiple requests** without needing to re-authenticate each time.

- **Why Use Session Authentication?**

✓ More **secure than token-based authentication** for sensitive applications.
✓ Prevents users from **re-entering login credentials repeatedly**.
✓ Uses **session IDs stored in cookies** for authentication.

- **How Session Authentication Works:**

1. **User logs in** → Server **creates a session** and stores user info.
2. **Session ID is sent** to the client as a **cookie**.
3. **Client makes authenticated requests** → Server verifies session ID.
4. **User logs out** → Session is destroyed on the server.

CHAPTER 2: SETTING UP EXPRESS SESSIONS

2.1 Installing express-session

📌 **Step 1: Install Required Packages**

npm install express-session connect-mongo

✓ express-session → Middleware for handling sessions.
✓ connect-mongo → Stores sessions in MongoDB (persistent storage).

2.2 Configuring Session Middleware in Express.js

📌 Step 2: Set Up Sessions in server.js

```
const express = require("express");

const session = require("express-session");


const app = express();


app.use(session({
  secret: "mysecretkey", // Used to sign the session ID
  resave: false,
  saveUninitialized: false,
  cookie: { secure: false, maxAge: 1000 * 60 * 30 } // 30 minutes expiry
}));


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

✓ secret → Encrypts the session ID.
✓ resave: false → Prevents unnecessary session saving.
✓ saveUninitialized: false → Prevents empty sessions from being

stored.

✓ cookie.maxAge → **Session expiration time (30 min)**.

CHAPTER 3: IMPLEMENTING USER AUTHENTICATION WITH SESSIONS

3.1 User Login with Session Authentication

📌 Step 3: Implement Login Route (routes/auth.js)

```
const express = require("express");

const session = require("express-session");


const router = express.Router();


// Mock User Database

const users = [{ id: 1, username: "admin", password: "password123" }];


router.post("/login", (req, res) => {

  const { username, password } = req.body;

  const user = users.find(u => u.username === username && u.password === password);


  if (!user) {

    return res.status(401).json({ message: "Invalid credentials" });

  }
```

```
  req.session.user = user; // Store user info in session
  res.json({ message: "Login successful!" });
});

module.exports = router;
```

✓ **Stores user data in req.session upon successful login.**

📌 **Example Request (Login User)**

```
POST http://localhost:3000/auth/login
Content-Type: application/json
{
  "username": "admin",
  "password": "password123"
}
```

📌 **Response (Successful Login)**

```
{ "message": "Login successful!" }
```

3.2 Protecting Routes Using Session Authentication

📌 Step 4: Create Middleware to Check Session (middleware/authMiddleware.js)

```
function isAuthenticated(req, res, next) {
  if (!req.session.user) {
```

```
    return res.status(403).json({ message: "Unauthorized access" });
  }
  next();
}
module.exports = isAuthenticated;
```

✓ Ensures **only logged-in users can access protected routes**.

📌 **Step 5: Protect Routes in Express**

```
const isAuthenticated = require("../middleware/authMiddleware");

router.get("/profile", isAuthenticated, (req, res) => {
  res.json({ message: `Welcome ${req.session.user.username}!` });
});
```

✓ Returns **profile information only if the user is logged in**.

3.3 Implementing Logout Functionality

📌 **Step 6: Destroying User Session on Logout**

```
router.post("/logout", (req, res) => {
  req.session.destroy(err => {
    if (err) return res.status(500).json({ message: "Logout failed" });
    res.json({ message: "Logged out successfully" });
  });
});
```

✓ Calls req.session.destroy() to **remove the session from the server**.

📌 **Example Request (Logout User)**

POST http://localhost:3000/auth/logout

📌 **Response (Successful Logout)**

{ "message": "Logged out successfully" }

CHAPTER 4: STORING SESSIONS IN MONGODB USING CONNECT-MONGO

4.1 Why Store Sessions in MongoDB?

✓ Allows **session persistence even after server restarts**.
✓ Scales well with **multiple servers (load balancing)**.

📌 **Step 7: Configure MongoDB Session Store in server.js**

```
const session = require("express-session");
const MongoStore = require("connect-mongo");
const mongoose = require("mongoose");

mongoose.connect("mongodb://127.0.0.1:27017/sessionDB", {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

app.use(session({
```

```
  secret: "mysecretkey",
  resave: false,
  saveUninitialized: false,
  store: MongoStore.create({ mongoUrl: "mongodb://127.0.0.1:27017/sessionDB" }),
  cookie: { secure: false, maxAge: 1000 * 60 * 30 }
}));
```

✓ Uses connect-mongo to **store sessions in MongoDB**.

CHAPTER 5: BEST PRACTICES FOR SECURE SESSIONS

🔹 1. Set Secure Cookie Options

✓ Enable cookie.secure: true in production (requires HTTPS).

🔹 2. Implement Session Expiry

✓ Use cookie.maxAge to **auto-expire sessions** after inactivity.

🔹 3. Prevent Session Hijacking

✓ Use **httpOnly: true** to prevent **JavaScript access to cookies**.
✓ Use **sameSite: strict** to prevent **cross-site request forgery (CSRF)**.

📌 Example: Secure Session Configuration

```
app.use(session({
  secret: "mysecretkey",
  resave: false,
  saveUninitialized: false,
  cookie: {
```

```
    secure: true, // Enables HTTPS-only cookies
    httpOnly: true, // Prevents JavaScript from accessing cookies
    sameSite: "strict", // Prevents CSRF attacks
    maxAge: 1000 * 60 * 30 // Session expires after 30 min
  }
}));
```

Case Study: How Amazon Uses Session Authentication

Challenges Faced by Amazon

✓ Managing **millions of user sessions** across multiple servers.
✓ Preventing **unauthorized access** to shopping carts and payments.

Solutions Implemented

✓ Used **session-based authentication** for logged-in users.
✓ Stored sessions in **distributed databases** (MongoDB, DynamoDB).
✓ Implemented **session expiry policies** for inactive users.

- **Key Takeaways from Amazon's Security Model:**

✓ Session-based authentication improves security over token-based auth.
✓ Persistent sessions allow users to stay logged in across multiple devices.
✓ Session management helps prevent unauthorized shopping cart access.

📝 **Exercise**

✅ Implement **session-based authentication** in an Express.js app.
✅ Store session data **in MongoDB using connect-mongo**.
✅ Create a **protected route (/profile) that requires authentication**.
✅ Implement **session expiration and secure cookie options**.

Conclusion

✓ Session authentication stores user login state on the server.
✓ Express-session provides an easy way to handle sessions in Node.js.
✓ Sessions can be stored in MongoDB for persistence across server restarts.
✓ Best practices like httpOnly, sameSite, and secure cookies improve security.

ASSIGNMENT:

BUILD AN EXPRESS API WITH USER AUTHENTICATION AND ROLE-BASED ACCESS CONTROL (RBAC).

Assignment Solution: Build an Express API with User Authentication & Role-Based Access Control (RBAC)

Step 1: Setting Up the Express.js Project

1.1 Create a New Project Folder

📌 **Open the terminal and run:**

mkdir express-auth-rbac

cd express-auth-rbac

✓ Creates a new project directory.

1.2 Initialize a Node.js Project

📌 **Run the following command:**

npm init -y

✓ Generates a package.json file.

1.3 Install Required Dependencies

📌 **Install Express and authentication packages:**

npm install express mongoose jsonwebtoken bcryptjs dotenv cors body-parser

✓ **Express.js** → Handles API routes.
✓ **Mongoose** → Manages MongoDB database.
✓ **jsonwebtoken** → Implements JWT authentication.
✓ **bcryptjs** → Encrypts user passwords.

✓ **dotenv** → Manages secret environment variables.

✓ **CORS & body-parser** → Handles API requests securely.

Step 2: Setting Up the Express Server

📌 **Create a file server.js and add:**

```
const express = require("express");

const mongoose = require("mongoose");

const dotenv = require("dotenv");

const cors = require("cors");

const bodyParser = require("body-parser");


// Load environment variables

dotenv.config();


const app = express();

app.use(express.json());

app.use(cors());

app.use(bodyParser.json());


const PORT = process.env.PORT || 5000;


// Connect to MongoDB
```

```
mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log("Connected to MongoDB"))
  .catch(err => console.error("MongoDB connection error:", err));

// Import routes
const authRoutes = require("./routes/authRoutes");
const userRoutes = require("./routes/userRoutes");

app.use("/api/auth", authRoutes);
app.use("/api/users", userRoutes);

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✓ **Starts an Express.js server** on port 5000.
✓ **Connects to MongoDB** using Mongoose.
✓ **Loads authentication and user routes**.

📌 **Create a .env file and add:**

```
MONGO_URI=mongodb://127.0.0.1:27017/authDB
JWT_SECRET=your_jwt_secret
```

✓ Stores **database connection and JWT secret** securely.

Step 3: Creating the User Model in MongoDB

📌 Create a folder models/ and inside, create User.js:

```
const mongoose = require("mongoose");
const bcrypt = require("bcryptjs");

const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  role: { type: String, default: "user", enum: ["user", "admin"] }
});

// Hash password before saving user
UserSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  this.password = await bcrypt.hash(this.password, 10);
  next();
});

module.exports = mongoose.model("User", UserSchema);
```

✓ **Encrypts passwords before saving**.
✓ **Includes role field for RBAC (user or admin)**.

Step 4: Implementing Authentication (Signup & Login)

📌 Create a routes/authRoutes.js file:

```
const express = require("express");
const User = require("../models/User");
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");

const router = express.Router();

// User Registration
router.post("/register", async (req, res) => {
  try {
    const { name, email, password, role } = req.body;
    let user = await User.findOne({ email });
    if (user) return res.status(400).json({ message: "User already exists" });

    user = new User({ name, email, password, role });
    await user.save();

    res.status(201).json({ message: "User registered successfully" });
  } catch (error) {
    res.status(500).json({ message: "Error registering user" });
```

```
  }
});

// User Login
router.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ message: "Invalid credentials" });

    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(400).json({ message: "Invalid credentials" });

    const token = jwt.sign({ userId: user._id, role: user.role }, process.env.JWT_SECRET, { expiresIn: "1h" });
    res.json({ token });
  } catch (error) {
    res.status(500).json({ message: "Error logging in" });
  }
});
```

```
module.exports = router;
```

✓ **Registers new users and stores hashed passwords**.
✓ **Generates JWT token upon successful login**.

Step 5: Creating Authentication Middleware

📌 **Create a middleware/authMiddleware.js file:**

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


module.exports = (req, res, next) => {
  const token = req.header("Authorization");
  if (!token) return res.status(401).json({ message: "Access Denied. No token provided." });

  try {
    const verified = jwt.verify(token.replace("Bearer ", ""), process.env.JWT_SECRET);
    req.user = verified;
    next();
  } catch (error) {
    res.status(403).json({ message: "Invalid or expired token" });
  }
};
```

✓ **Validates JWT tokens** before granting access.

Step 6: Implementing Role-Based Access Control (RBAC)

📌 Create a middleware/roleMiddleware.js file:

```
module.exports = (role) => {
  return (req, res, next) => {
    if (req.user.role !== role) return res.status(403).json({ message: "Access Denied" });
    next();
  };
};
```

✓ **Restricts route access based on user role**.

📌 Apply to Admin Routes (routes/userRoutes.js)

```
const express = require("express");
const authMiddleware = require("../middleware/authMiddleware");
const roleMiddleware = require("../middleware/roleMiddleware");

const router = express.Router();

// Protected route for all users
router.get("/profile", authMiddleware, (req, res) => {
  res.json({ message: `Welcome, User ID: ${req.user.userId}` });
});
```

```
// Admin-only route
router.get("/admin", authMiddleware, roleMiddleware("admin"), (req, res) => {
  res.json({ message: "Welcome, Admin!" });
});

module.exports = router;
```

✓ **Only admins can access /admin route.**

Step 7: Testing the API with Postman or cURL

📌 Register a User (User Role: Admin)

```
POST http://localhost:5000/api/auth/register
Content-Type: application/json
{
  "name": "Admin User",
  "email": "admin@example.com",
  "password": "adminpassword",
  "role": "admin"
}
```

📌 Login & Get JWT Token

```
POST http://localhost:5000/api/auth/login
```

Content-Type: application/json

{

"email": "admin@example.com",

"password": "adminpassword"

}

📌 **Access Admin Route (Requires Token)**

GET http://localhost:5000/api/users/admin

Authorization: Bearer YOUR_JWT_TOKEN

Conclusion

✓ Implemented JWT authentication in Express.js.
✓ Created authentication middleware to protect routes.
✓ Implemented role-based access control (RBAC) using middleware.
✓ Tested API with Postman & cURL.