Advanced Authentication

Content

- Introduction To OAuth2 & Passport.js
- OAuth2 Grant Types
- Passport.js Google Strategy
- Facebook Login Integration
- GitHub Login Integration
- Security Best Practices

Introduction To OAuth2

OAuth2

- **What is OAuth 2.0?**
  - OAuth 2.0 is an authorization framework that allows third-party applications to obtain limited access to user accounts on an HTTP service.
  - It delegates user authentication to the service that hosts the user account.
  - Common providers: Google, Facebook, GitHub, Twitter, etc.
  - The application never sees the user’s password.

OAuth2

- **Why Use Third-Party Authentication?**
  - Improved Security: Users don’t create new passwords; authentication is handled by trusted providers.
  - Better UX: One-click login using existing Google/Facebook/GitHub accounts.
  - Fewer Support Tickets: No need to manage forgotten passwords or password reset flows.
  - Trust Factor: Users trust large providers with secure authentication.

Passport.js

- **What is Passport.js?**
  - A popular authentication middleware for Node.js.
  - Supports multiple strategies: Local, OAuth, JWT, SAML, etc.
  - Passport.js simplifies the integration of OAuth 2.0 (and other authentication strategies) into web applications.
  - Handles:
    - Initializing authentication
    - Redirecting to providers
    - Managing sessions or stateless auth (JWT)
    - Strategy-based: You plug in strategies like passport-google-oauth20, passport-facebook, etc.

OAuth2 & Passport.js Workflow

- If you want to enable users to log in to your application using their Google accounts, you would:
  - Register your application with Google:
    - You'll get a client ID and client secret from Google, and configure a callback URL.
  - Use passport-google-oauth20 in your application:
    - Configure the strategy with your client ID, client secret, and callback URL.
  - Implement authentication routes:
    - Passport.js handles the redirect to Google's authentication page and then processes the callback from Google, verifying the user and issuing a session for the user.
  - Protect routes:
    - You can use Passport.js's **isAuthenticated()** function to protect routes that require authentication.

OAuth2
Grant Types

OAuth2 Grant Types

- OAuth 2.0 defines several grant types, which are different ways that a client application can obtain an access token to access protected resources. The most common OAuth 2.0 grant types includes:

  1. Authorization Code Grant
  2. Implicit Grant
  3. Resource Owner Password Credentials Grant
  4. Client Credentials Grant
  5. Refresh Tokens Grant

Authorization Code Grant

- This is the most common and recommended grant type for web applications and native apps. It involves redirecting the user to the authorization server, where they authenticate and grant consent, and then receiving an authorization code that the client exchanges for an access token.
- Used by Google, Facebook, GitHub, and others.
- Involves a server-side exchange of an authorization code for an access token.
- Requires both client ID and client secret.

Authorization Code Grant

- **Workflow:**
  - User clicks “Login with Google”
  - Redirects to Google for login & consent
  - Google redirects back to your app with code
  - Server exchanges code for access token (and refresh token)
- **Use Case:** Web apps with a backend server

Implicit Grant

- This grant type was once used for single-page applications and mobile apps, but it's now generally discouraged due to security concerns. It directly returns an access token without the intermediate authorization code step.

- Designed for single-page apps (SPA) with no backend.
- The access token is returned directly in the URL fragment.
- No refresh tokens supported.

- Security Issues:
  - Access token is exposed in browser history
  - Vulnerable to token leakage via XSS
  - No secure token storage in frontend

ROPC Grant

- **Resource Owner Password Credentials Grant:**
  - This grant type involves the client directly requesting an access token using the user's username and password. It is generally discouraged due to security risks and should only be used in specific trusted scenarios.
  - No redirection or user consent screens.
  - User credentials are handled by the client, making it insecure.
  - App gains full access to user credentials
  - Insecure if used in public clients like SPAs or mobile apps

ROPC Grant

- **Workflow:**
  - App sends username/password to Auth Server then it receives the access token
- **When to Use (Not Recommended):**
  - Legacy systems where OAuth2 wasn't originally designed.
  - Trusted apps only (e.g. internal tools).
  - Never for third-party or public clients.

Client Credentials Grant

- This grant type is used for machine-to-machine communication, where the client application acts on its own behalf without involving a user. It's suitable for server-to-server interactions or for accessing resources that don't require user context.
- The client authenticates with the provider using client ID and secret, and receives an access token.
- No user interaction involved.
- Use Case:
    - Internal server-to-server communication (Protected APIs)
    - Admin dashboards or background workers

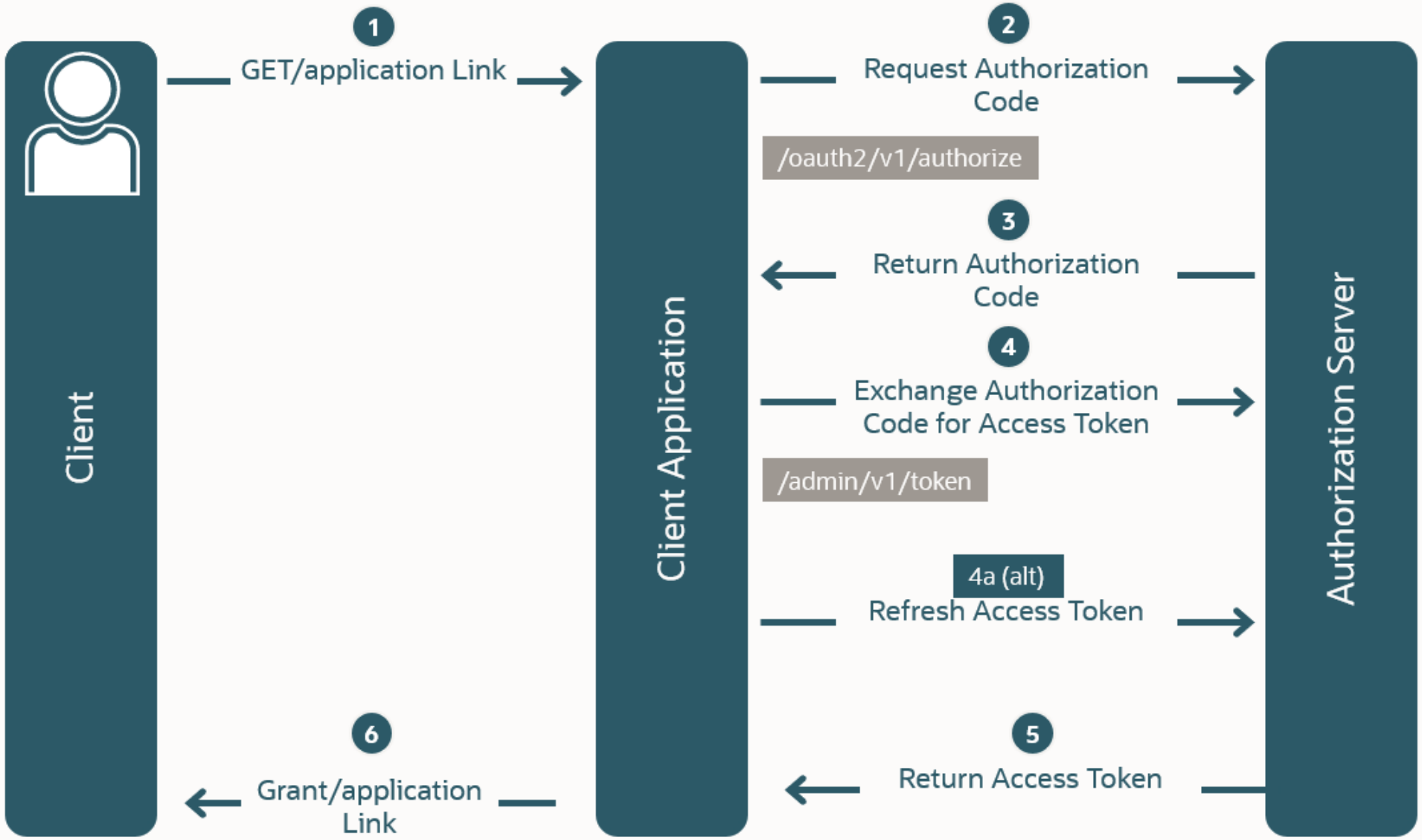
Refresh Tokens Grant

- This grant type is used to obtain a new access token when the original access token expires. The client uses a refresh token, which was previously issued during the authorization code, to request a new access token.
- Refresh tokens are long-lived and used to obtain new access tokens without re-login
- Issued only in Authorization Code Grant
- Refresh tokens should be stored securely (e.g. httpOnly cookies)

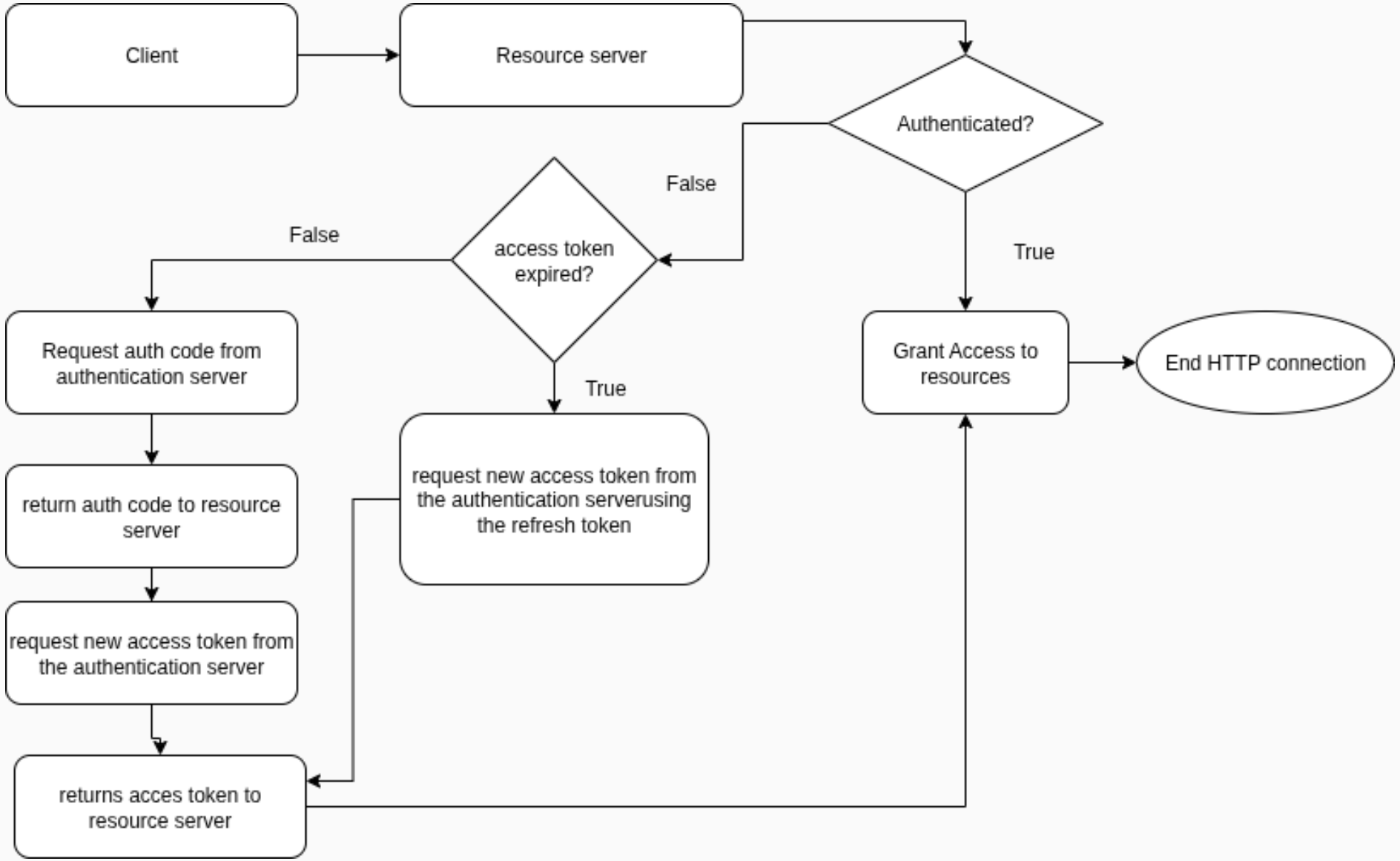
Refresh Tokens Grant

- **Workflow:**
  - Auth provider issues access token + refresh token
  - When access token expires, app sends refresh token to get a new access token
  - Now every time the client gets an unauthorized status, an API call with the refresh token is automatically sent to the authentication server in the background and user is logged in with the new access token
- Best Practice: Rotate & invalidate refresh tokens and store securely

OAuth2


Client
Client Application
Authorization Server
1
GET/application Link
2
Request Authorization Code
/oauth2/v1/authorize
3
Return Authorization Code
4
Exchange Authorization Code for Access Token
/admin/v1/token
4a (alt)
Refresh Access Token
5
Return Access Token
6
Grant/application Link

OAuth2


Client
Resource server
Authenticated?
False
True
access token expired?
False
True
Request auth code from authentication server
return auth code to resource server
request new access token from the authentication server
returns acces token to resource server
request new access token from the authentication serverusing the refresh token
Grant Access to resources
End HTTP connection

Passport.js + Google Strategy

Passport.js Sessions

- Passport.js works with express sessions as an authentication strategy
- First, start by installing the required node modules:

```
npm install passport express-session
```

- passport: Core middleware for authentication
- express-session: Required for session-based authentication
- Each provider (Google, Facebook, GitHub) has a separate OAuth2 strategy

Passport.js Sessions

- express-session must be configured before initializing passport
- Passport needs both .initialize() and .session() middleware

```
1  import express from 'express';
2  import session from 'express-session';
3  import passport from 'passport';
4
5  const app = express();
6
7  app.use(session({
8    secret: 'yourSecret',
9    resave: false,
10   saveUninitialized: true
11 }));
12
13 app.use(passport.initialize());
14 app.use(passport.session());
```

Passport.js + Google Strategy

- Now, start by configuring your strategy:

```
passport.use(new GoogleStrategy({
  clientID: process.env.GOOGLE_CLIENT_ID, // From Google Cloud Console
  clientSecret: process.env.GOOGLE_CLIENT_SECRET,
  callbackURL: '/auth/google/callback'    // Where Google redirects after login
}, async (accessToken, refreshToken, profile, done) => {
  let user = await User.findOne({ googleId: profile.id });  // Check if user already exists
  if (!user) {
    user = await User.create({           // Save new user if first time login
      googleId: profile.id,
      displayName: profile.displayName
    });
  }
  return done(null, user);              // Pass user to Passport
}));
```

Passport.js + Google Strategy

- To store users info in the session, start by serializing and deserializing your users:

```
passport.serializeUser((user, done) => {
  done(null, user.id);                    // Save user ID in the session cookie
});

passport.deserializeUser(async (id, done) => {
  const user = await User.findById(id); // Fetch user info from DB
  done(null, user);                       // Attach user to req.user
});
```

Passport.js + Google Strategy

- To start a Google login, simply add you login API and redirect them to a google login:

```
app.get('/auth/google',
  passport.authenticate('google', { scope: ['profile', 'email'] })
);
```

- Google OAuth2 will use your predefined callback URL to redirect after a successful login

```
app.get('/auth/google/callback',
  passport.authenticate('google', { failureRedirect: '/login' }),
  (req, res) => {
    res.json({ message: 'Logged in with Google', user: req.user });
  }
);
```

Passport.js + Google Strategy

- You can create a middleware to use on protected routes to check if a user is authenticated:

```
function ensureAuth(req, res, next) {
  if (req.isAuthenticated()) return next();
  res.status(401).json({ error: 'Unauthorized' });
}
```

```
app.get('/me', ensureAuth, (req, res) => {
  res.json({ user: req.user });
});
```

Passport.js + Google Strategy

- To logout a user, simply clear the session cookie using:

```
app.get('/logout', (req, res) => {
  req.logout(() => {
    res.json({ message: 'Logged out' });
  });
});
```

Passport.js JWT

- Passport.js also works with JWT for authentication strategies
- First, start by installing the required node modules:

```
1  npm install passport jsonwebtoken
```

- passport: Core middleware for authentication
- jsonwebtoken: Required for JWT-based authentication

Passport.js + Google Strategy

- Update your callback to start signing JWT to the user after a successful login

```
app.get('/auth/google/callback',
  passport.authenticate('google', { session: false, failureRedirect: '/login' }),
  (req, res) => {
    const user = req.user;
    const token = jwt.sign(
      { id: user._id, name: user.displayName },
      process.env.JWT_SECRET,
      { expiresIn: '1h' }
    );

    res.json({
      message: 'Login successful',
      token,
      user: { id: user._id, name: user.displayName }
    });
  }
);
```

Passport.js + Google Strategy

- Then add a middleware to verify the submitted token

```
function verifyToken(req, res, next) {
  const authHeader = req.headers.authorization;
  if (!authHeader?.startsWith('Bearer ')) {
    return res.status(401).json({ message: 'No token provided' });
  }

  const token = authHeader.split(' ')[1];
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).json({ message: 'Invalid token' });
  }
}
```

Passport.js + Google Strategy

- Now for your protected routes, start integrating this middleware

```
app.get('/profile', verifyToken, (req, res) => {
  res.json({
    message: 'Protected profile info',
    user: req.user
  });
});
```

Facebook Login Integration

Passport.js + Facebook Strategy

- Passport.js supports Facebook login strategies
- First, start by installing the required node modules:

```
npm install passport passport-facebook
```

- passport: Core middleware for authentication
- passport-facebook: Required for implementing the Facebook Strategy

Passport.js + Facebook Strategy

- Then start by configuring your facebook strategy

```
passport.use(new FacebookStrategy(
  {
    clientID: process.env.FACEBOOK_APP_ID,
    clientSecret: process.env.FACEBOOK_APP_SECRET,
    callbackURL: process.env.FACEBOOK_CALLBACK,
    profileFields: ['id', 'displayName', 'emails']
  },
  async (accessToken, refreshToken, profile, done) => {
    const existingUser = await User.findOne({ facebookId: profile.id })
    if (existingUser) return done(null, existingUser)

    const newUser = await User.create({
      facebookId: profile.id,
      name: profile.displayName,
      email: profile.emails?.[0]?.value || ''
    })

    done(null, newUser)
  }
))
```

Passport.js + Facebook Strategy

- Then start implementing your login routes & JWT/sessions:

```
app.get('/auth/facebook',
  passport.authenticate('facebook', { scope: ['email'] })
)

app.get('/auth/facebook/callback',
  passport.authenticate('facebook', { session: false, failureRedirect: '/login' }),
  (req, res) => {
    const token = jwt.sign({ id: req.user._id, name: req.user.name }, process.env.JWT_SECRET, { expiresIn: '1h' })
    res.json({ token, user: req.user })
  }
)
```

Passport.js + Facebook Strategy

- Logged in users can now access protected routes using basic JWT flow authentication:

```
app.get('/profile', async (req, res) => {
  const authHeader = req.headers.authorization
  if (!authHeader) return res.status(401).json({ message: 'Missing token' })

  try {
    const decoded = jwt.verify(authHeader.split(' ')[1], process.env.JWT_SECRET)
    const user = await User.findById(decoded.id)
    res.json({ user })
  } catch (err) {
    res.status(401).json({ message: 'Invalid or expired token' })
  }
})
```

Github Login Integration

Passport.js + Github Strategy

- Passport.js also supports Github login strategies
- First, start by installing the required node modules:

```
1  npm install passport passport-github2
```

- passport: Core middleware for authentication
- passport-github2: Required for implementing the Github Strategy

Passport.js + Github Strategy

- Then start by configuring your github strategy

```
passport.use(new GitHubStrategy({
  clientID: process.env.GITHUB_CLIENT_ID,
  clientSecret: process.env.GITHUB_CLIENT_SECRET,
  callbackURL: process.env.GITHUB_CALLBACK,
  scope: ['user:email']
}, async (accessToken, refreshToken, profile, done) => {
  const existingUser = await User.findOne({ githubId: profile.id })
  if (existingUser) return done(null, existingUser)

  const email = profile.emails?.[0]?.value || ''

  const newUser = await User.create({
    githubId: profile.id,
    username: profile.username,
    email
  })

  done(null, newUser)
}))
```

Passport.js + Github Strategy

- Then start implementing your login routes & JWT/sessions:

```
app.get('/auth/github',
  passport.authenticate('github', { scope: ['user:email'] })
)

app.get('/auth/github/callback',
  passport.authenticate('github', { session: false, failureRedirect: '/login' }),
  (req, res) => {
    const token = jwt.sign({ id: req.user._id, username: req.user.username }, process.env.JWT_SECRET, { expiresIn: '1h' })
    res.json({ token, user: req.user })
  }
)
```

Passport.js + Github Strategy

- Logged in users can now access protected routes using basic JWT flow authentication:

```
app.get('/profile', async (req, res) => {
  const authHeader = req.headers.authorization
  if (!authHeader) return res.status(401).json({ message: 'Missing token' })

  try {
    const decoded = jwt.verify(authHeader.split(' ')[1], process.env.JWT_SECRET)
    const user = await User.findById(decoded.id)
    res.json({ user })
  } catch (err) {
    res.status(401).json({ message: 'Invalid or expired token' })
  }
})
```

OAuth Security

OAuth Security

1. **Use HTTPS in Production:**

    - Why it's important:

        - OAuth2 redirects and tokens go through the browser.
        - Without HTTPS, access tokens can be intercepted via man-in-the-middle attacks.

    - How to do it:

        - Use SSL certificates (e.g. via Let's Encrypt).
        - On services like Heroku, Vercel, or Netlify — HTTPS is enabled by default.
        - In your Express app, redirect all HTTP to HTTPS

OAuth Security

2. **Keep OAuth Secrets Safe:**

    - Never commit secrets (client_id, client_secret) to GitHub.
    - Store in .env files and load using dotenv.

    - Use secure secret managers for cloud environments:

        - AWS Secrets Manager
        - Google Secret Manager
        - GitHub Actions secrets

OAuth Security

3. **CSRF Protection:**

   - OAuth2 login flows involve redirects, which can be vulnerable to Cross-Site Request Forgery (CSRF).
   - Passport handles some CSRF internally using the state parameter
   - Use csurf middleware for form submissions or JWT protected routes.
   - Use SameSite Strict and httpOnly flags on cookies

OAuth Security

4. **Secure Logout Strategy:**
    - Simply destroying the session is not enough in token-based systems.
    - Users should be fully logged out across all clients
    - Delete the token on the client side (cookie or localStorage).
    - Invalidate refresh tokens in DB or cache (Redis)
    - Optional: implement a blacklist of tokens.

Any Questions?