React Router OAuth2 Handbook

Sergio Xalambrí

2025-05-03

Contents

Introduction

User authentication and authorization are essential for most web applications. OAuth2 is the most widely used protocol for handling authorization, but it's also known for being complex and tricky to implement correctly.

This handbook will help you understand how OAuth2 works and how to integrate it into a web application. You'll build a secure, production-ready implementation using React Router as the example framework.

While the examples in this book use React Router, the concepts apply to any framework or platform—whether you're building a Next.js app, a mobile client, or even a backend with Laravel or Rails.

We'll walk through each step of the OAuth2 Authorization Code Flow with PKCE, using a simple application that authenticates users and fetches contact data from a remote API.

If you're not familiar with React Router, check out the official documentation and follow their Address Book tutorial. This tutorial serves as the base for the example app. Once you're comfortable with the basics, return here to learn how to add OAuth2.

OAuth2 in Simple Terms

Before implementing OAuth2, it's important to understand the core ideas behind it. This guide won't go into every technical detail, but it will give you the foundation you need to follow along and understand the code.

You'll learn how the Authorization Code Flow with PKCE works by building it step by step.

Why OAuth2?

OAuth2 was created to solve a common problem: how can applications securely access a user's data stored in another service?

Imagine you're building a task management app that integrates with Google Calendar to create events. In the past, you'd have to ask users for their Google credentials. That approach had serious problems:

- **Security risks**: Your app had full access to the user's account.
- **Storing passwords**: You had to store and protect user credentials.
- **Bad UX**: Users needed to re-enter credentials every time they changed their password.

OAuth2 solves all of this by introducing a secure, standardized flow. Instead of asking for passwords, your app requests authorization. If the user approves, the app receives an **access token** that grants limited access to their data.

OAuth2 improves security and user experience by:

1. **Protecting credentials**: Your app never sees the user's password.
2. **Providing limited access**: Apps get permission for specific actions only.
3. **Giving users control**: They can revoke access at any time.

How OAuth2 Works

OAuth2 allows a **Client Application** to access a **Resource Server** on behalf of a **Resource Owner**, using a token issued by an **Authorization Server**. The protocol defines a flow that ensures secure access without exposing sensitive credentials.

Let's walk through a simplified version of this process:

1. The user (Resource Owner) authorizes the application.
2. The Authorization Server issues an Access Token.
3. The Client Application uses this token to access the Resource Server.
4. The Access Token eventually expires.
5. If a Refresh Token was provided, the application can use it to get a new Access Token without asking the user to log in again.
6. Permissions are defined using scopes, which control what the application can do with the token.

Imagine your app is like a personal assistant that needs temporary access to a user's calendar. Instead of giving it full control (like a password would), the user grants specific, limited access through OAuth2.

The Authorization Server issues a key (Access Token) that only works for certain tasks (like reading events), and for a limited time. Once it expires, the app can ask for a new one using the Refresh Token—if the user initially allowed it.

This way, OAuth2 provides access **without sharing passwords**, **limits what the app can do**, and **lets users revoke access at any time**.

OpenID Connect

OAuth2 was designed for authorization—it lets applications act on behalf of a user. But it doesn't define how to identify who the user is. That's where **OpenID Connect (OIDC)** comes in.

OIDC is an authentication layer built on top of OAuth2. It introduces a new type of token called the **ID Token**, which contains identity information about the user, such as their name, email, and unique ID.

This ID Token is usually returned alongside the Access Token when the user logs in. It's formatted as a JWT and includes claims that let the Client Application know who the user is—without needing to make another request.

For example, when someone clicks "Sign in with Google," the app receives an ID Token from Google. Even if it never accesses the user's calendar or contacts, the ID Token confirms who the user is.

If you're building an app that needs both authentication (who the user is) and authorization (what the app can access), OpenID Connect lets you do both in a single flow.

Actors in OAuth2

OAuth2 introduces four main actors that interact during the flow. Understanding each of their roles helps clarify how the protocol works.

- **Resource Owner** – The person or entity that owns the data.
- **Client Application** – The app that wants access to that data.
- **Resource Server** – The API or service that holds the data.
- **Authorization Server** – The system that verifies users and issues tokens.

These roles are always the same, no matter what kind of app you're building.

Resource Owner

The Resource Owner is the entity that owns the protected data. In most cases, it's an individual user. But it could also be an organization—for example, a company account that owns documents, calendars, or customer records.

When the Resource Owner grants your app access to their data (like allowing you to read their calendar), they authorize it through the Authorization Server. The app never sees their credentials—only the tokens granted by that authorization.

Resource Server

The Resource Server is the system that stores and manages the Resource Owner's data. It could be a REST or GraphQL API, a storage service, or any backend that requires authentication before returning protected data.

When a Client Application sends a request with an Access Token, the Resource Server validates that token—performing multiple validations. If everything is valid, it allows access to the requested resource.

Sometimes the Resource Server and the Authorization Server are the same application. In that case, verifying the token can be as simple as querying the database directly. When they're separate systems, you'll often need to verify the token using a JWKS endpoint or introspection (more on this later).

Client Application

The Client Application is the software requesting access to the Resource Server on behalf of the Resource Owner. It could be a web app, a mobile app, a desktop client, or even a server-side process.

This application doesn't store the user's password or credentials. Instead, it follows an OAuth2 flow to obtain an Access Token and, optionally, a Refresh Token. These tokens are then used to securely access the user's data.

In this book, you'll see examples built using React Router. But everything covered here can be applied to any web application—regardless of the frontend framework or language. React Router is just the example used to make the implementation concrete.

Authorization Server

The Authorization Server is responsible for authenticating users and issuing tokens. It verifies that the Resource Owner has granted permission and ensures the Client Application is authorized to make that request.

This server handles the login flow, asks the user for consent, and returns tokens if the request is valid. It also exposes endpoints for token issuance, token revocation, introspection, and public key discovery (JWKS).

Popular Authorization Servers include Auth0, Okta, Keycloak, and cloud providers like Google or Microsoft. You can also build your own, but it requires a deep understanding of the protocol and a strong security focus.

Sometimes the Authorization Server is the same as the Resource Server. When that happens, token verification and data access can be handled together in a simpler way.

Scopes

Scopes define the level of access a Client Application has to the user's resources. They ensure the app only requests permissions necessary for its functionality, following the principle of least privilege.

For example, when using an application that integrates with Google Calendar, you might see permissions like:

- **`calendar.read`**: View your calendar events.
- **`calendar.write`**: Create and edit your calendar events.

When the user authorizes an application, the requested scopes are presented explicitly, allowing the user to make an informed decision before granting access.

By limiting applications to the minimum permissions required, OAuth2 significantly reduces security risks and enhances user trust.

In practice, scopes typically appear as a list of permissions granted in a token, for instance:

```
{
  "scope": "contacts.read contacts.write profile.read"
}
```

The Resource Server then enforces these scopes, ensuring the Client Application only performs authorized operations on behalf of the user.

Well-Known Endpoints

OAuth2 and OpenID Connect providers expose standardized configuration endpoints under the `.well-known` path. These endpoints allow applications to discover the necessary URLs and capabilities of the Authorization Server without hardcoding them.

This discovery mechanism simplifies client configuration and ensures compatibility with different providers.

In the following sections, you'll learn about the three most important well-known endpoints:

- The OAuth2 Authorization Server Endpoint
- The OpenID Configuration Endpoint
- The JWKS Endpoint

Each of these plays a key role in enabling dynamic and secure communication between your application and the Authorization Server.

OAuth2 Authorization Server Metadata Endpoint

OAuth2 defines a standard way for Authorization Servers to expose their configuration via a metadata document. This document is served at the `.well-known/oauth-authorization-server` endpoint.

The full URL typically looks like this:

```
https://auth.example.com/.well-known/oauth-authorization-server
```

This metadata includes key information such as:

- `issuer`: A unique identifier for the Authorization Server
- `authorization_endpoint`: URL where the client sends users to log in and authorize access
- `token_endpoint`: URL where the client exchanges authorization codes or credentials for tokens
- `revocation_endpoint`: URL to revoke tokens
- `introspection_endpoint`: URL to validate opaque tokens
- `jwks_uri`: URL to fetch the public keys used to verify JWTs
- Supported scopes, grant types, response types, and more

By fetching this metadata dynamically, Client Applications can avoid hardcoding URLs and rely on a single source of truth for server configuration.

This endpoint is especially useful in multi-tenant systems or when supporting multiple OAuth2 providers.

OpenID Configuration Endpoint

OpenID Connect (OIDC) defines a discovery mechanism similar to OAuth2's metadata. It allows Client Applications to retrieve the server's configuration dynamically by requesting a JSON document from a standardized endpoint.

This document is served at the `.well-known/openid-configuration` path.

The full URL typically looks like this:

`https://auth.example.com/.well-known/openid-configuration`

The discovery document includes most of the same information as the OAuth2 metadata endpoint, but adds a few OIDC-specific fields such as:

- `userinfo_endpoint`: URL to retrieve user profile information
- `id_token_signing_alg_values_supported`: Supported algorithms for signing ID tokens
- `claims_supported`: Supported claims that may appear in ID tokens
- `subject_types_supported`: How subject identifiers are generated

Just like with OAuth2, this endpoint simplifies client configuration and ensures your application uses the correct settings for the server it's interacting with.

JWKS Endpoint

The JWKS (JSON Web Key Set) endpoint provides the public keys used to verify the signature of JSON Web Tokens (JWTs) issued by the Authorization Server.

These keys are exposed as a JSON document at a well-known URL, typically indicated by the `jwks_uri` field in the OAuth2 or OIDC metadata.

A typical URL looks like this:

`https://auth.example.com/.well-known/jwks.json`

Each key in the JWKS includes metadata such as:

- `kid`: Key ID used to match the key with the JWT
- `alg`: The signing algorithm (e.g., RS256, ES256)
- `kty`: The key type (e.g., RSA, EC)
- `use`: Intended usage (usually `sig` for signature)
- Public key material (n, e, x, y, etc.)

The Resource Server uses these public keys to verify that a JWT was issued by a trusted Authorization Server and has not been tampered with.

Since keys may rotate over time, fetching them dynamically ensures clients always use the latest valid keys.

OAuth2 Flows

OAuth2 defines multiple authorization flows designed to accommodate different types of applications and use cases. Each flow outlines how a Client Application can obtain an access token from the Authorization Server.

Choosing the right flow depends on your application's architecture, level of trust, and ability to keep secrets securely.

The most common OAuth2 flows are:

- Authorization Code Flow
- Authorization Code Flow with PKCE

- Implicit Flow
- Client Credentials Flow
- Resource Owner Password Credentials (ROPC)
- Device Authorization Flow

This section focuses on the most secure and widely recommended flow for web applications: Authorization Code Flow with PKCE. If you're interested in the other flows, you'll find more details in the Appendix.

Authorization Code Flow with PKCE

The Authorization Code Flow with PKCE (Proof Key for Code Exchange) is the recommended OAuth2 flow for public clients—apps that cannot store a client secret securely—like mobile apps, SPAs, and even server-rendered web apps.

PKCE adds an additional layer of security to the standard Authorization Code Flow by ensuring that the client exchanging the code for an access token is the same client that initiated the flow.

Here's how the flow works:

1. The Client Application generates a random `code_verifier` and derives a `code_challenge` from it.
2. The user is redirected to the Authorization Server along with the `code_challenge`.
3. The user authenticates and grants access.
4. The Authorization Server redirects back to the Client Application with an authorization code.
5. The Client Application exchanges the code and the original `code_verifier` for an access token.

If the `code_verifier` doesn't match the `code_challenge`, the request is rejected.

This prevents malicious actors from intercepting the authorization code and exchanging it for tokens, since they wouldn't have access to the original `code_verifier`.

Even if your application runs on the server and *can* store secrets, using PKCE adds a layer of protection and is now considered a best practice for all OAuth2 applications.

PKCE (Proof Key for Code Exchange)

PKCE (pronounced "pixy") enhances the Authorization Code Flow by protecting against authorization code interception attacks.

It works by requiring the Client Application to generate a random string called the `code_verifier`. This string is hashed using SHA-256 to produce the `code_challenge`, which is sent to the Authorization Server during the initial request.

Later, when exchanging the authorization code for tokens, the Client Application must send the original `code_verifier`. The Authorization Server hashes this value and compares it to the original `code_challenge`. If they don't match, the request is denied.

Here's a simplified version of how the PKCE exchange looks in code:

```
let codeVerifier = crypto.randomUUID().replace(/-/g, "");
let codeChallenge = await sha256Base64Url(codeVerifier);

let authorizeUrl = new URL("https://auth.example.com/authorize");
authorizeUrl.searchParams.set("client_id", env.clientId);
authorizeUrl.searchParams.set("response_type", "code");
authorizeUrl.searchParams.set("redirect_uri", env.redirectURI);
authorizeUrl.searchParams.set("code_challenge", codeChallenge);
authorizeUrl.searchParams.set("code_challenge_method", "S256");
```

When the app later exchanges the authorization code for tokens:

```
let response = await fetch("https://auth.example.com/token", {
  method: "POST",
  headers: { "Content-Type": "application/x-www-form-urlencoded" },
  body: new URLSearchParams({
    client_id: env.clientId,
    client_secret: env.clientSecret,
    grant_type: "authorization_code",
    code: authorizationCode,
    redirect_uri: env.redirectURI,
    code_verifier: codeVerifier,
  }),
});
```

Since only the original client has access to the `code_verifier`, this ensures a secure exchange—even if someone intercepts the authorization code.

PKCE was originally designed for SPAs and mobile apps, but it's now widely used in server-rendered and hybrid applications for extra protection. Use it by default in any OAuth2 setup.

Other Flows

OAuth2 defines several flows, or "grant types", to support different types of applications and use cases. Each one handles authentication and token exchange slightly differently depending on where the client runs and how much it can be trusted.

This handbook focuses on the Authorization Code Flow with PKCE, which is the most appropriate for web applications. However, it's worth knowing that other flows exist, especially when working with other types of clients or APIs.

You'll find detailed explanations of these flows—including when to use each, and their tradeoffs—in the appendix.

Here's a quick overview of what's covered there:

- **Implicit Flow**: Designed for public clients like SPAs, but now deprecated due to security concerns.

- **Client Credentials Flow**: For machine-to-machine communication where no user is involved.
- **Resource Owner Password Credentials (ROPC)**: Only used in trusted environments. Avoid in most cases.
- **Device Authorization Flow**: Used for devices with limited input (like Smart TVs or game consoles).

Each of these flows serves a purpose—but unless you're building something very specific, Authorization Code with PKCE is the recommended default.

Tokens in OAuth2

Tokens are at the heart of how OAuth2 works. They're the mechanism by which clients prove they have permission to access a user's data, without having to handle or store the user's credentials.

OAuth2 uses multiple types of tokens, each with its own purpose:

- **Authorization Code**: A short-lived code used to obtain an access token.
- **Access Token**: Grants temporary access to a resource server.
- **Refresh Token**: Allows obtaining a new access token without requiring the user to log in again.
- **ID Token**: Provided by OpenID Connect to carry information about the authenticated user.

Throughout this chapter, you'll learn what each token is for, how it's used, and how to handle it securely in your application.

Authorization Code

An **Authorization Code** is a temporary credential issued to the client application after the user grants authorization.

This code is not the actual access token. Instead, it acts as a one-time-use exchange key. The client sends it to the authorization server—along with its credentials—to obtain an access token (and optionally a refresh token and ID token).

The main reasons for this two-step process are:

- **Security**: The access token isn't exposed during the redirect from the authorization server to the client.
- **Flexibility**: The server can include additional logic during the token exchange (like PKCE verification or client authentication).

The Authorization Code is short-lived and usually expires within a few minutes. It's valid for one use only and is tied to the redirect URI used during the login process.

It's also specific to the client that requested it—other clients cannot exchange it for a token.

Access Token

An **Access Token** is the credential the client application uses to access the user's data from the Resource Server. It represents the authorization granted by the user and is typically short-lived—often expiring in minutes to reduce the risk if compromised.

Depending on the authorization server, access tokens can be opaque strings or JSON Web Tokens (JWTs).

To use the access token, the client includes it in the `Authorization` header when making requests to the Resource Server:

`Authorization: Bearer ACCESS_TOKEN`

The `Bearer` prefix signals that the token itself is the proof of authorization. If an attacker obtains it, they can access protected resources until it expires. This makes secure storage and transmission essential.

The Resource Server must validate the token before allowing access. If it's a JWT, validation involves checking its signature and claims. If it's opaque, the server must use the introspection endpoint to confirm its validity.

Some access tokens include useful information like expiration time, scopes, and user ID. Others are completely opaque and require a lookup.

Refresh Token

A **Refresh Token** allows the client application to obtain a new access token without requiring the user to log in again. It's typically long-lived and is issued alongside the access token during the initial authorization.

Unlike access tokens, refresh tokens are never sent to the Resource Server. They are only used between the client and the Authorization Server.

The client sends the refresh token to the Authorization Server's token endpoint to get a new access token:

```
let response = await fetch("https://auth.example.com/token", {
  method: "POST",
  headers: {
    "Content-Type": "application/x-www-form-urlencoded",
  },
  body: new URLSearchParams({
    grant_type: "refresh_token",
    refresh_token: REFRESH_TOKEN,
    client_id: env.clientId,
    client_secret: env.clientSecret,
  }),
});

let tokens = await response.json();
```

If valid, the server returns a new access token—and optionally a new refresh token if using rotation.

Because refresh tokens are sensitive, they must be stored securely. In web applications, that often means using an HTTP-only session cookie or storing them in the backend session. If a refresh token is leaked, an attacker could generate new access tokens and impersonate the user.

To reduce risks, some servers implement **refresh token rotation**: every time the client uses a refresh token, the server issues a new one and invalidates the previous one. This way, if a stolen token is reused, it can be detected and revoked.

The use of refresh tokens is optional. In some setups, access tokens are long-lived and refreshed through a new login instead.

ID Tokens in OIDC

OpenID Connect (OIDC) is an identity layer built on top of OAuth2. When a user authenticates using OIDC, the Authorization Server issues an **ID Token** along with the access token.

The ID Token is a **JWT** that contains information about the user—like their ID, email, and name. It's meant to be read by the client application and used to identify the authenticated user, not to access APIs.

For example, when you sign in with Google, your app may receive an access token to call APIs and an ID Token to identify who the user is.

The client can decode the ID Token to extract claims like:

```
{
  "iss": "https://auth.example.com",
  "sub": "user_123",
  "email": "user@example.com",
  "name": "User Name",
  "aud": "my-client-id",
  "iat": 1712500000,
  "exp": 1712503600
}
```

In this case:

- `iss` is the issuer (Authorization Server)
- `sub` is the user ID
- `aud` is the client ID of your app
- `iat` and `exp` are the issue and expiration times

Because it's a JWT, the client can verify the signature to ensure it hasn't been tampered with.

The ID Token is useful when your app needs to show the user's name or email, or verify who logged in—without having to make an additional request to the Authorization Server.

Token Formats

OAuth2 doesn't require a specific format for access tokens. The Authorization Server decides how they're structured.

The most common options are:

- **Opaque Tokens** — random strings with no embedded information.
- **JWTs (JSON Web Tokens)** — self-contained tokens that include claims and can be verified locally.

Each format has trade-offs, and the right choice depends on your use case and infrastructure.

Opaque Tokens

An opaque token is a random string. It doesn't include any information about the user or its purpose. The Resource Server can't validate it on its own.

To validate it, the Resource Server must send it to the Authorization Server's **introspection endpoint**, which is part of the well-known metadata defined by both OAuth2 and OpenID Connect.

The Authorization Server will return a JSON object with token metadata, such as:

- `active`: whether the token is valid
- `sub`: the user ID (subject)
- `scope`: granted scopes
- `exp`: expiration timestamp
- `client_id`: which client issued the token

This endpoint requires authentication. The Resource Server usually uses the **Client Credentials Flow** to obtain an access token with permission to introspect other tokens. In the example below, `clientCredentialsAccessToken` refers to that token.

```
let response = await fetch("https://auth.example.com/introspect", {
  method: "POST",
  headers: {
    Authorization: `Bearer ${clientCredentialsAccessToken}`,
    "Content-Type": "application/x-www-form-urlencoded",
  },
  body: new URLSearchParams({
    token: userAccessToken,
  }),
});

let data = await response.json();

if (!data.active) throw new Error("Token is not valid");
```

JSON Web Tokens (JWT)

A JSON Web Token (JWT) is a compact, URL-safe token format. Unlike opaque tokens, a JWT contains structured data that can be read and validated locally by the Resource Server.

A JWT is composed of three parts:

1. **Header** – Metadata like the signing algorithm (e.g., ES256)
2. **Payload** – Claims that describe the user and the token (e.g., `sub`, `exp`, `scope`)
3. **Signature** – A cryptographic signature used to verify that the token hasn't been tampered with

To validate a JWT, the Resource Server uses the **JWKS (JSON Web Key Set)** exposed by the Authorization Server. This endpoint is part of the well-known metadata for both OAuth2 and OpenID Connect.

Here's how to verify a JWT using `@edgefirst-dev/jwt`:

```
import { JWT, JWK } from "@edgefirst-dev/jwt";

let token = extractToken(request.headers.get("Authorization"));

if (!token) throw new Error("Missing access token");

let jwks = await JWK.importRemote(
  new URL("/.well-known/jwks.json", env.issuerHost),
  { alg: JWK.Algoritm.ES256 }
);

let jwt = await JWT.verify(token, jwks, {
  issuer: env.issuer,
  audience: env.audience,
});

console.log(jwt.subject); // User ID
```

JWTs allow local validation, reducing network overhead and improving performance. However, they can't be revoked unless you implement additional mechanisms like token denylist or short expiration times.

Token Claims

Claims are pieces of information embedded within tokens that describe the authenticated user, permissions, and other metadata. While commonly associated with JWT (JSON Web Tokens), claims can also exist in opaque tokens—though in that case, the Resource Server usually retrieves claims from an introspection endpoint rather than decoding the token directly.

Common OAuth2 claims include:

- `iss` (Issuer): Identifies who issued the token (typically the Authorization Server).

- `sub` (Subject): Represents the user or entity the token refers to (e.g., a user ID).
- `aud` (Audience): Specifies the intended recipient of the token (Resource Server identifier).
- `exp` (Expiration Time): Indicates when the token expires.
- `scope`: Lists the permissions granted to the Client Application.

Opaque tokens don't carry readable claims. Instead, when validating an opaque token, the Resource Server typically calls an introspection endpoint (covered previously) to retrieve claims:

```
let claims = await introspect(token, clientCredentialsAccessToken);

if (!claims.active || claims.aud !== env.audience) {
  throw new Error("Token invalid or incorrect audience");
}
```

When using JWTs, claims are embedded directly into the token, simplifying validation without additional network requests:

```
let jwt = await JWT.verify(token, jwks, {
  issuer: env.issuer,
  audience: env.audience,
});

let userId = jwt.sub;
let permissions = jwt.scope.split(" ");
```

Whether using JWT or opaque tokens, correctly handling and validating claims is critical to ensure secure and appropriate access to your resources.

Audience (**aud**)

The **audience** (aud) claim identifies the intended recipient of the token. It ensures that tokens issued by your Authorization Server are only used by the specific Resource Server they're meant for.

Common values for audience include:

- The URL of your API (e.g., `api.yourservice.com`).
- A unique identifier defined by your API or Authorization Server
- For ID Tokens (OIDC), the audience is typically the Client ID.

When generating a token, ensure your Authorization Server includes the appropriate `aud` claim, matching what your Resource Server expects.

Here's how you validate the audience claim in a JWT:

```
let jwt = await JWT.verify(token, jwks, {
  issuer: env.issuer,
  audience: env.audience,
});
```

If the audience claim doesn't match the expected value (`env.audience` in this case), validation fails, preventing tokens from being misused by other services or APIs.

Choosing the right audience value helps ensure tokens are used securely and correctly across your application's ecosystem.

Issuer (`iss`)

The **issuer (`iss`)** claim identifies the Authorization Server that issued the token. It's essential to validate this claim to ensure tokens originate from a trusted source.

When configuring your OAuth2 setup, the Authorization Server provides a known issuer identifier—usually a URL like `auth.example.com`. Your Resource Server must confirm that tokens contain the expected issuer before accepting them.

For instance, here's how you validate it using the `@edgefirst-dev/jwt` library:

```
let jwt = await JWT.verify(token, jwks, {
  issuer: env.issuer,
});
```

This validation step ensures tokens from unauthorized or malicious servers are immediately rejected, enhancing the security of your application.

Subject (`sub`)

The **subject (`sub`)** claim identifies the user or entity that the token represents, usually a unique user ID provided by your Authorization Server. This claim lets your application securely recognize and interact with the user associated with each token.

For instance, after validating a JWT, you can use the subject claim to retrieve the user's details from your database:

```
let jwt = await JWT.verify(token, jwks, {
  issuer: env.issuer,
});

let userId = jwt.sub;
let user = await db.query("SELECT * FROM users WHERE id = ?", userId);
```

Always ensure the subject claim is validated and corresponds correctly to an existing user or entity. This approach allows your Resource Server to reliably and securely manage user-specific operations.

Expiration Time (`exp`)

The **expiration time (`exp`)** claim indicates when the token becomes invalid. This timestamp is a Unix epoch time (seconds since January 1, 1970). After this moment, your Resource Server must reject the token, ensuring expired tokens can't be misused.

When verifying JWTs using a library like `@edgefirst-dev/jwt`, expiration validation typically happens automatically. However, explicitly checking is a good practice:

```
try {
  let jwt = await JWT.verify(token, jwks, {
    issuer: env.issuer,
  });
} catch (error) {
  if (error.name === "JWTExpired") {
    throw new Error("The provided token has expired.");
  }
  throw error;
}
```

This ensures your server always rejects expired tokens, reducing security risks associated with prolonged or unauthorized access.

Scope (`scope`)

The **scope (`scope`)** claim contains a list of permissions granted by the user, defining what the client application is allowed to do on behalf of the user. This claim is typically a space-separated string of individual permissions.

For example, a token might contain:

```
{
  "scope": "contacts.read contacts.write profile.read"
}
```

JWT ID (`jti`)

The **JWT ID (`jti`)** claim is a unique identifier for the JWT itself. It's useful for tracking and managing tokens, especially in scenarios where you need to revoke or invalidate specific tokens.

When issuing a JWT, the Authorization Server generates a unique `jti` value. This value can be stored in a database or cache, allowing you to track issued tokens and manage their lifecycle.

For example, if you want to revoke a specific token, you can check the `jti` claim against your database:

```
let { jti } = await JWT.verify(token, jwks, {
  issuer: env.issuer,
});
let [revoked] = await db.query(
  "SELECT EXISTS (SELECT 1 FROM revoked_tokens WHERE jti = ?) AS revoked",
  [jti]
);

if (revoked) throw new Error("Token has been revoked.");
```

This approach allows you to implement token revocation and manage the lifecycle of JWTs effectively.

Building the Application

Now that you understand how OAuth2 works and the different types of tokens involved, it's time to look at how to implement it in a real application.

This book uses a React Router application as an example, but the same approach applies to any web application with a server component. You can adapt the same logic if you're using Next.js, Nuxt, SvelteKit, Rails, Laravel, or other frameworks.

You'll build a address book application where:

- The React Router app is the **Client Application**.
- The API storing the contact data is the **Resource Server**.
- A third service will act as the **Authorization Server**, issuing access and refresh tokens.

Let's go step by step through how each part works.

How the Application Will Work

We'll build a simple contact list app based on the Address Book tutorial from React Router. The book focuses on implementing OAuth2 authentication and authorization, not on styling or UI design.

However, the base app already includes styles and interactions, and the full example repository contains a complete, working application with a styled user interface and all the features you'd expect in a real project.

Here's a high-level overview of how the app behaves:

- The user visits the React Router application.
- When accessing a protected route, they're redirected to the Authorization Server.
- After logging in and granting access, the Authorization Server redirects back with an authorization code.
- The Client Application exchanges the code for an access token and a refresh token.
- The refresh token is stored securely in a session.
- The Client Application uses the access token to call the Resource Server and fetch the user's contacts.

The browser never sees or handles tokens directly. All token storage and API communication happen server-side in the Client Application. This prevents exposing sensitive information to the user or potential attackers.

This setup gives you a secure and clean separation between authentication, authorization, and user interaction.

The Authorization Server

The Authorization Server handles user authentication and token issuance. It's the central authority in your OAuth2 setup.

There are many options available depending on your needs:

- **Managed services** like Auth0, Okta, or Clerk.
- **Self-hosted solutions** like Keycloak or OpenAuth.js.
- **Built-in implementations** inside your own API.

You don't have to build one from scratch—unless you want full control and understand the risks and complexity.

In the example app, we'll use a minimal custom Authorization Server that issues JWTs and exposes a JWKS endpoint. This setup allows you to understand how the flow works without relying on third-party infrastructure.

Any standards-compliant OAuth2 server will work. You can swap in your preferred provider or extend the example for production scenarios.

Just make sure your Authorization Server:

- Supports the Authorization Code Flow with PKCE.
- Exposes metadata through a well-known endpoint.
- Issues access tokens and refresh tokens.
- Offers a JWKS endpoint or introspection endpoint to validate tokens.

Later in the book, you'll learn how the Client Application discovers this metadata and interacts with the server using standard endpoints.

The Resource Server

The Resource Server protects user data and requires a valid access token for every request. In this case, the API that manages contacts plays that role.

When receiving a request, it reads the access token from the `Authorization` header. How the token is verified depends on its format:

- **Opaque tokens** require a call to the introspection endpoint.
- **JWTs** can be verified locally using the JWKS provided by the Authorization Server.

If your API also acts as the Authorization Server, you may skip those steps and validate tokens directly—by checking them against the database or verifying the JWT signature with a local key.

The next sections explain how to validate tokens. Follow the instructions based on the type of token your Authorization Server provides:

- For opaque tokens, refer to the section on validating opaque tokens.
- For JWTs, skip to the JWT validation section.

Choose the method that matches your API's token format.

Verifying an Opaque Token

To verify an opaque token, your Resource Server needs to send it to the Authorization Server's introspection endpoint. Typically, you'll authenticate this request using a client ID and secret obtained through the Client Credentials Flow.

First, get an access token for the Resource Server to authorize introspection requests:

```
async function getClientCredentialsToken() {
  let response = await fetch(new URL("/token", env.issuerHost), {
    method: "POST",
    headers: {
      "Content-Type": "application/x-www-form-urlencoded",
      Authorization: `Basic ${Buffer.from(
        `${env.clientId}:${env.clientSecret}`
      ).toString("base64")}`,
    },
    body: new URLSearchParams({ grant_type: "client_credentials" }),
  });

  let data = await response.json();
  return data.access_token;
}
```

Then, use this token to call the introspection endpoint:

```
async function introspectToken(token: string, clientToken: string) {
  let response = await fetch(new URL("/introspect", env.issuerHost), {
    method: "POST",
    headers: {
      "Content-Type": "application/x-www-form-urlencoded",
      Authorization: `Bearer ${clientToken}`,
    },
    body: new URLSearchParams({ token }),
  });

  let tokenData = await response.json();
  if (!tokenData.active) throw new Error("Invalid token.");

  return tokenData;
}
```

You can now encapsulate this logic in a reusable `authenticate` function:

```
async function authenticate(request: Request) {
  let authHeader = request.headers.get("Authorization");
  if (!authHeader) throw new Error("Missing Authorization header.");

  let [type, token] = authHeader.split(" ");
  if (type !== "Bearer" || !token) {
    throw new Error("Invalid Authorization header.");
```

```
  }

  let clientToken = await getClientCredentialsToken();
  let tokenData = await introspectToken(token, clientToken);

  // e.g., { active: true, sub: "user_id", scope: "contacts.read" }
  return tokenData;
}
```

Finally, once the token is validated, you can query the database using the user's identifier:

```
let tokenData = await authenticate(request);
let userId = tokenData.sub;

let contacts = await db.query("SELECT * FROM contacts WHERE user_id = ?", [
  userId,
]);
```

This method ensures centralized control over token validity, provides an easy way to revoke access, and secures your application's protected resources.

Verifying a JWT

Verifying a JWT doesn't require network requests for every validation. Instead, your Resource Server fetches a JSON Web Key Set (JWKS) from the Authorization Server once, then uses it to verify JWT signatures locally.

First, fetch and import the JWKS into your server:

```
import { JWK, JWT } from "@edgefirst-dev/jwt";

const jwks = await JWK.importRemote(
  new URL("/.well-known/jwks.json", env.issuerHost),
  { alg: JWK.Algoritm.ES256 }
);
```

Next, create a reusable `authenticate` function to validate incoming JWTs:

```
async function authenticate(request: Request) {
  let authHeader = request.headers.get("Authorization");
  if (!authHeader) throw new Error("Missing Authorization header.");

  let [type, token] = authHeader.split(" ");
  if (type !== "Bearer" || !token) {
    throw new Error("Invalid Authorization header.");
  }

  let jwt = await JWT.verify(token, jwks, {
    issuer: env.issuer,
    audience: env.audience,
```

```
  });

  // JWT claims, e.g., { sub: "user_id", scope: "contacts.read", ... }
  return jwt;
}
```

Finally, after successful JWT validation, query your database using the user identifier:

```
let jwt = await authenticate(request);

let contacts = await db.query("SELECT * FROM contacts WHERE user_id = ?", [
  jwt.subject,
]);
```

This approach reduces overhead since your server doesn't need to contact the Authorization Server on each request. However, token revocation becomes more complex, usually requiring short-lived tokens or additional mechanisms.

The Client Application

The **Client Application** is responsible for authenticating users through OAuth2 and interacting with the Resource Server. This book uses React Router as an example framework, but you can adapt these concepts to other server-side frameworks like Next.js, SvelteKit, Rails, Laravel, and more.

You'll configure an OAuth2 client to handle the authentication flow. While the basic OAuth2 client implementation is straightforward, several nuanced steps are involved, making it preferable to use an existing authentication library for ease and security.

For the React Router example in this book, Remix Auth will be used. Remix Auth provides a convenient way to handle OAuth2 authentication with Remix and React Router. If you're familiar with Passport.js, Remix Auth provides similar functionality but aligned with modern web standards.

First, install the necessary dependency:

```
bun add remix-auth-oauth2
```

Now, configure the OAuth2 client:

```
// web/app/auth.ts
import { OAuth2Strategy } from "remix-auth-oauth2";
import env from "./env";

export type Tokens = OAuth2Strategy.VerifyOptions["tokens"];

export default await OAuth2Strategy.discover<Tokens>(
  new URL("/.well-known/oauth-authorization-server", env.ISSUER_HOST),
  {
    clientId: env.CLIENT_ID,
    clientSecret: env.CLIENT_SECRET,
    redirectURI: "http://localhost:3000/auth",
```

```
    scopes: ["openid", "contacts:read:own", "contacts:write:own"],
    audience: env.AUDIENCE,
  },
  async ({ tokens }) => tokens
);
```

This configuration leverages the OAuth2 discovery process (`well-known` endpoint) to avoid hardcoding Authorization Server endpoints, improving maintainability and security.

Session Management

After authentication, the application must persist user state securely. Using **React Router's Session Storage API**, the refresh token can be stored server-side.

```
// web/app/middleware/session.ts
import { createCookie, createCookieSessionStorage } from "react-router";
import env from "./env";

export interface SessionData {
  access?: string; // Optional in case you want to store the access token
  refresh: string;
  email: string;
}

const cookie = createCookie("session", {
  httpOnly: true,
  maxAge: 60 * 60 * 24 * 30, // One month (approx)
  path: "/",
  sameSite: "lax",
  secrets: [env.sessionSecret],
  secure: import.meta.env.PROD,
});

export const sesionStorage = createCookieSessionStorage<SessionData>({
  cookie,
});
```

This cookie configuration uses secure defaults:

- **httpOnly**: Blocks client-side JavaScript from accessing the cookie, protecting against **XSS**.
- **maxAge**: Defines the cookie's lifetime. Here it's set to 30 days.
- **sameSite**: Set to **lax** to restrict cross-site usage, mitigating **CSRF** risks.
- **secrets**: Signs the cookie to ensure integrity.
- **secure**: Sends the cookie only over **HTTPS** in production.

Next, install **Remix Utils**, which provides middleware for session handling:

```
bun add remix-utils
```

Then configure the middleware:

```
// web/app/middleware/session.ts

// Note: React Router currently mark middleware as `unstable_`, mostly
// because it lacks documentation. The API is considered stable.
import { unstable_createSessionMiddleware } from "remix-utils/middleware/session";

export const [sessionMiddleware, getSession] =
  unstable_createSessionMiddleware(sessionStorage);
```

This middleware reads the session from the cookie and provides a `getSession` function to access the session across the application.

The Authorization Route

To start the login flow, the application uses an authorization route that triggers the OAuth2 process.

This route calls the `authenticate` function from `remix-auth-oauth2`, handles the returned tokens, stores the refresh token in the session, and redirects the user to the application.

Additionally, it decodes the ID token to extract the user's email and store it in the session.

```
// web/app/routes/auth.ts
import { redirect } from "react-router";
import { authenticate } from "~/auth.server";
import { getSession } from "~/middlewares/session";
import type { Route } from "./+types/auth";

export async function loader({ request, context }: Route.LoaderArgs) {
  let tokens = await auth.authenticate(request);
  let session = getSession(context);

  if (tokens.hasRefreshToken()) session.set("refresh", tokens.refreshToken());

  let idToken = IDToken.decode(tokens.idToken());
  session.set("email", idToken.email);

  return redirect(href("/contacts"));
}
```

This flow ensures that only the refresh token and email is persisted in the session. The access token, which is short-lived, is requested later when needed.

Why We Only Store the Refresh Token

Instead of storing the access token in the session, the application stores only the refresh token.

This is intentional. Access tokens are short-lived and may expire while the user is still active in the app. Storing them in the session introduces complexity: you'd need to check expiration, refresh the token, and handle potential race conditions across multiple tabs or requests.

Imagine a user opens two tabs. If both try to use an expired access token, they'll both attempt to refresh it. Depending on the authorization server's behavior, one request might succeed and the other fail—forcing the user to log in again.

To avoid this, the application requests a new access token on demand using the stored refresh token. Once the response is sent, the access token is discarded. The refresh token remains in the session for future use.

This strategy guarantees each request starts with a valid access token, simplifying token management and avoiding race conditions.

Rate Limits in Tokens Endpoint

Some authorization servers impose rate limits on the token endpoint—especially for refresh token requests.

If you use the refresh token to get a new access token on every request, this can lead to hitting those limits. This is particularly problematic when:

- The user opens multiple tabs
- Several requests are sent in parallel
- Your app makes many subrequests during navigation

To avoid this, you might prefer storing the access token in the session and increasing its expiration time. This way, you only refresh it occasionally, keeping the number of token refreshes low and reducing the chance of being rate-limited.

This approach reduces flexibility but can be a practical trade-off depending on the limitations of your authorization provider.

Making Requests to the Resource Server

Now that we have the **refresh token** in the session, we can use it to obtain a new **access token** before making requests to the **Resource Server**.

We'll create an API client to handle this process:

```
bun add @edgefirst-dev/api-client
```

This API client extends **APIClient** from **@edgefirst-dev/api-client**.

- The **before** method adds the **Authorization** header with the **access token**.
- The **after** method checks the response. If the request fails, it parses the error message and throws an exception.
- The **viewer** method retrieves user details from the **Resource Server**.

To start using the API client, we instantiate it with the **access token** we obtain by using the **refresh token**, or by using the **access token** directly if it's available.

```ts
// web/app/clients/api.ts
import { APIClient as BaseClient } from "@edgefirst-dev/api-client";
import { ObjectParser } from "@edgefirst-dev/data/parser";
import { Contact } from "~/entities/contact";
import { User } from "~/entities/user";
import env from "~/env";

export class APIClient extends BaseClient {
  constructor(protected accessToken: string) {
    super(new URL(env.resourceHost));
  }

  override async before(request: Request) {
    request.headers.set("Authorization", `Bearer ${this.accessToken}`);
    return request;
  }

  override async after(_: Request, response: Response) {
    if (response.ok) return response;
    let data = await response.json();
    let parser = new ObjectParser(data);
    throw new Error(parser.string("error"));
  }

  async viewer() {
    let response = await this.get("/user");
    let data = await response.json();
    let parser = new ObjectParser(data);
    return new User(parser.object("user"));
  }

  // Other methods for working with the API
}
```

Now, let's create a middlware that will use the **refresh token** to obtain a new **access token** and save it into the request context.

```ts
// web/app/middleware/refresh.ts
import type { unstable_MiddlewareFunction } from "react-router";
import {
  href,
  redirect,
  unstable_RouterContextProvider,
  unstable_createContext,
} from "react-router";
import auth, { type Tokens } from "~/auth";
import { getSession } from "./session";

const accessTokenContext = unstable_createContext<string>();
```

```ts
export const refreshMiddleware: unstable_MiddlewareFunction<Response> = async (
  { context },
  next
) => {
  let session = getSession(context);

  let refreshToken = session.get("refresh");

  if (!refreshToken) throw redirect(href("/auth"));

  let tokens: Tokens | null = null;

  try {
    tokens = await auth.refreshToken(refreshToken);
  } catch {
    // If the refresh token is invalid, we need to redirect to the login page
    // to get a new one.
    session.unset("refresh");
    session.unset("email");
    throw redirect(href("/auth"));
  }

  context.set(accessTokenContext, tokens.accessToken());

  return await next();
};

export function getAccessToken(context: unstable_RouterContextProvider) {
  return context.get(accessTokenContext);
}
```

A second middleware will use the **access token** to create an instance of the **API-Client** and store it in the context.

```ts
// web/app/middleware/api-client.ts
import type { unstable_MiddlewareFunction } from "react-router";
import {
  unstable_RouterContextProvider,
  unstable_createContext,
} from "react-router";
import { APIClient } from "~/clients/api";
import { getAccessToken } from "./refresh";

const apiContext = unstable_createContext<APIClient>();

export const apiClientMiddleware: unstable_MiddlewareFunction<
  Response
> = async ({ context }, next) => {
```

```
  let accessToken = getAccessToken(context);
  context.set(apiContext, new APIClient(accessToken));
  return await next();
};

export function getAPIClient(context: unstable_RouterContextProvider) {
  return context.get(apiContext);
}
```

You can assign them directly to the layout route of the logged-in routes so that it is applied to all routes that require authentication.

```
// web/app/views/contacts/layout.tsx
import { refreshMiddleware } from "~/middleware/refresh";
import { apiClientMiddleware } from "~/middleware/api-client";

export const unstable_middleware = [
  refreshMiddleware,
  apiClientMiddleware,
] satisfies Route.unstable_MiddlewareFunction[];
```

Using the API Client

You can now use the API client inside any **route loader**:

```
import { getAPIClient } from "~/middleware/api-client";
import { getSession } from "~/middleware/session";

// web/app/views/contacts/layout.tsx
export async function loader({ request, context }: Route.LoaderArgs) {
  let api = getAPIClient(context);
  let session = getSession(context);

  let url = new URL(request.url);
  let query = url.searchParams.get("q") ?? undefined;

  let contacts = await api.contacts(query);

  return data({
    query,
    contacts,
    viewer: { email: session.get("email") },
  });
}
```

This example uses the **APIClient** to retrieve the user list of contacts. The client handles authentication behind the scenes, so you don't have to worry about managing tokens or headers.

Logging Out

When logging out, we need to clear the session and redirect the user to the login page.

```
// web/app/routes/logout.ts
import { href, redirectDocument } from "react-router";
import type { Route } from "./+types/logout";

export async function action({ context }: Route.ActionArgs) {
  return redirectDocument(href("/"), { headers: { "Clear-Site-Data": "*" } });
}
```

Here, we clear the session and use the **Clear-Site-Data** header to ensure a clean logout experience. This header instructs the browser to clear all site data, including cookies, storage, and caches.

The **Clear-Site-Data** header is supported in modern browsers and provides a more secure way to log out users by removing all traces of their session.

Token Revocation

Revoking tokens ensures that leaked credentials can't be used.

Use the **revoke** endpoint to invalidate the **refresh token**. Update the logout route to handle this step:

```
import { href, redirectDocument } from "react-router";
import auth from "~/auth";
import { getSession } from "~/middlewares/session";
import type { Route } from "./+types/logout";

export async function action({ context }: Route.ActionArgs) {
  let session = getSession(context);

  let token = session.get("refresh");
  if (token) await auth.revokeToken(token);

  session.unset("refresh");

  return redirectDocument(href("/"), { headers: { "Clear-Site-Data": "*" } });
}
```

The `revokeToken` method invalidates the refresh token so it can't be used to get new access tokens. This adds a layer of security, especially in case the token gets compromised.

OIDC RP Initiated Logout

Some Authorization Servers support **RP-Initiated Logout**, allowing the client application to log out users from the Authorization Server.

This is useful when the Authorization Server keeps track of sessions and can log out users from all clients. To implement this, add a logout route that redirects to the Authorization Server's logout endpoint:

```
import { redirectDocument } from "react-router";
import auth from "~/auth";
import { getSession } from "~/middlewares/session";
import type { Route } from "./+types/logout";

export async function action({ context }: Route.ActionArgs) {
  let session = getSession(context);

  let token = session.get("refresh");
  if (token) await auth.revokeToken(token);

  session.unset("refresh");

  // The actual endpoint depends on the configuration of your Authorization Server
  return redirectDocument(
    new URL("/oauth2/logout", env.issuerHost).toString(),
    { headers: { "Clear-Site-Data": "*" } }
  );
}
```

When the user logs out, the application clears the session and redirects to the Authorization Server's logout endpoint. This ensures that the user is logged out from both the client application and the Authorization Server.

Not every provider may support this feature, so check the documentation for your Authorization Server to see if it's available. But it's a good practice to implement it if possible, as it provides a more consistent user experience across different applications.

Refresh Token Rotation

Refresh Token Rotation increases security by issuing a new refresh token every time a new access token is generated. If someone steals a refresh token but the user refreshes their session first, the stolen token becomes useless.

Most OAuth2 providers enforce a **reuse interval**—a short window (like 30 seconds) where the same refresh token can be used twice. After that, trying to reuse an old token triggers a security event and can revoke all active tokens.

To support rotation, update the middleware to replace the refresh token in the session whenever new tokens are issued:

```
export const refreshMiddleware: unstable_MiddlewareFunction<Response> = async (
  { context },
  next
) => {
  let session = getSession(context);
```

```
  let refreshToken = session.get("refresh");

  if (!refreshToken) throw redirect(href("/auth"));

  let tokens: Tokens | null = null;

  try {
    tokens = await auth.refreshToken(refreshToken);
  } catch {
    // If the refresh token is invalid, we need to redirect to the login page
    // to get a new one.
    session.unset("refresh");
    session.unset("email");
    throw redirect(href("/auth"));
  }

  // Stores the new refresh token in the session if available
  if (tokens.hasRefreshToken()) {
    session.set("refresh", tokens.refreshToken());
  }

  context.set(accessTokenContext, tokens.accessToken());

  return await next();
};
```

This ensures the session always has the most recent refresh token, making token reuse much harder for attackers.

Conclusion

OAuth2 isn't simple. It has many moving parts, new terminology, and a wide surface for mistakes. But it's also powerful—and once you understand the flow, the roles, and the tokens, it becomes much easier to apply.

This book showed how to implement OAuth2 in a real web app using React Router. You learned how to:

- Authenticate users securely using Authorization Code Flow with PKCE
- Store and rotate tokens safely using HTTP-only cookies
- Keep the browser free of tokens by using a Backend-for-Frontend
- Interact with Resource Servers using access tokens
- Revoke tokens and support refresh token rotation

More importantly, you saw **why** these decisions matter—how they affect security, developer experience, and how the pieces work together in production.

The example application used React Router, but the principles apply to any modern

web framework that runs server-side. Whether you're using Rails, Laravel, Express, or something else, the core ideas remain the same.

You now have a solid foundation for using OAuth2 in your own projects. When you need to connect to an external API, act on behalf of a user, or build your own authentication system—you'll know how to do it securely, correctly, and with confidence.

Appendix: JSON Web Tokens (JWT)

Now that you've seen how JWTs work, it's helpful to understand how they compare with other approaches commonly used in authentication systems—like opaque tokens, sessions, and cookies.

Each has tradeoffs in terms of validation, storage, revocation, and ease of use. Here's what you need to know.

JWT vs Opaque Tokens

An opaque token is just a random string that has no intrinsic meaning to the Resource Server. When the Resource Server receives an opaque token, it must validate it by making a request to the Authorization Server—typically using the introspection endpoint.

In contrast, a JWT is self-contained: it includes all the necessary claims and is cryptographically signed. The Resource Server can verify the signature using a public key and validate the token locally, without needing to contact the Authorization Server.

This improves performance and resilience, especially in distributed systems. However, JWTs come with tradeoffs:

- If a JWT is compromised, it can't be revoked unless additional mechanisms like a revocation list are implemented.
- Opaque tokens can be revoked by removing them from the Authorization Server's database.

When using JWTs, make sure to:

- Use short expiration times (`exp`)
- Verify the `aud` (audience) and `iss` (issuer)
- Rotate signing keys periodically

JWT vs Sessions

JWTs and traditional session-based authentication take very different approaches to managing user identity.

With sessions, the server stores user data and issues a session ID. This ID is saved in a cookie on the client and sent with each request. The server uses the ID to look up the session and retrieve user information.

With JWTs, the user information is encoded in the token itself. The server doesn't need to store anything—it simply verifies the token's signature and reads its claims.

This makes JWTs stateless and ideal for distributed systems, where storing session data centrally can be a bottleneck or failure point.

However, sessions have a key advantage: they can be revoked immediately. If a user logs out or an admin invalidates a session, the session ID is removed from the store and is no longer valid. JWTs can't be revoked unless you build your own revocation system.

In the context of OAuth2, even if you're using JWTs as access tokens, the Client Application still maintains a session to keep track of the logged-in user.

This session typically stores the access token or a refresh token which are used to authenticate requests to the Resource Server. So while JWTs provide a stateless way to authorize API requests, a session still plays an important role in managing the user's login state within the application.

JWT vs Cookies

JWTs and cookies serve different purposes but are often compared because they both relate to how authentication is handled in web applications.

Cookies are a browser feature for storing small pieces of data. They're commonly used to store session IDs and are automatically included in requests to the same origin, making them convenient for session-based authentication.

JWTs, on the other hand, are tokens used to transmit information, typically about the authenticated user. They are usually sent manually in the `Authorization` header using the `Bearer` scheme.

In OAuth2, a common and secure approach is to store the access or refresh token in an **HTTP-only cookie**. This provides protection against JavaScript access (which mitigates XSS attacks) while still letting the browser send the token automatically on requests to the same origin.

This technique allows you to combine the advantages of both: the stateless, self-contained nature of JWTs and the secure, automatic transmission of cookies.

However, cookies are only automatically included on same-origin requests (or explicitly configured for cross-origin), so this setup works best when the frontend and backend share a domain or subdomain.

When using a Backend-for-Frontend (BFF) architecture, the tokens are often stored server-side in the session, and the client only receives a session cookie. In this setup, cookies carry the session identifier, while JWTs are used behind the scenes to authorize requests to the Resource Server.

Appendix: With Access Token in Session

Some apps store the **access token** in the session, not just the **refresh token**. This has trade-offs.

Storing the access token reduces the number of requests to the **token endpoint**, which is useful if you make frequent API calls or the endpoint is rate-limited.

But it also means you need to handle access token expiration and refresh manually. And you still need to store the token securely—preferably in an HTTP-only cookie or an encrypted server-side session.

This appendix shows how to adapt the implementation from earlier chapters to:

- Store the **access token** in the session
- Refresh the token when it's expired
- Keep the session updated with the latest token

It's a good option if your app calls the API often, or if you want to avoid hitting the token endpoint on every request.

Making Requests to the Resource Server

This is a variation of the earlier middleware you saw, updated to store the **access token** in the session and only refresh it when needed.

```
// web/app/middleware/refresh.ts
export const refreshMiddleware: unstable_MiddlewareFunction<Response> = async (
  { context },
  next
) => {
  let session = getSession(context);

  let accessToken = session.get("access");
  let refreshToken = session.get("refresh");

  if (!refreshToken) throw redirect(href("/auth"));

  let jwt = JWT.decode(accessToken);

  // If access token is expired, refresh it
  if (jwt.expired) {
    let tokens: Tokens | null = null;

    try {
      tokens = await auth.refreshToken(refreshToken);
    } catch {
      // If the refresh token is invalid, we need to redirect to the login page
      // to get a new one.
      session.unset("refresh");
```

```
      session.unset("email");
      throw redirect(href("/auth"));
    }

    accessToken = tokens.accessToken();
    session.set("access", accessToken); // Store the new access token
  }

  return await next();
};
```

The middleware decodes the **access token**, checks if it's expired, and refreshes it if needed. It then stores the new token in the session and uses it to authenticate requests.

This reduces token endpoint traffic by only refreshing the **access token** when it's expired.

With this approach, the `accessTokenContext` is no longer needed, as the access token is stored in the session. You can remove the `getAccessToken` function and access the token directly from the session in your API client.

Refresh Token Rotation

This is a modified version of the previous middleware that adds support for **refresh token rotation** while storing the **access token** in the session.

```
// web/app/middleware/refresh.ts
export const refreshMiddleware: unstable_MiddlewareFunction<Response> = async (
  { context },
  next
) => {
  let session = getSession(context);

  let accessToken = session.get("access");
  let refreshToken = session.get("refresh");

  if (!refreshToken) throw redirect(href("/auth"));

  let jwt = JWT.decode(accessToken);

  // If access token is expired, refresh it
  if (jwt.expired) {
    let tokens: Tokens | null = null;

    try {
      tokens = await auth.refreshToken(refreshToken);
    } catch {
      // If the refresh token is invalid, we need to redirect to the login page
      // to get a new one.
```

```
      session.unset("refresh");
      session.unset("email");
      throw redirect(href("/auth"));
    }

    accessToken = tokens.accessToken();
    session.set("access", accessToken); // Store the new access token

    if (tokens.hasRefreshToken()) {
      session.set("refresh", tokens.refreshToken());
    }
  }

  return await next();
};
```

Each time the **access token** is refreshed, the new **refresh token** replaces the old one in the session—if provided. This supports **refresh token rotation**, helping to mitigate the risks of leaked or reused tokens.

Appendix: Other OAuth2 Flows

In this book, we focused on the **Authorization Code Flow with PKCE**, which is the most secure and flexible option for web applications.

However, OAuth2 defines several other flows, each suited for different types of clients and contexts. This appendix provides a more complete overview of these flows, including when to use them and their main characteristics.

Authorization Code Flow without PKCE

This is the original version of the flow. It's intended for **confidential clients**, like traditional server-rendered web apps, that can safely store a `client_secret`.

The flow works like this:

1. The user is redirected to the Authorization Server to log in and authorize the application.
2. The Authorization Server sends an **authorization code** to the redirect URI.
3. The Client Application exchanges the code for an **access token** using its `client_id` and `client_secret`.

Here's an example of how to obtain an access token using this flow:

```
let response = await fetch("https://auth.example.com/token", {
  method: "POST",
  headers: {
    "Content-Type": "application/x-www-form-urlencoded",
  },
  body: new URLSearchParams({
```

```
    grant_type: "authorization_code",
    code: "abc123",
    redirect_uri: "https://example.com/callback",
    client_id: CLIENT_ID,
    client_secret: CLIENT_SECRET,
  }),
});
let tokens = await response.json();
```

It's almost identical to the Authorization Code Flow with PKCE, except it doesn't generate and includes the `code_verifier` and `code_challenge` parameters.

Implicit Flow

Designed for early SPAs that couldn't securely handle secrets or perform server-side logic.

Instead of exchanging an authorization code, the Authorization Server **returns the access token directly in the URL fragment** after user login.

This introduces several security risks:

- The token is exposed in browser history.
- It can be accessed by scripts.
- It doesn't support refresh tokens.

Due to these issues, the Implicit Flow is now **deprecated** and should be avoided. Use Authorization Code with PKCE instead, even for SPAs.

Client Credentials Flow

This flow is used when the client is **acting on its own behalf**, rather than on behalf of a user.

It's perfect for server-to-server integrations, background jobs, or internal tools.

```
let response = await fetch("https://auth.example.com/token", {
  method: "POST",
  headers: {
    "Content-Type": "application/x-www-form-urlencoded",
    Authorization: "Basic " + btoa(`${CLIENT_ID}:${CLIENT_SECRET}`),
  },
  body: new URLSearchParams({
    grant_type: "client_credentials",
  }),
});

let tokens = await response.json();
```

No refresh tokens are issued, and there's no user identity involved.

Resource Owner Password Credentials (ROPC)

This flow allows the client to ask the user for their **username and password** directly, then sends those credentials to the Authorization Server.

It's now discouraged because:

- The client sees the user's password.
- It bypasses the Authorization Server's UI and multi-factor flows.
- It doesn't support modern consent and identity best practices.

ROPC should only be used in **highly trusted environments**, like first-party mobile apps or legacy systems that can't support other flows.

Here's an example of how to use ROPC to obtain an access token:

```
let response = await fetch("https://auth.example.com/token", {
  method: "POST",
  headers: {
    "Content-Type": "application/x-www-form-urlencoded",
  },
  body: new URLSearchParams({
    grant_type: "password",
    username: "user@example.com",
    password: "supersecret",
    client_id: CLIENT_ID,
    client_secret: CLIENT_SECRET,
  }),
});
let tokens = await response.json();
```

Device Authorization Flow

Designed for devices with limited input (e.g., Smart TVs, consoles, IoT).

Here's how it works:

1. The device requests a **device code** and a **user code**.
2. It shows the user a screen with a short URL and the user code. This is usually a QR code.
3. The user visits the URL on a separate device, logs in, and enters the code.
4. Meanwhile, the device **polls** the Authorization Server until authorization is complete.
5. Once authorized, it receives an access token.

Requesting the device/user code:

```
let response = await fetch("https://auth.example.com/device/code", {
  method: "POST",
  headers: {
    "Content-Type": "application/x-www-form-urlencoded",
  },
```

```
  body: new URLSearchParams({
    client_id: CLIENT_ID,
    scope: "read:contacts",
  }),
});
let deviceData = await response.json();
// Show deviceData.user_code and deviceData.verification_uri to the user
```

Polling for the access token:

```
let response = await fetch("https://auth.example.com/token", {
  method: "POST",
  headers: {
    "Content-Type": "application/x-www-form-urlencoded",
  },
  body: new URLSearchParams({
    grant_type: "urn:ietf:params:oauth:grant-type:device_code",
    device_code: deviceData.device_code,
    client_id: CLIENT_ID,
  }),
});
let tokens = await response.json();
```

If you have used a TV streaming application, you may have seen this flow in action. It's a great way to authenticate devices that lack a keyboard or mouse, or where using the in-app keyboard is cumbersome.

Choosing the Right Flow

Each of these flows exists for a reason. Choosing the right one depends on your app's environment, security needs, and how much control you have over the client.

For modern web apps—especially those built with **React Router** or **Remix**—Authorization Code Flow with PKCE remains the best balance of **security**, **developer experience**, and **future-proofing**.

Appendix: Error Handling

Error handling is crucial in any application, especially when dealing with authentication and authorization. OAuth2 and JWTs can fail for various reasons, and your application should gracefully handle these errors.

The package used in the Client Application, `remix-auth`, provides built-in error handling for common OAuth2 errors. However, before implementing it, ensure that you understand the specific errors that can occur and how to handle them appropriately.

OAuth2 Errors

OAuth2 defines several error types that can occur during the authorization process. Here are some common ones:

- `invalid_request`: The request is missing a required parameter or is malformed.
- `unauthorized_client`: The client is not authorized to request an authorization code.
- `access_denied`: The resource owner or authorization server denied the request.
- `unsupported_response_type`: The authorization server does not support the response type.
- `invalid_scope`: The requested scope is invalid, unknown, or malformed.
- `server_error`: The authorization server encountered an error while processing the request.
- `temporarily_unavailable`: The authorization server is temporarily unable to handle the request.

When handling these errors, you should provide meaningful feedback to the user. For example, if the user denies access, inform them that they need to grant permission for the application to work correctly.

When the Authorization Server returns an error, it typically includes an `error` parameter in the response. You can use this to determine the type of error and handle it accordingly.

`https://example.com/callback?error=access_denied&error_description=User%20denied%2`

In this case, you can check the `error` parameter and display a message to the user indicating that they need to grant access.

There's also an `error_description` parameter that provides additional information about the error. This can be useful for debugging or logging purposes. **It's not recommended to show this to users**, as it may contain sensitive information.

The `error_uri` parameter can also be included, which provides a URL with more information about the error. This is useful for developers but should not be shown to users.

Of these, only `error` is required. The other two are optional and may not be present in all error responses.

Remix Auth OAuth2

The `remix-auth-oauth2` package provides built-in error handling for OAuth2 flows. When using the `OAuth2Strategy`, you can catch errors in your authentication route and handle them accordingly.

```
import { OAuth2RequestError } from "remix-auth-oauth2";
import { redirect } from "react-router";
```

```ts
export async function loader({ request, context }: Route.LoaderArgs) {
  try {
    let tokens = await auth.authenticate(request);
    // Handle successful authentication
  } catch (error) {
    if (error instanceof OAuth2RequestError) {
      // Handle OAuth2 errors
      switch (error.code) {
        case "access_denied":
          return redirect("/auth?error=access_denied");
        case "invalid_request":
          return redirect("/auth?error=invalid_request");
        default:
          return redirect("/auth?error=unknown");
      }
    } else {
      // Handle other errors
      return redirect("/auth?error=unknown");
    }
  }
}
```

The `OAuth2RequestError` class provides a `code` property that contains the error type. You can use this to determine how to handle the error.

There's also a `description` and `uri` properties that provide additional context about the error.

The strategy may also throw other non-OAuth2 errors, such as range errors, reference errors, etc. You can catch these errors in the same way and handle them accordingly.

```ts
import { OAuth2RequestError } from "remix-auth-oauth2";
import { redirect } from "react-router";

export async function loader({ request, context }: Route.LoaderArgs) {
  try {
    let tokens = await auth.authenticate(request);
    // Handle successful authentication
  } catch (error) {
    if (error instanceof OAuth2RequestError) {
      // Handle OAuth2 errors
      switch (error.code) {
        case "access_denied":
          return redirect("/auth?error=access_denied");
        case "invalid_request":
          return redirect("/auth?error=invalid_request");
        default:
          return redirect("/auth?error=unknown");
      }
    }
```

```
    if (error instanceof ReferenceError) {
      // Handle reference errors
      return redirect("/auth?error=unknown");
    }

    // Handle other errors
    return redirect("/auth?error=unknown");
  }
}
```

Appendix: The Backend for Frontend (BFF) Pattern in React Router

The **Backend for Frontend (BFF)** pattern is an architectural pattern that creates a dedicated backend service for each frontend application. This backend service acts as an intermediary between the frontend and one or more backend services, including first-party APIs, and third-party services.

Why is it important? When using React Router to consume an API from server-side loaders and actions, as described in this book, you're implementing the BFF pattern.

Loaders and actions in React Router act as the BFF, fetching data from the multiple API endpoints, formatting it, and sending to the UI exactly what it needs. This reduces the complexity of the frontend code and allows you to tailor the API responses to the specific needs of each frontend application.

But more importantly, it allows you to keep token management and authentication logic on the server side, away from the browser. This is crucial for security, as it prevents exposing sensitive tokens or credentials to the client-side code.

While there are ways to implement OAuth2 directly in the frontend, such as using the **Authorization Code Flow with PKCE**, this approach can lead to security vulnerabilities if not done correctly. By using the BFF pattern, you can ensure that all authentication and token management is handled securely on the server side, and that the browser never has direct access to sensitive tokens.

Appendix: Glossary

This glossary provides definitions for key terms and concepts related to OAuth2, JWTs, and authentication, in short and simple terms.

- **Access Token**: A token that grants access to a resource server on behalf of a user. It's usually short-lived and must be sent with each request.
- **Authorization Code Flow with PKCE**: An enhanced version of the Authorization Code Flow that adds security by requiring a code challenge and verifier. It's recommended for public clients like SPAs and mobile apps.

- **Authorization Code Flow without PKCE**: The original version of the Authorization Code Flow that requires a client secret. It's intended for confidential clients that can safely store secrets.
- **Authorization Code Flow**: An OAuth2 flow that involves redirecting the user to the authorization server, obtaining an authorization code, and exchanging it for an access token. It's the most secure flow for web applications.
- **Authorization Code**: A temporary code issued by the authorization server after user login. It's exchanged for an access token.
- **Authorization Server**: The server that issues access tokens after authenticating the user. It handles the OAuth2 flow.
- **Client Application**: The application that requests access to a resource server on behalf of the user. It can be a web app, mobile app, or server-side application.
- **Client Credentials Flow**: An OAuth2 flow used when the client is acting on its own behalf, without user interaction. It's suitable for server-to-server integrations and background jobs.
- **Client ID**: A unique identifier for the client application. It's used to identify the app when requesting tokens.
- **Client Secret**: A secret key used to authenticate the client application with the authorization server. It should be kept confidential.
- **Device Authorization Flow**: An OAuth2 flow designed for devices with limited input. It involves displaying a user code and verification URI to the user, who authorizes the client application on a separate device.
- **ID Token**: A token that contains information about the user's identity. It's issued by the authorization server and is usually a JWT.
- **Implicit Flow**: An OAuth2 flow that returns the access token directly in the URL fragment after user login. It's now deprecated due to security risks.
- **JWKS (JSON Web Key Set)**: A set of public keys exposed by the authorization server used to verify JWT signatures.
- **JWT (JSON Web Token)**: A compact, URL-safe token format that contains claims about the user. It's self-contained and can be verified without contacting the authorization server.
- **OAuth2**: An authorization framework that allows third-party applications to obtain limited access to a user's resources without sharing their credentials.
- **OIDC (OpenID Connect)**: An identity layer built on top of OAuth2 that adds authentication via ID tokens and a userinfo endpoint.
- **PKCE (Proof Key for Code Exchange)**: An extension to the Authorization Code Flow that adds security by requiring a code challenge and verifier.
- **Refresh Token**: A long-lived token used to obtain new access tokens. It's usually stored securely and can be used to refresh the session without requiring user login.
- **Resource Owner Password Credentials Flow (ROPC)**: An OAuth2 flow that allows the client to ask the user for their username and password directly. It's discouraged due to security risks.
- **Resource Owner**: The user who owns the resources being accessed. They must authorize the client application to access their resources.
- **Resource Server**: The server that hosts the user's resources. It validates access tokens and serves requests from the client application.
- **Scopes**: Permissions requested by the client application to access specific re-

sources. They define the level of access granted to the client.
- **Token Audience**: The intended recipient of the token, identified by the `aud` claim.
- **Token Claims**: The pieces of information contained in a JWT.
- **Token Endpoint**: The endpoint on the authorization server used to exchange authorization codes for access tokens or refresh tokens.
- **Token Expiration**: The process of marking a token as invalid after a certain period, typically defined in the `exp` claim.
- **Token Introspection Endpoint**: An endpoint on the authorization server used to validate access tokens.
- **Token Introspection**: A process where the resource server validates an access token by contacting the authorization server.
- **Token Issuer**: The entity that issued the token, identified by the `iss` claim.
- **Token Revocation Endpoint**: An endpoint on the authorization server used to revoke access or refresh tokens.
- **Token Revocation**: The process of invalidating a token so it can no longer be used.
- **Token Rotation**: The practice of issuing a new refresh token every time one is used, to prevent reuse.
- **Token Storage**: The method used to store tokens securely, like HTTP-only cookies or server-side sessions.
- **Token Type**: The kind of token being used, e.g., Bearer, Refresh, or ID.
- **Token**: A general term for credentials used to access resources—includes Access, Refresh, and ID tokens.
- **User Agent**: The software (usually a web browser) that interacts with the authorization server and client application on behalf of the user.
- **User Code**: A code displayed to the user during the Device Authorization Flow.
- **User Info Endpoint**: An endpoint on the authorization server that returns user information based on the access token.
- **Verification URI**: A URL displayed to the user during the Device Authorization Flow.
- **Well-Known Endpoints**: Standardized URLs for OAuth2/OIDC configuration, such as JWKS, openid-configuration, and authorization server metadata.

Appendix: Comparison Tables

This section provides a quick reference for comparing different aspects of OAuth2, JWTs, and token types.

OAuth2 vs OpenID Connect

Feature	OAuth2	OpenID Connect (OIDC)
Purpose	Authorization	Authentication built on top of OAuth2

Feature	OAuth2	OpenID Connect (OIDC)
Defines user identity	No	Yes
Returns ID Token	No	Yes (JWT with user info)
Token types	Access Token, Refresh Token	Access Token, Refresh Token, ID Token
Standardized user info	No	Yes (via `userinfo` endpoint)
Used for login	Indirectly (custom)	Yes, designed for user login
Identity layer	Not included	Included
Popular providers	Stripe, GitHub, Spotify	Google, Microsoft, Okta, Auth0

JWT vs Opaque Tokens

Feature	JWT	Opaque Token
Format	Self-contained (JSON)	Random string
Can be inspected	Yes	No
Validation	Can be validated locally	Requires introspection
Performance	Faster (no extra request)	Slower (needs API call)
Revocation	Harder	Easier
Data exposure	Can leak sensitive info	Safer
Expiration enforced by	Client	Authorization Server

Access Token vs Refresh Token vs ID Token

Token Type	Purpose	Lifespan	Where it's used	Can be reused
Access Token	Authorize access to APIs	Short-lived	Sent to Resource Server	No
Refresh Token	Obtain new Access Tokens	Long-lived	Sent to Authorization Server	Yes
ID Token	Represent user's identity (OIDC only)	Short-lived	Used by Client to identify user	No

OAuth2 Flows

Flow	Best For	Requires Client Secret	Supports PKCE	Notes
Authorization Code	Web apps with backend	Yes	Optional	Most secure for server-side apps

Flow	Best For	Requires Client Secret	Supports PKCE	Notes
Authorization Code + PKCE	Public clients (SPA, mobile, web)	No	Yes	Recommended for apps that can’t store secrets
Implicit	Legacy SPAs	No	No	Deprecated, not recommended
Client Credentials	Machine-to-machine (M2M)	Yes	No	No user involved
Resource Owner Password (ROPC)	Highly trusted apps only	Yes	No	Deprecated, not recommended
Device Authorization	Smart TVs, IoT	No	No	For limited input devices