Express Authentication: Cookies, Sessions & JWT

Table of Contents

Cookie-Based Authentication

What are Cookies?

Cookies are small pieces of data stored in the user's browser and sent with every HTTP request to the same domain. They're the foundation for maintaining state in stateless HTTP communication.

Basic Cookie Authentication Flow

```javascript
```

```
// Setting a cookie
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  if (validateUser(username, password)) {
    // Set a simple authentication cookie
    res.cookie('authenticated', 'true', {
      maxAge: 24 * 60 * 60 * 1000, // 24 hours
      httpOnly: true,
      secure: true, // HTTPS only
      sameSite: 'strict'
    });
    res.json({ message: 'Login successful' });
  } else {
    res.status(401).json({ message: 'Invalid credentials' });
  }
});

// Reading cookies (with cookie-parser middleware)
app.use(cookieParser());
app.get('/protected', (req, res) => {
  if (req.cookies.authenticated === 'true') {
    res.json({ message: 'Access granted' });
  } else {
    res.status(401).json({ message: 'Unauthorized' });
  }
});
```

Cookie Security Attributes

- **httpOnly**: Prevents client-side JavaScript access (XSS protection)
- **secure**: Only sent over HTTPS connections
- **sameSite**: CSRF protection ('strict', 'lax', or 'none')
- **maxAge/expires**: Cookie expiration time
- **domain**: Specifies which domains can access the cookie
- **path**: Specifies which paths can access the cookie

Session-Based Authentication

How Sessions Work

Session-based authentication stores user data on the server and uses a session ID cookie to link the client to their server-side session data.

Session Flow

1. User logs in with credentials
2. Server validates credentials
3. Server creates a session object with user data
4. Server generates a unique session ID
5. Session ID is sent to client as a cookie
6. Client includes session ID cookie in subsequent requests
7. Server uses session ID to retrieve user data from session store

Basic Express Session Implementation

```javascript
```

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

// Session configuration
app.use(session({
  secret: process.env.SESSION_SECRET, // Used to sign session ID
  name: 'sessionId', // Cookie name (default: 'connect.sid')
  resave: false, // Don't save unchanged sessions
  saveUninitialized: false, // Don't create sessions for unauthenticated users
  cookie: {
    secure: process.env.NODE_ENV === 'production', // HTTPS in production
    httpOnly: true,
    maxAge: 24 * 60 * 60 * 1000 // 24 hours
  },
  store: MongoStore.create({
    mongoUrl: process.env.MONGODB_URI,
    touchAfter: 24 * 3600 // Lazy session update
  })
}));

// Login endpoint
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  if (validateUser(username, password)) {
    req.session.userId = user.id;
    req.session.username = user.username;
    req.session.role = user.role;
    res.json({ message: 'Login successful' });
  } else {
    res.status(401).json({ message: 'Invalid credentials' });
  }
});

// Protected route middleware
const requireAuth = (req, res, next) => {
  if (req.session.userId) {
    next();
  } else {
    res.status(401).json({ message: 'Unauthorized' });
  }
};

app.get('/profile', requireAuth, (req, res) => {
  res.json({
    userId: req.session.userId,
```

```
    username: req.session.username,
    role: req.session.role
  });
});

// Logout
app.post('/logout', (req, res) => {
  req.session.destroy((err) => {
    if (err) {
      return res.status(500).json({ message: 'Could not log out' });
    }
    res.clearCookie('sessionId');
    res.json({ message: 'Logged out successfully' });
  });
});
```

Session Storage Options

- **Memory Store** (default): Not suitable for production
- **Redis**: Fast, scalable option for session storage
- **MongoDB**: Document-based session storage
- **PostgreSQL/MySQL**: Relational database storage

```
javascript

// Redis store example
const RedisStore = require('connect-redis')(session);
const redis = require('redis');
const client = redis.createClient();

app.use(session({
  store: new RedisStore({ client: client }),
  secret: process.env.SESSION_SECRET,
  // ... other options
}));
```

JWT (JSON Web Tokens)

JWT Structure

JWTs consist of three parts separated by dots:

- **Header**: Algorithm and token type
- **Payload**: Claims (user data)
- **Signature**: Verification signature

JWT Authentication Flow

javascript

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

// Login and generate JWT
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  if (await validateUser(username, password)) {
    const payload = {
      userId: user.id,
      username: user.username,
      role: user.role
    };

    const token = jwt.sign(
      payload,
      process.env.JWT_SECRET,
      { expiresIn: '24h' }
    );

    // Send as HTTP-only cookie (more secure than localStorage)
    res.cookie('token', token, {
      httpOnly: true,
      secure: process.env.NODE_ENV === 'production',
      sameSite: 'strict',
      maxAge: 24 * 60 * 60 * 1000
    });

    res.json({ message: 'Login successful' });
  } else {
    res.status(401).json({ message: 'Invalid credentials' });
  }
});

// JWT Verification Middleware
const verifyJWT = (req, res, next) => {
  const token = req.cookies.token || req.headers.authorization?.split(' ')[1];

  if (!token) {
    return res.status(401).json({ message: 'No token provided' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
```

```
    res.status(401).json({ message: 'Invalid token' });
  }
};

app.get('/profile', verifyJWT, (req, res) => {
  res.json({
    userId: req.user.userId,
    username: req.user.username,
    role: req.user.role
  });
});
```

JWT Refresh Token Pattern

javascript

```
// Generate both access and refresh tokens
app.post('/login', async (req, res) => {
  if (await validateUser(username, password)) {
    const accessToken = jwt.sign(payload, process.env.JWT_SECRET, { expiresIn: '15m' });
    const refreshToken = jwt.sign(payload, process.env.REFRESH_SECRET, { expiresIn: '7d' });

    // Store refresh token in database
    await storeRefreshToken(user.id, refreshToken);

    res.cookie('accessToken', accessToken, { httpOnly: true, maxAge: 15 * 60 * 1000 });
    res.cookie('refreshToken', refreshToken, { httpOnly: true, maxAge: 7 * 24 * 60 * 60 * 1000 });

    res.json({ message: 'Login successful' });
  }
});

// Refresh endpoint
app.post('/refresh', async (req, res) => {
  const refreshToken = req.cookies.refreshToken;

  try {
    const decoded = jwt.verify(refreshToken, process.env.REFRESH_SECRET);
    const isValid = await validateRefreshToken(decoded.userId, refreshToken);

    if (isValid) {
      const newAccessToken = jwt.sign(
        { userId: decoded.userId, username: decoded.username },
        process.env.JWT_SECRET,
        { expiresIn: '15m' }
      );

      res.cookie('accessToken', newAccessToken, { httpOnly: true, maxAge: 15 * 60 * 1000 });
      res.json({ message: 'Token refreshed' });
    }
  } catch (error) {
    res.status(401).json({ message: 'Invalid refresh token' });
  }
});
```

Express Session Middleware Deep Dive

How Express Session Sets Session ID as Cookie

The Express session middleware follows this process:

1. **Request Processing**:

```javascript
// When a request comes in, express-session:
app.use(session({
  secret: 'your-secret-key',
  name: 'connect.sid', // This becomes the cookie name
  // ... other options
}));
```

2. **Session ID Generation**:

```javascript
// Internal process (simplified):
const generateSessionId = () => {
  // Uses 'uid-safe' library by default
  return require('uid-safe').sync(24); // Generates cryptographically secure ID
};
```

3. **Cookie Setting Process**:

```javascript
```

```
// Middleware logic (conceptual):
function sessionMiddleware(req, res, next) {
  let sessionId = req.cookies[options.name]; // Check for existing session ID

  if (!sessionId) {
    // Generate new session ID
    sessionId = generateSessionId();
  }

  // Load session data from store
  store.get(sessionId, (err, sessionData) => {
    if (sessionData) {
      req.session = sessionData;
    } else {
      req.session = {}; // Create new session object
    }

    // Set session ID cookie
    res.cookie(options.name, sessionId, {
      httpOnly: options.cookie.httpOnly,
      secure: options.cookie.secure,
      maxAge: options.cookie.maxAge,
      // ... other cookie options
    });

    next();
  });
}
```

Session Lifecycle

1. **First Request (No Session)**:
   - No session cookie found
   - Generate new session ID
   - Create empty session object
   - Set session ID cookie in response
2. **Subsequent Requests**:
   - Read session ID from cookie
   - Load session data from store using ID
   - Attach session data to `req.session`
3. **Session Modification**:

```
javascript
```

```
app.post('/login', (req, res) => {
  // Modify session data
  req.session.userId = user.id;
  // Middleware automatically saves to store and updates cookie
});
```

4. **Session Saving**:
   - Modified sessions are automatically saved to the store
   - Cookie is updated if needed (new expiration, etc.)

Session Store Integration

javascript

```
// Custom session store implementation
class CustomSessionStore extends session.Store {
  get(sid, callback) {
    // Retrieve session data by session ID
    database.findSession(sid)
      .then(data => callback(null, data))
      .catch(err => callback(err));
  }

  set(sid, sessionData, callback) {
    // Store session data with session ID
    database.saveSession(sid, sessionData)
      .then(() => callback())
      .catch(err => callback(err));
  }

  destroy(sid, callback) {
    // Delete session from store
    database.deleteSession(sid)
      .then(() => callback())
      .catch(err => callback(err));
  }
}
```

Comparison and Best Practices

Session-Based vs JWT Authentication

Aspect	Sessions	JWT
Storage	Server-side	Client-side
Scalability	Requires shared storage	Stateless, more scalable
Security	Server controls all data	Data embedded in token
Revocation	Immediate	Requires blacklist or short expiry
Payload Size	Minimal cookie	Larger tokens

Security Best Practices

1. **Cookie Security**:

```javascript
app.use(session({
  cookie: {
    httpOnly: true,       // Prevent XSS
    secure: true,         // HTTPS only
    sameSite: 'strict',   // CSRF protection
    maxAge: 24 * 60 * 60 * 1000 // Reasonable expiry
  }
}));
```

2. **Session Management**:

```javascript
// Session cleanup
app.use('/admin', (req, res, next) => {
  if (req.session.role !== 'admin') {
    req.session.destroy(); // Clean up unauthorized session
    return res.status(403).json({ message: 'Forbidden' });
  }
  next();
});
```

3. **JWT Security**:
   - Use strong secrets
   - Implement token refresh
   - Consider token blacklisting
   - Store in httpOnly cookies when possible

When to Use Each Approach

Use Sessions When:

- Building traditional web applications
- Need immediate session revocation
- Have single server or shared session store
- Prioritize server-side security control

Use JWT When:

- Building APIs or microservices
- Need stateless authentication
- Have distributed systems
- Mobile app authentication

Hybrid Approach

javascript

```javascript
// Combine both for maximum flexibility
app.use(session({
  // Session configuration
}));

const authMiddleware = (req, res, next) => {
  // Try JWT first
  const token = req.cookies.token;
  if (token) {
    try {
      req.user = jwt.verify(token, process.env.JWT_SECRET);
      return next();
    } catch (err) {
      // Fall through to session check
    }
  }

  // Check session
  if (req.session.userId) {
    req.user = { userId: req.session.userId };
    return next();
  }

  res.status(401).json({ message: 'Unauthorized' });
};
```

This comprehensive approach allows you to choose the right authentication method based on your specific requirements while maintaining security best practices.