Node.js Authentication - Comprehensive Notes

JWT, Sessions, Cookies & bcrypt Security

🎯 Learning Objectives

By the end of this guide, you will understand:

- Why authentication is necessary in web applications
- How to securely hash passwords using bcrypt
- How JWT (JSON Web Tokens) work and when to use them
- How session-based authentication works with cookies
- Security best practices for authentication systems
- Real-world implementation with code examples

Part 1: The Foundation - Understanding Authentication

1.1 The Core Problem: HTTP is Stateless

⚠️ The Fundamental Challenge

HTTP protocol is **stateless**, meaning each request is independent. The server doesn't "remember" previous requests from the same client.

Example: If you log in, and then request your profile page, the server has NO IDEA you just logged in!

🏦 *Real-World Analogy: The Forgetful Bank Teller*

Imagine a bank teller with severe amnesia who forgets you after every interaction:

- ***You:*** *"Hi, I'd like to check my balance."*
- ***Teller:*** *"Sure! Can I see your ID?" (verifies)*
- ***You:*** *"Now I'd like to withdraw $100."*
- ***Teller:*** *"Who are you? I need to see your ID again!"*

This is HTTP! Every request needs proof of identity.

1.2 The Solutions

We have two main approaches to solve this problem:

Approach	How It Works	Storage Location	Nature
Sessions (with Cookies)	Server stores user data, sends ID to client	Server-side (database/memory)	Stateful
JWT Tokens	Server signs user data, sends to client	Client-side (localStorage/cookie)	Stateless

Part 2: Password Security with bcrypt CRITICAL

2.1 Why NEVER Store Plain Text Passwords

❌ NEVER DO THIS!

```
// BAD! TERRIBLE! NEVER! const user = { email: "john@example.com", password:
"myPassword123" // 😱 PLAIN TEXT! } await database.save(user);
```

Why this is catastrophic:

- If your database is breached, ALL passwords are exposed
- Database admins can see passwords
- Logs might accidentally record passwords
- Users often reuse passwords across sites

2.2 Hashing: The One-Way Street

What is Hashing?

Hashing is a one-way mathematical function that transforms input data into a fixed-length string of characters.

Key Properties:

1. **Deterministic:** Same input always produces same output
2. **One-Way:** Cannot reverse the process (can't "unhash")
3. **Avalanche Effect:** Small input change = completely different output
4. **Fixed Length:** Output is always the same length

🍳 *The Scrambled Egg Analogy*

Input: *Raw egg (your password)* 🥚

Process: *Cooking/scrambling (hash function)* 🔥

Output: *Scrambled egg (hash)* 🍽️

Critical Point: *You CANNOT "unscramble" the egg back to its raw state! Similarly, you cannot "unhash" a password hash.*

2.3 Enter bcrypt: The Gold Standard

Why bcrypt?

- **Salting:** Adds random data to prevent rainbow table attacks
- **Slow by Design:** Intentionally computationally expensive to prevent brute-force
- **Adaptive:** Can increase cost factor as computers get faster
- **Battle-Tested:** Industry standard for password hashing

2.3.1 Installing bcrypt

```
npm install bcrypt
```

2.3.2 Hashing a Password (Registration)

```
const bcrypt = require('bcrypt'); async function registerUser(email, plainPassword) { // Salt
rounds: computational cost (10-12 is standard) // Higher = more secure but slower const
saltRounds = 10; try { // Hash the password const hashedPassword = await
bcrypt.hash(plainPassword, saltRounds); // hashedPassword looks like: //
"$2b$10$N9qo8uLOickgx2ZMRZoMye..." // ─┬─ ─┬ ─────────┬───────── ─────────┬───────── // │ │ │ │
// │ │ │ └─ Hash // │ │ └─ Salt (auto-generated) // │ └─ Cost factor (10) // └─ bcrypt
algorithm version // Save to database await database.users.insert({ email: email, password:
hashedPassword // Store ONLY the hash! }); console.log('User registered successfully!'); }
catch (error) { console.error('Registration failed:', error); } } // Example usage
registerUser('john@example.com', 'mySecurePassword123');
```

⚠️ Important: Understanding Salt Rounds

Salt Rounds = 2^n iterations

Rounds	Time (approx)	Recommendation
10	~100ms	✅ Good for most apps
12	~300ms	✅ High security apps
15	~3 seconds	⚠️ Very slow, use with caution

Balance: Higher rounds = more secure but slower login experience

2.3.3 Verifying a Password (Login)

```
const bcrypt = require('bcrypt'); async function loginUser(email, enteredPassword) { try { //
1. Find user in database const user = await database.users.findOne({ email: email }); if
(!user) { return { success: false, message: 'Invalid credentials' }; } // 2. Compare entered
password with stored hash const isPasswordCorrect = await bcrypt.compare( enteredPassword,
user.password // The stored hash ); if (isPasswordCorrect) { // 3. Password is correct! return
{ success: true, message: 'Login successful', userId: user.id }; } else { // 4. Password is
wrong return { success: false, message: 'Invalid credentials' }; } } catch (error) {
console.error('Login failed:', error); return { success: false, message: 'Login error' }; } }
// Example usage const result = await loginUser('john@example.com', 'mySecurePassword123');
console.log(result);
```

🎯 How bcrypt.compare() Works

bcrypt.compare() is magical because:

1. It extracts the **salt** from the stored hash

2. It uses the same salt to hash the entered password
3. It compares the two hashes
4. Returns true if they match, false otherwise

You don't need to store the salt separately! It's embedded in the hash.

🧠 Brain Teaser

If two users have the same password ("password123"), will their bcrypt hashes be the same?

Answer: NO!

Each bcrypt hash includes a randomly generated salt, so even identical passwords produce different hashes:

```
User 1: $2b$10$abc123...xyz789 User 2: $2b$10$def456...uvw012
```

This prevents attackers from identifying users with the same password (rainbow table attack prevention).

Part 3: JWT (JSON Web Tokens) - Stateless Authentication

3.1 What is JWT?

JWT Definition

JWT (JSON Web Token) is a compact, URL-safe means of representing claims to be transferred between two parties.

Think of it as a **digitally signed certificate** that proves who you are.

🎟️ *The Concert Ticket Analogy*

A JWT is like a concert ticket:

- ✅ ***Contains information:*** *Your name, seat number, show time (payload)*
- ✅ ***Signed by the venue:*** *Has a hologram/stamp that can't be forged (signature)*
- ✅ ***You carry it:*** *You hold the ticket, not the venue (client-side storage)*
- ✅ ***Show it to enter:*** *Present it each time you need access (send with requests)*
- ✅ ***Self-contained:*** *Has all info needed, no need to look you up (stateless)*

3.2 JWT Structure: Three Parts

```
JWT = HEADER.PAYLOAD.SIGNATURE Example:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiIxMjMiLCJlbWFpbCI6InVzZXJAZXhhbXBsZS5jb20ifQ.4kN7vN0w2Xr
└────────── Part 1 ──────────┘ └──────────────── Part 2 ────────────────┘ └───────── Part
3 ─────────┘ HEADER PAYLOAD SIGNATURE
```

Part 1: HEADER

```
{ "alg": "HS256", // Algorithm: HMAC SHA-256 "typ": "JWT" // Type: JSON Web Token }
```

This is **Base64URL encoded** to create the first part of the JWT.

Part 2: PAYLOAD (Claims)

```
{ "userId": "12345", "email": "user@example.com", "role": "admin", "iat": 1516239022, //
Issued At (timestamp) "exp": 1516242622 // Expiration (timestamp) }
```

This contains the actual data (claims). Also **Base64URL encoded**.

> ### ⚠️ CRITICAL: Payload is NOT Encrypted!
>
> Anyone can decode the payload. **Never put sensitive data like passwords in JWT!**
>
> Base64 encoding ≠ Encryption. It's just encoding for URL-safe transmission.

Part 3: SIGNATURE

```
HMACSHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload), secret_key )
```

The signature ensures:

1. **Integrity:** The token hasn't been tampered with
2. **Authenticity:** It was created by someone with the secret key (your server)

3.3 How JWT Authentication Works

```
JWT Authentication Flow: 1. [User] ─────(Login: email + password)─────> [Server] │ ├
Verify credentials ├ Hash password matches? │ 2. [User] <────(JWT Token:
"eyJhbGc...")──────── [Server] │ ├ Store token (localStorage/cookie) │ 3. [User]
─────(Request + JWT in header)──────> [Server] "GET /profile" │ "Authorization: Bearer
eyJhbGc..." ├ Verify JWT signature ├ Check expiration ├ Extract user data 4. [User]
<────(Protected resource)─────────── [Server] "Your profile data..."
```

3.4 Implementing JWT in Node.js

3.4.1 Installation

```
npm install jsonwebtoken bcrypt express dotenv
```

3.4.2 Complete JWT Authentication System

```
// server.js const express = require('express'); const jwt = require('jsonwebtoken'); const
bcrypt = require('bcrypt'); require('dotenv').config(); const app = express();
app.use(express.json()); // In-memory user storage (use database in production) const users =
[]; // ──────────────────────────────────────────────────────── // REGISTRATION ENDPOINT //
──────────────────────────────────────────────────────── app.post('/register', async (req,
res) => { try { const { email, password } = req.body; // Validation if (!email || !password) {
return res.status(400).json({ error: 'Email and password required' }); } // Check if user
exists const existingUser = users.find(u => u.email === email); if (existingUser) { return
res.status(409).json({ error: 'User already exists' }); } // Hash password const
hashedPassword = await bcrypt.hash(password, 10); // Create user const newUser = { id:
users.length + 1, email: email, password: hashedPassword }; users.push(newUser);
res.status(201).json({ message: 'User registered successfully', userId: newUser.id }); } catch
(error) { res.status(500).json({ error: 'Registration failed' }); } }); //
──────────────────────────────────────────────────────────── // LOGIN ENDPOINT //
──────────────────────────────────────────────────────────── app.post('/login', async (req, res)
=> { try { const { email, password } = req.body; // Find user const user = users.find(u =>
u.email === email); if (!user) { return res.status(401).json({ error: 'Invalid credentials'
}); } // Verify password const isPasswordValid = await bcrypt.compare(password,
user.password); if (!isPasswordValid) { return res.status(401).json({ error: 'Invalid
credentials' }); } // Create JWT token const token = jwt.sign( { userId: user.id, email:
user.email }, process.env.JWT_SECRET, // Secret key from .env { expiresIn: '24h' // Token
expires in 24 hours } ); res.json({ message: 'Login successful', token: token }); } catch
(error) { res.status(500).json({ error: 'Login failed' }); } }); //
──────────────────────────────────────────────────────────── // AUTHENTICATION MIDDLEWARE //
──────────────────────────────────────────────────────────── function authenticateToken(req, res,
next) { // Get token from Authorization header // Format: "Authorization: Bearer " const
authHeader = req.headers['authorization']; const token = authHeader && authHeader.split(' ')
[1]; if (!token) { return res.status(401).json({ error: 'Access token required' }); } //
Verify token jwt.verify(token, process.env.JWT_SECRET, (err, user) => { if (err) { return
res.status(403).json({ error: 'Invalid or expired token' }); } // Attach user info to request
object req.user = user; next(); }); } //
──────────────────────────────────────────────────────────── // PROTECTED ROUTES (require
authentication) // ────────────────────────────────────────────────────────────
app.get('/profile', authenticateToken, (req, res) => { // req.user contains decoded JWT data
const user = users.find(u => u.id === req.user.userId); res.json({ message: 'Welcome to your
profile!', user: { id: user.id, email: user.email } }); }); app.get('/dashboard',
authenticateToken, (req, res) => { res.json({ message: `Hello ${req.user.email}!`, data: 'Your
secret dashboard data 📊 ' }); }); // ────────────────────────────────────────────────────────────
// START SERVER // ──────────────────────────────────────────────────────────── const PORT =
process.env.PORT || 3000; app.listen(PORT, () => { console.log(`Server running on port
${PORT}`); });
```

3.4.3 Environment Variables (.env file)

```
# .env file - NEVER commit this to git!
JWT_SECRET=your_super_secret_jwt_key_here_make_it_long_and_random_123456789 PORT=3000
```

> ### 🔒 Security Warning
>
> **NEVER hardcode secrets in your code!**
>
> Always use environment variables and add `.env` to your `.gitignore` file.

3.4.4 Testing the JWT API

```
# 1. Register a new user curl -X POST http://localhost:3000/register \ -H "Content-Type:
application/json" \ -d '{"email":"test@example.com","password":"SecurePass123"}' # Response:
{"message":"User registered successfully","userId":1} # 2. Login to get JWT token curl -X POST
http://localhost:3000/login \ -H "Content-Type: application/json" \ -d
'{"email":"test@example.com","password":"SecurePass123"}' # Response: {"message":"Login
successful","token":"eyJhbGc..."} # 3. Access protected route with token curl
http://localhost:3000/profile \ -H "Authorization: Bearer eyJhbGc..." # Response:
{"message":"Welcome to your profile!","user":{...}}
```

Part 4: Session-Based Authentication with Cookies

4.1 What are Sessions?

> ### Session Definition
>
> A **session** is server-side storage of user state. The server stores user information and gives the client a session ID.

> ### 🏨 *The Hotel Key Card Analogy*
>
> *Session-based auth is like a hotel:*
>
> - 🎫 ***Check-in:*** *You register/login (authenticate)*
> - 🔑 ***Key card:*** *Server gives you a session ID (in a cookie)*
> - 🏨 ***Reservation system:*** *Hotel keeps all your details (server-side session store)*
> - 🚪 ***Room access:*** *You show the key card, hotel looks up your reservation*
> - 🛎️ ***Services:*** *Hotel knows who you are from the key card number*
> - 👋 ***Check-out:*** *Logout destroys the session*

4.2 What are Cookies?

> ### Cookie Definition
>
> A **cookie** is a small piece of data (max 4KB) that:
>
> - ✅ Is stored by the browser
> - ✅ Is sent automatically with every request to the same domain
> - ✅ Can have an expiration date
> - ✅ Can be marked `HttpOnly` (not accessible via JavaScript - XSS protection)
> - ✅ Can be marked `Secure` (only sent over HTTPS)
> - ✅ Can be marked `SameSite` (CSRF protection)

4.3 Session vs JWT Comparison

Aspect	Sessions	JWT
Storage	Server-side (database/Redis)	Client-side (localStorage/cookie)
State	Stateful (server remembers)	Stateless (server doesn't store)
Scalability	Harder (need shared session store)	Easier (no server storage)
Revocation	Easy (delete session from DB)	Hard (token valid until expiry)
Size	Small (just session ID)	Larger (contains all data)
Security	More control (server-side)	Less control (client-side)

4.4 Implementing Sessions in Node.js

4.4.1 Installation

```
npm install express-session connect-mongo bcrypt dotenv
```

4.4.2 Complete Session-Based Authentication

```
// server.js const express = require('express'); const session = require('express-session');
const MongoStore = require('connect-mongo'); const bcrypt = require('bcrypt');
require('dotenv').config(); const app = express(); app.use(express.json()); //
──────────────────────────────────────────────────────────── // SESSION CONFIGURATION //
──────────────────────────────────────────────────────────── app.use(session({ secret:
process.env.SESSION_SECRET, // Secret key for signing resave: false, // Don't save session if
unmodified saveUninitialized: false, // Don't create session until something stored // Store
sessions in MongoDB (production-ready) store: MongoStore.create({ mongoUrl:
process.env.MONGODB_URI, touchAfter: 24 * 3600 // Lazy session update (once per 24h) }), //
Cookie settings cookie: { secure: process.env.NODE_ENV === 'production', // HTTPS only in
production httpOnly: true, // Not accessible via JavaScript (XSS protection) maxAge: 1000 * 60
* 60 * 24 * 7, // 7 days sameSite: 'strict' // CSRF protection } })); // In-memory user
storage (use database in production) const users = []; //
──────────────────────────────────────────────────────────── // REGISTRATION //
──────────────────────────────────────────────────────────── app.post('/register', async (req,
res) => { try { const { email, password } = req.body; if (!email || !password) { return
res.status(400).json({ error: 'Missing fields' }); } const existingUser = users.find(u =>
u.email === email); if (existingUser) { return res.status(409).json({ error: 'User exists' });
} const hashedPassword = await bcrypt.hash(password, 10); const newUser = { id: users.length +
1, email: email, password: hashedPassword }; users.push(newUser); res.status(201).json({
message: 'Registration successful', userId: newUser.id }); } catch (error) {
res.status(500).json({ error: 'Registration failed' }); } }); //
──────────────────────────────────────────────────────────── // LOGIN //
──────────────────────────────────────────────────────────── app.post('/login', async (req, res)
=> { try { const { email, password } = req.body; const user = users.find(u => u.email ===
email); if (!user) { return res.status(401).json({ error: 'Invalid credentials' }); } const
isValid = await bcrypt.compare(password, user.password); if (!isValid) { return
res.status(401).json({ error: 'Invalid credentials' }); } // Create session req.session.userId
= user.id; req.session.email = user.email; res.json({ message: 'Login successful', user: { id:
user.id, email: user.email } }); } catch (error) { res.status(500).json({ error: 'Login
failed' }); } }); // ──────────────────────────────────────────────────────────── // LOGOUT //
──────────────────────────────────────────────────────────── app.post('/logout', (req, res) => {
req.session.destroy((err) => { if (err) { return res.status(500).json({ error: 'Logout failed'
}); } res.clearCookie('connect.sid'); // Clear session cookie res.json({ message: 'Logged out
successfully' }); }); }); // ──────────────────────────────────────────────────────────── //
AUTHENTICATION MIDDLEWARE // ────────────────────────────────────────────────────────────
function requireAuth(req, res, next) { if (!req.session.userId) { return
res.status(401).json({ error: 'Not authenticated' }); } next(); } //
──────────────────────────────────────────────────────────── // PROTECTED ROUTES //
──────────────────────────────────────────────────────────── app.get('/profile', requireAuth,
(req, res) => { const user = users.find(u => u.id === req.session.userId); res.json({ message:
'Your profile', user: { id: user.id, email: user.email } }); }); app.get('/dashboard',
requireAuth, (req, res) => { res.json({ message: `Welcome ${req.session.email}!`, data: 'Your
dashboard data 📊' }); }); // ──────────────────────────────────────────────────────────── //
START SERVER // ──────────────────────────────────────────────────────────── const PORT =
process.env.PORT || 3000; app.listen(PORT, () => { console.log(`Server running on port
${PORT}`); });
```

4.4.3 Environment Variables for Sessions

```
# .env SESSION_SECRET=your_super_secret_session_key_make_it_random_and_long
MONGODB_URI=mongodb://localhost:27017/myapp PORT=3000 NODE_ENV=development
```

Part 5: Security Best Practices CRITICAL

5.1 The Security Checklist

🛡️ Essential Security Measures

1. ✅ **Always use HTTPS in production** (encrypts data in transit)
2. ✅ **Hash passwords with bcrypt** (never store plain text)
3. ✅ **Store secrets in environment variables** (never hardcode)
4. ✅ **Use httpOnly cookies** (prevent XSS attacks)
5. ✅ **Use secure cookies** (HTTPS only)
6. ✅ **Set SameSite=strict** (prevent CSRF attacks)
7. ✅ **Implement rate limiting** (prevent brute-force)
8. ✅ **Validate all input** (prevent injection attacks)
9. ✅ **Set token expiration** (limit damage if stolen)
10. ✅ **Use refresh tokens** (short-lived access tokens)

5.2 Rate Limiting Implementation

```
const rateLimit = require('express-rate-limit'); // Create limiter for authentication
endpoints const authLimiter = rateLimit({ windowMs: 15 * 60 * 1000, // 15 minutes max: 5, //
Limit each IP to 5 requests per window message: 'Too many attempts, please try again later',
standardHeaders: true, // Return rate limit info in headers legacyHeaders: false, }); // Apply
to login and register routes app.post('/login', authLimiter, async (req, res) => { // Login
logic }); app.post('/register', authLimiter, async (req, res) => { // Registration logic });
```

5.3 Input Validation

```
const { body, validationResult } = require('express-validator'); // Validation middleware
const validateRegistration = [ body('email') .isEmail() .normalizeEmail() .withMessage('Valid
email required'), body('password') .isLength({ min: 8 }) .withMessage('Password must be at
least 8 characters') .matches(/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[!@#$%^&*])/)
.withMessage('Password must contain uppercase, lowercase, number, and special character') ];
app.post('/register', validateRegistration, async (req, res) => { // Check for validation
errors const errors = validationResult(req); if (!errors.isEmpty()) { return
res.status(400).json({ errors: errors.array() }); } // Proceed with registration });
```

5.4 Refresh Token Strategy

Why Refresh Tokens?

Problem: If an access token is stolen, the attacker has access until it expires.

Solution: Use short-lived access tokens (15 min) + long-lived refresh tokens (7 days)

How It Works:

1. User logs in → receives both access token & refresh token
2. Access token used for API requests (expires in 15 min)
3. When access token expires → use refresh token to get new access token
4. Refresh token stored securely (httpOnly cookie or secure storage)
5. If refresh token is compromised → revoke it from database

```
// Refresh token implementation app.post('/refresh', async (req, res) => { const {
refreshToken } = req.body; if (!refreshToken) { return res.status(401).json({ error: 'No
refresh token' }); } // Verify refresh token jwt.verify(refreshToken,
process.env.REFRESH_TOKEN_SECRET, async (err, user) => { if (err) { return
res.status(403).json({ error: 'Invalid refresh token' }); } // Check if token is in database
(not revoked) const isValid = await db.checkRefreshToken(refreshToken); if (!isValid) { return
res.status(403).json({ error: 'Token revoked' }); } // Create new access token const
accessToken = jwt.sign( { userId: user.userId }, process.env.JWT_SECRET, { expiresIn: '15m' }
// Short-lived ); res.json({ accessToken }); }); });
```

Part 6: Final Assessment

📝 Knowledge Check: Multiple Choice Questions

Question 1: Password Hashing

Why should you use bcrypt instead of MD5 for password hashing?

a) bcrypt is faster

b) bcrypt includes automatic salting and is computationally expensive (slow)

c) bcrypt produces shorter hashes

d) MD5 is deprecated by Node.js

Answer: b) bcrypt includes automatic salting and is computationally expensive

Explanation: bcrypt is specifically designed for password hashing with built-in salting and intentional slowness to resist brute-force attacks. MD5 is fast and has known vulnerabilities.

Question 2: JWT Structure

Which part of a JWT ensures the token hasn't been tampered with?

a) Header

b) Payload

c) Signature

d) All three parts

Answer: c) Signature

Explanation: The signature is created by hashing the header and payload with a secret key. Any change to the header or payload will invalidate the signature.

Question 3: Session Storage

Where is session data primarily stored in session-based authentication?

a) Client browser (localStorage)

b) JWT token

c) Server-side (database or memory)

d) URL parameters

Answer: c) Server-side (database or memory)

Explanation: Sessions are stateful - the server stores user data and gives the client only a session ID (in a cookie).

Question 4: Cookie Flags

What does the "httpOnly" flag do for cookies?

a) Makes the cookie only work on HTTP (not HTTPS)

b) Prevents JavaScript from accessing the cookie (XSS protection)

c) Makes the cookie faster to transmit

d) Compresses the cookie data

Answer: b) Prevents JavaScript from accessing the cookie

Explanation: httpOnly cookies cannot be read by JavaScript (document.cookie), providing protection against XSS attacks.

Question 5: JWT vs Sessions

What is the main advantage of JWT over sessions?

a) JWT is more secure

b) JWT is stateless (no server-side storage needed)

c) JWT can be easily revoked

d) JWT tokens are smaller

Answer: b) JWT is stateless (no server-side storage needed)

Explanation: JWT's main advantage is that the server doesn't need to store session data, making it easier to scale across multiple servers.

Summary & Decision Guide

🎯 When to Use What?

Use JWT When:

- ✅ Building APIs for mobile apps
- ✅ Microservices architecture
- ✅ Need horizontal scaling
- ✅ Cross-domain authentication required

- ✅ Stateless design preferred

Use Sessions When:

- ✅ Traditional web applications
- ✅ Need immediate token revocation
- ✅ Server-side rendering
- ✅ Sensitive data in session
- ✅ Simpler security model preferred

Best Practice: Hybrid Approach

Many modern apps use **both**:

- Short-lived JWT access tokens (15 min)
- Long-lived refresh tokens stored in httpOnly cookies
- Refresh tokens tracked in database (can be revoked)

🎓 Congratulations!

You now have a comprehensive understanding of Node.js authentication:

- ✅ Password hashing with bcrypt
- ✅ JWT token-based authentication
- ✅ Session-based authentication with cookies
- ✅ Security best practices
- ✅ Real-world implementation patterns

Next Steps:

1. Build a complete authentication system
2. Implement OAuth2 (Google/Facebook login)
3. Learn about two-factor authentication (2FA)
4. Explore password reset flows
5. Study security vulnerabilities (OWASP Top 10)