**Appendix B. OAuth 2.0 authorization and OpenID Connect authentication**[**1**](https://learning.oreilly.com/library/view/acing-the-system/9781633439108/OEBPS/Text/B.html#footnote-000)

**B.1 Authorization vs. authentication**

*Authorization* is the process of giving a user (a person or system) permission to access a specific resource or function. *Authentication* is identity verification of a user. *OAuth 2.0* is a common authorization algorithm. (The OAuth 1.0 protocol was published in April 2010, while OAuth 2.0 was published in October 2012.) *OpenID Connect* is an extension to OAuth 2.0 for authentication. Authentication and authorization/access control are typical security requirements of a service. OAuth 2.0 and OpenID Connect may be briefly discussed in an interview regarding authorization and authentication.

A common misconception online is the idea of “login with OAuth2.” Such online resources mix up the distinct concepts of authorization and authentication. This section is an introduction to authorization with OAuth2 and authentication with OpenID Connect and makes their authorization versus authentication distinction clear.

**B.2 Prelude: Simple login, cookie-based authentication**

The most basic type of authentication is commonly referred to as *simple login*, *basic authentication*, or *forms authentication*. In simple login, a user enters an (identifier, password) pair. Common examples are (username, password) and (email, password). When a user submits their username and password, the backend will verify that the password is correct for the associated username. Passwords should be salted and hashed for security. After verification, the backend creates a session for this user. The backend creates a cookie that will be stored in both the server’s memory and in the user’s browser. The UI will set a cookie in the user’s browser, such as Set-Cookie: sessionid=f00b4r; Max-Age: 86400;. This cookie contains a session ID. Further requests from the browser will use this session ID for authentication, so the user does not have to enter their username and password again. Each time the browser makes a request to the backend, the browser will send the session ID to the backend, and the backend will compare this sent session ID to its own copy to verify the user’s identity.

This process is called *cookie-based authentication*. A session has a finite duration, after which it expires/times out and the user must reenter their username and password. Session expiration has two types of timeouts: absolute and inactivity. *Absolute timeout* terminates the session after a specified period has elapsed. *Inactivity timeout* terminates the solution after a specified period during which a user has not interacted with the application.

**B.3 Single sign-on**

*Single sign-on* (SSO) allows one to log in to multiple systems with a single master account, such as an Active Directory account. SSO is typically done with a protocol called Security Assertion Markup Language (SAML). The introduction of mobile apps in the late 2000s necessitated the following:

* Cookies are unsuitable for devices, so a new mechanism was needed for long-lived sessions, where a user remains logged into a mobile app even after they close the app.
* A new use case called *delegated authorization*. The owner of a set of resources can delegate access to some but not all of these resources to a designated client. For example, one may grant a certain app permission to see certain kinds of their Facebook user information, such as their public profile and birthday, but not post on your wall.

**B.4 Disadvantages of simple login**

The disadvantages of simple login include complexity, lack of maintainability, and no partial authorization.

**B.4.1 Complexity and lack of maintainability**

Much of a simple login (or session-based authentication in general) is implemented by the application developer, including the following:

* The login endpoint and logic, including the salting and hashing operations
* The database table of usernames and salted+hashed passwords
* Password creation and reset, including 2FA operations such as password reset emails

This means that the application developer is responsible for observing security best practices. In OAuth 2.0 and OpenID Connect, passwords are handled by a separate service. (This is true of all token-based protocols. OAuth 2.0 and OpenID Connect are token-based protocols.) The application developer can use a third-party service that has good security practices, so there is less risk of passwords being hacked.

Cookies require a server to maintain state. Each logged-in user requires the server to create a session for it. If there are millions of sessions, the memory overhead may be too expensive. Token-based protocols have no memory overhead.

The developer is also responsible for maintaining the application to stay in compliance with relevant user privacy regulations such as the General Data Protection Regulation (GDPR), California Consumer Privacy Act (CCPA), and the Health Insurance Portability and Accountability Act (HIPAA).

**B.4.2 No partial authorization**

Simple login does not have the concept of partial access control permissions. One may wish to grant another party partial access to the former’s account for specific purposes. Granting complete access is a security risk. For example, one may wish to grant a budgeting app like Mint permission to see their bank account balance, but not other permissions like transferring money. This is impossible if the bank app only has simple login. The user must pass their bank app account’s username and password to Mint, giving Mint complete access to their bank account, just for Mint to view their bank balance.

Another example was Yelp before the development of OAuth. As illustrated in figure B.1, at the end of one’s Yelp user registration, Yelp will request the user for their Gmail login, so it can send a referral link or invite link to their contact list. The user has to grant Yelp complete access to their Gmail account just to send a single referral email to each of their contacts.

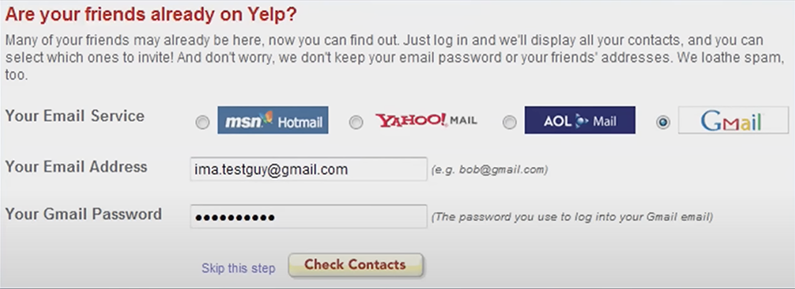


Figure B.1 Screenshot of Yelp’s browser app referral feature prior to OAuth, reflecting a shortcoming of no partial authorization in simple login. The user is requested to enter their email address and password, granting Yelp full permissions to their email account even though Yelp only wishes to send a single email to each of their contacts. Image from <http://oauthacademy.com/talk>.

OAuth 2.0 adoption is now widespread, so most apps do not use such practices anymore. A significant exception is the banking industry. As of 2022, most banks have not adopted OAuth.

**B.5 OAuth 2.0 flow**

This section describes an OAuth 2.0 flow, how an app like Google can use OAuth 2.0 for users to authorize apps like Yelp to access resources belonging to a Google user, such as send emails to a user’s Google contacts.

Figure B.2 illustrates the steps in an OAuth 2.0 flow between Yelp and Google. We closely follow figure B.2 in this chapter.

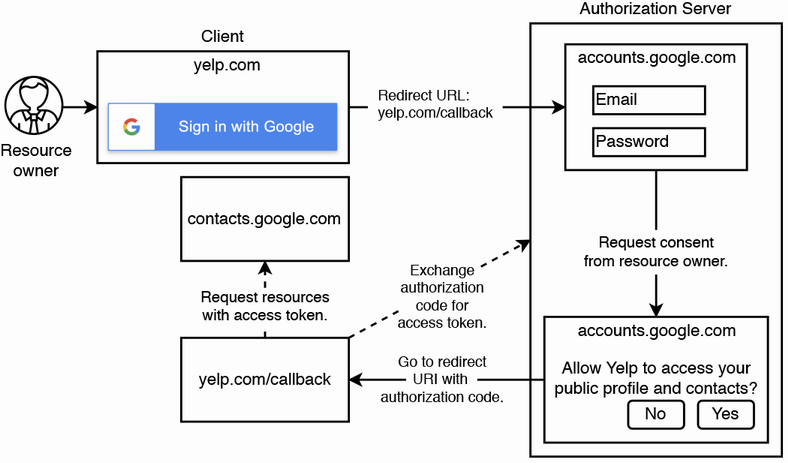


Figure B.2 Illustration of OAuth2 flow, discussed in detail through this section. Front-channel communications are represented by solid lines. Back-channel communications are represented by dashed lines.

**B.5.1 OAuth 2.0 terminology**

* *Resource owner—*The user who owns the data or controls certain operations that the application is requesting for. For example, if you have contacts in your Google account, you are the resource owner of that data. You can grant permission to an application to access that data. In this section, we refer to a resource owner as a user for brevity.
* *Client—*The application that is requesting the resources.
* *Authorization server—*The system the user uses to authorize the permission, such as accounts.google.com.
* *Resource server—*API of the system that holds the data the client wants, such as the Google Contacts API. Depending on the system, the authorization server and resource server may be the same or separate systems.
* *Authorization grant—*The proof of the user’s consent to the permission necessary to access the resources.
* *Redirect URI*, *also called callback—*The URI or destination when the authorization server redirects back to the client.
* *Access token—*The key that the client uses to get the authorized resource.
* *Scope—*The authorization server has a list of scopes that it understands (e.g., read a user’s Google contacts list, read emails, or delete emails). A client may request a certain set of scopes, depending on its required resources.

**B.5.2 Initial client setup**

An app (like Mint or Yelp) has to do a one-time setup with the authorization server (like Google) to become a client and enable users to use OAuth. When Mint requests Google to create a client Google provides:

* Client ID, which is typically a long, unique string identifier. This is passed with the initial request on the front channel.
* Client secret, which is used during token exchange.

1. Get authorization from the user

The flow begins with the (Google) resource owner on the client app (Yelp). Yelp displays a button for a user to grant access to certain data on their Google account. Clicking that button puts the user through an OAuth flow, a set of steps that results in the application having authorization and being able to access only the requested information.

When the user clicks on the button, the browser is redirected to the authorization server (e.g., a Google domain, which may be accounts.google.com, or a Facebook or Okta authorization server). Here, the user is prompted to log in (i.e., enter their email and password and click Login). They can see in their browser’s navigation bar that they is in a Google domain. This is a security improvement, as they provide their email and password to Google, rather than another app like Mint or Yelp.

In this redirect, the client passes configuration information to the authorization server via a query with a URL like “https://accounts.google.com/o/oauth2/v2/auth?client\_id=yelp&redirect\_uri=https%3A%2F%2Foidcdebugger.com%2Fdebug&scope=openid&response\_type=code&response\_mode=query&state=foobar&nonce=uwtukpm946m”. The query parameters are:

* *client\_id—*Identifies the client to the authorization server; for example, tells Google that Yelp is the client.
* *redirect\_uri (also called callback URI)—*The redirect URI.
* *scope—*The list of requested scopes.
* *response\_type—*The type of authorization grant the client wants. There are a few different types, to be described shortly. For now, we assume the most common type, called an authorization code grant. This is a request to the authorization server for a code.
* *state—*The state is passed from the client to the callback. As discussed in step 4 below, this prevents cross-site request forgery (CSRF) attacks.
* *nonce—*Stands for “number used once.” A server-provided random value used to uniquely label a request to prevent replay attacks (outside the scope of this book).

2. User consents to client’s scope

After they log in, the authorization server prompts the user to consent to the client’s requested list of scopes. In our example, Google will present them with a prompt that states the list of resources that the other app is requesting (such as their public profile and contact list) and a request for confirmation that they consent to granting these resources to that app. This ensures they are not tricked into granting access to any resource that they did not intend to grant.

Regardless of whether they click “no” or “yes,” the browser is redirected back to the app’s callback URI with different query parameters depending on the user’s decision. If they click “no,” the app is not granted access. The redirect URI may be something like “https://yelp.com/callback?error=access\_denied&error\_description=The user did not consent.” If they click “yes,” the app can request the user’s granted resources from a Google API such as the Google Contacts API. The authorization server redirects to the redirect URI with the authorization code. The redirect URI may be something like https://yelp.com/callback?code=3mPDQbnIOyseerTTKPV&state=foobar, where the query parameter “code” is the authorization code.

3. Request access token

The client sends a POST request to the authorization server to exchange the authorization code for an access token, which includes the client’s secret key (that only the client and authorization server know). Example:

POST www.googleapis.com/oauth2/v4/token

Content-Type: application/x-www-form-urlencoded

code=3mPDQbnIOyseerTTKPV&client\_id=yelp&client\_secret=secret123&grant\_type=authorization\_code

The authorization server validates the code and then responds with the access token, and the state that it received from the client.

4. Request resources

To prevent CSRF attacks, the client verifies that the state it sent to the server is identical to the state in the response. Next, the client uses the access token to request the authorized resources from the resource server. The access token allows the client to access only the requested scope (e.g., read-only access to the user’s Google contacts). Requests for other resources outside the scope or in other scopes will be denied (e.g., deleting contacts or accessing the user’s location history):

ET api.google.com/some/endpoint

Authorization: Bearer h9pyFgK62w1QZDox0d0WZg

**B.5.3 Back channel and front channel**

Why do we get an authorization code and then exchange it for the access token? Why don’t we just use the authorization code, or just get the access token immediately?

We introduce the concepts of a back channel and a front channel, which are network security terminology.

*Front-channel communication* is communication between two or more parties that are observable within the protocol. *Back-channel communication* is communication that is not observable to at least one of the parties within the protocol. This makes back channel more secure than front channel.

An example of a back channel or highly secure channel is a SSL-encrypted HTTP request from the client’s server to a Google API server. An example of a front channel is a user’s browser. A browser is secure but has some loopholes or places where data may leak from the browser. If you have a secret password or key in your web application and put it in the HTML or JavaScript of a web app, this secret is visible to someone who views the page source. The hacker can also open the network console or Chrome Developer Tools and see and modify the JavaScript. A browser is considered to be a front channel because we do not have complete trust in it, but we have complete trust in the code that is running on our backend servers.

Consider a situation where the client is going over to the authorization server. This is happening in the front channel. The full-page redirects, outgoing requests, redirect to the authorization server, and content of the request to the authorization server are all being passed through the browser. The authorization code is also transmitted through the browser (i.e., the front channel). If this authorization code was intercepted, for example, by a malicious toolbar or a mechanism that can log the browser requests, the hacker cannot obtain the access code because the token exchange happens on the back channel.

The token exchange happens between the backend and the authorization channel, not the browser. The backend also includes its secret key in the token exchange, which the hacker does not know. If the transmission of this secret key is via the browser, the hacker can steal it, so the transmission happens via the back channel.

The OAuth 2.0 flow is designed to take advantage of the best characteristics of the front channel and back channel to ensure it is highly secure. The front channel is used to interact with the user. The browser presents the user the login screen and consent screen because it is meant to interact directly with the user and present these screens. We cannot completely trust the browser with secret keys, so the last step of the flow (i.e., the exchange, happens on the back channel, which is a system we trust).

The authorization server may also issue a refresh token to allow a client to obtain a new access token if the access token is expired, without interacting with the user. This is outside the scope of this book.

**B.6 Other OAuth 2.0 flows**

We described the authorization code flow, which involves both back channel and front channel. The other flows are the implicit flow (front channel only), resource owner password credentials (back channel only), and client credentials (back channel only).

An implicit flow is the only way to use OAuth 2.0 if our app does not have a backend. Figure B.3 illustrates an example of implicit flow. All communications are front channel only. The authorization server returns the access code directly, with no authorization code and no exchange step.

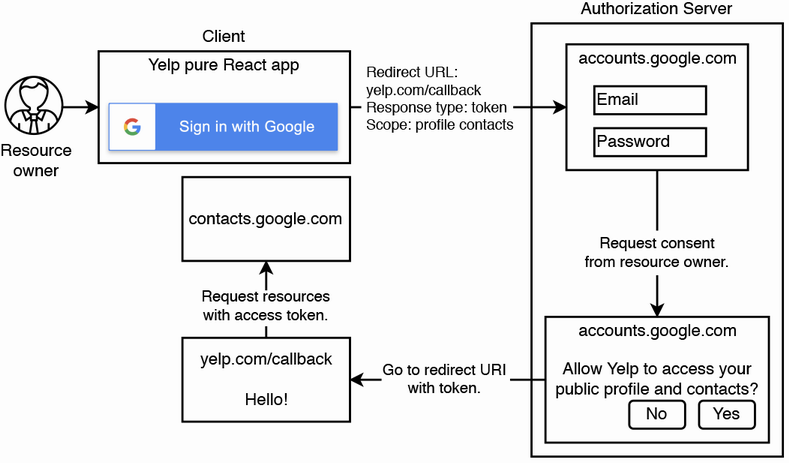


Figure B.3 Illustration of an OAuth2 implicit flow. All communications are front channel. Note that the request to the authorization server has response type “token” instead of “code.”

Implicit flow carries a security tradeoff because the access token is exposed to the browser.

The resource owner password flow or resource owner password credentials flow is used for older applications and is not recommended for new applications. The backend server uses its credentials to request the authorization server for an access token. The client credentials flow is sometimes used when you’re doing a machine-to-machine or service communications.

**B.7 OpenID Connect authentication**

The Login with Facebook button was introduced in 2009, followed by the Login with Google button and similar buttons by many other companies like Twitter, Microsoft, and LinkedIn. One could login to a site with your existing credentials with Facebook, Google, or other social media. These buttons became ubiquitous across the web. The buttons served the login use cases well and were built with OAuth 2.0 even though OAuth 2.0 was not designed to be used for authentication. Essentially, OAuth 2.0 was being used for its purpose beyond delegated authorization.

However, using OAuth for authentication is bad practice because there is no way of getting user information in OAuth. If you log in to an app with OAuth 2.0, there is no way for that app to know who just logged in or other information like your email address and name. OAuth 2.0 is designed for permissions scopes. All it does is verify that your access token is scoped to a particular resource set. It doesn’t verify who you are.

When the various companies built their social login buttons, using OAuth under the hood, they all had to add custom hacks on top of OAuth to allow clients to get the user’s information. If you read about these various implementations, keep in mind that they are different and not interoperable.

To address this lack of standardization, OpenID Connect was created as a standard for adopting OAuth 2.0 for authentication. OpenID Connect is a thin layer on top of OAuth 2.0 that allows it to be used for authentication. OpenID Connect adds the following to OAuth 2.0:

* *ID token*—The ID token represents the user’s ID and has some user information. This token is returned by the authorization server during token exchange.
* *User info endpoint*—If the client wants more information than contained in the ID token returned by the authorization server, the client can request more user information from the user info endpoint.
* *Standard set of scopes*.

So, the only technical difference between OAuth 2.0 and OpenID Connect is that OpenID Connect returns both an access code and ID token, and OpenID Connect provides a user info endpoint. A client can request the authorization server for an OpenID scope in addition to its desired OAuth 2.0 scopes and obtain both an access code and ID token.

Table B.1 summarizes the use cases of OAuth 2.0 (authorization) vs. OpenID Connect (authentication).

Table B.1 Use cases of OAuth 2.0 (authorization) vs. OpenID Connect (authentication)

| OAuth2 (authorization) | OpenID Connect (authentication) |
| --- | --- |
| Grant access to your API. | User login |
| Get access to user data in other systems. | Make your accounts available in other systems. |

An ID token consists of three parts:

* *Header*—Contains several fields, such as the algorithm used to encode the signature.
* *Claims*—The ID token body/payload. The client decodes the claims to obtain the user information.
* *Signature*—The client can use the signature to verify that the ID token has not been changed. That is, the signature can be independently verified by the client application without having to contact the authorization server.

The client can also use the access token to request the authorization server’s user info endpoint for more information about the user, such as the user’s profile picture. Table B.2 describes which grant type to use for your use case.

Table B.2 Which grant type to use for your use case

| Web application with server backend | Authorization code flow |
| --- | --- |
| Native mobile app | Authorization code flow with PKCE (Proof Key for Code Exchange) (outside the scope of this book) |
| JavaScript Single-Page App (SPA) with API backend | Implicit flow |
| Microservices and APIs | Client credentials flow |