
Chapter 7: User Authentication

Web applications today almost always require **user authentication**. Whether it's social media, online stores, blogs, or messaging apps, authentication ensures that users can securely **log in, access protected content, and perform actions based on their identity**. Authentication is one of the most important practical skills in web development because it connects the **user to your application** while maintaining **security and privacy**.

In this chapter, you will learn:

- What authentication is and why it matters
- The difference between authentication and authorization
- Common authentication strategies
- How **JSON Web Tokens (JWTs)** work
- Implementing user authentication in Node.js and Express
- Middleware and protecting routes
- Token storage strategies and refresh tokens
- Security best practices and common pitfalls
- Real-world examples including role-based access

By the end of this chapter, you will be able to build **secure, full-featured user authentication** for your applications.

---

1. Understanding Authentication

Authentication is the process of **verifying who a user is**. It's like showing an **ID card** before entering a building: the building knows who you are and grants access accordingly.

Authentication is **different from authorization**, which determines **what a user can do** after they are verified. For example:

- Logging in with a username and password → **Authentication**
- Accessing an admin dashboard → **Authorization**

Why Authentication is Important

1. **Security:** Prevent unauthorized access to sensitive data.
2. **Personalization:** Display user-specific content and preferences.
3. **Accountability:** Track user actions in the system.
4. **Business Logic:** Enable features like order history, messaging, and dashboards.

Without authentication, your server cannot distinguish between users, making dynamic applications **unsafe and impractical**.

---

2. Common Authentication Methods

Authentication can be implemented in multiple ways:

A. Session-Based Authentication

- The server stores a session for each logged-in user.
- Client stores a **session ID** in cookies and sends it with each request.
- Pros: Simple, easy to implement.
- Cons: Less scalable; sessions consume server memory.

B. Token-Based Authentication (JWT)

- The server issues a **token** after a successful login.
- Client includes the token in the HTTP header for each request.
- Pros: Stateless, scalable, works with modern front-end frameworks and mobile apps.
- Cons: Token management (expiration, revocation) requires attention.

In this chapter, we focus on **JWTs (JSON Web Tokens)**, which are **widely used, secure, and scalable**.

---

3. How JWTs Work

JWTs are **compact, URL-safe tokens** that allow servers to verify users **without storing session data**.

JWT Structure

A JWT consists of **three parts**, separated by dots:

header.payload.signature

1. **Header:** Contains metadata about the token and algorithm.
   Example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2. **Payload:** Contains claims, usually user information like ID, username, and role.
   Example:

```
{
  "id": "12345",
  "username": "john_doe",
  "role": "user"
}
```

3. **Signature:** Ensures token integrity. Generated by combining the header and payload with a **secret key**:

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

The server can **verify** the token by checking the signature. If valid, the user is authenticated.

---

4. Advantages of JWT Authentication

1. **Stateless:** Server does not need to store session data.
2. **Scalable:** Works well with distributed systems and multiple servers.
3. **Secure:** Token signature ensures integrity.
4. **Flexible:** Can include user roles, expiration, and custom claims.
5. **Standardized:** Supported across multiple platforms and programming languages.

---

5. Setting Up User Authentication in Node.js

To implement JWT authentication, we need:

- Node.js and Express server
- MongoDB database to store users
- `bcrypt` for password hashing
- `jsonwebtoken` to generate and verify tokens

Step 1: Install Dependencies

npm install express mongoose bcrypt jsonwebtoken body-parser dotenv

---

Step 2: Define the User Model

```
const mongoose = require('mongoose');
const bcrypt = require('bcrypt');

const userSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  role: { type: String, default: 'user' }
});

// Hash password before saving
userSchema.pre('save', async function(next) {
```

```
  if (!this.isModified('password')) return next();
  this.password = await bcrypt.hash(this.password, 10);
  next();
});

// Compare password
userSchema.methods.comparePassword = function(password) {
  return bcrypt.compare(password, this.password);
};

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

Explanation:

- `pre('save')`: Hashes the password before saving.
- `comparePassword`: Verifies password during login.
- `role`: Can be `user` or `admin`, used for role-based access.

Step 3: User Registration Endpoint

```
const express = require('express');
const bodyParser = require('body-parser');
const User = require('./models/User');

const app = express();
app.use(bodyParser.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 (err) {
    res.status(400).json({ error: err.message });
  }
});
```

- Stores hashed passwords in MongoDB.

- Returns a confirmation message after registration.

Step 4: User Login Endpoint

```
const jwt = require('jsonwebtoken');
const SECRET_KEY = process.env.SECRET_KEY || 'mysecretkey';

app.post('/login', async (req, res) => {
  const { email, password } = req.body;
  try {
    const user = await User.findOne({ email });
    if (!user) return res.status(404).json({ message: 'User not found' });

    const isMatch = await user.comparePassword(password);
    if (!isMatch) return res.status(401).json({ message: 'Invalid credentials' });

    const token = jwt.sign(
      { id: user._id, username: user.username, role: user.role },
      SECRET_KEY,
      { expiresIn: '1h' }
    );

    res.json({ message: 'Login successful', token });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

- Generates a JWT with `id`, `username`, and `role`.
- Token is valid for 1 hour.

---

6. Protecting Routes with Middleware

Middleware ensures that only authenticated users can access certain endpoints:

```
const authenticate = (req, res, next) => {
  const token = req.headers['authorization'];
  if (!token) return res.status(401).json({ message: 'Access denied' });

  try {
```

```
    const decoded = jwt.verify(token, SECRET_KEY);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).json({ message: 'Invalid token' });
  }
};

app.get('/profile', authenticate, (req, res) => {
  res.json({ message: 'This is your profile', user: req.user });
});
```

- Middleware verifies JWT before granting access.
- If invalid or expired, access is denied.

---

7. Role-Based Access Control

Roles allow restricting access to certain users:

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

// Example route: Admin only
app.get('/admin-dashboard', authenticate, authorize(['admin']), (req, res) => {
  res.json({ message: 'Welcome Admin!' });
});
```

- `authorize` middleware checks user role before allowing access.

---

8. Token Storage Strategies

Tokens must be stored securely:

1. **LocalStorage**: Easy but vulnerable to XSS attacks.

2. **HttpOnly Cookies**: More secure; JavaScript cannot access cookies.
3. **Memory (SPA)**: Stored in memory, disappears on refresh (good for short-lived tokens).

---

9. Refresh Tokens

Access tokens should be short-lived. Refresh tokens allow users to **get new access tokens without logging in again**:

- Store refresh token securely in HttpOnly cookies.
- When access token expires, client requests a new token using the refresh token.
- Server validates refresh token and issues a new access token.

**10. Security

Best Practices**

1. Never store passwords in plain text.
2. Use HTTPS to encrypt data in transit.
3. Use strong, unpredictable secret keys.
4. Implement token expiration and rotation.
5. Validate and sanitize user inputs.
6. Handle errors without exposing sensitive information.
7. Regularly audit authentication logic for vulnerabilities.

---

11. Real-World Example: Messaging App

- Users register and log in.

- JWT allows them to access their messages and send new messages.
- Admin role can view all messages, delete inappropriate content, or manage users.
- Front-end sends JWT in request headers to access protected endpoints.

---

12. Common Pitfalls

1. **Storing tokens in unsafe locations** → use HttpOnly cookies.
2. **Not hashing passwords** → always use bcrypt.
3. **No token expiration** → increases risk of misuse.
4. **Weak secrets** → easily guessable JWTs.
5. **Not validating inputs** → security vulnerabilities.

---

13. Summary

- Authentication identifies users; authorization determines what they can do.
- JWTs provide a **stateless, scalable, and secure** method.
- Passwords must always be hashed; tokens must be stored securely.
- Middleware protects routes and ensures role-based access.
- Refresh tokens allow long sessions while keeping access tokens short-lived.
- Following best practices ensures your authentication system is secure and reliable.

By mastering JWT authentication, you can now build **real-world applications with secure login, role-based access, and protected routes**, fully bridging the gap between front-end and back-end functionality.
