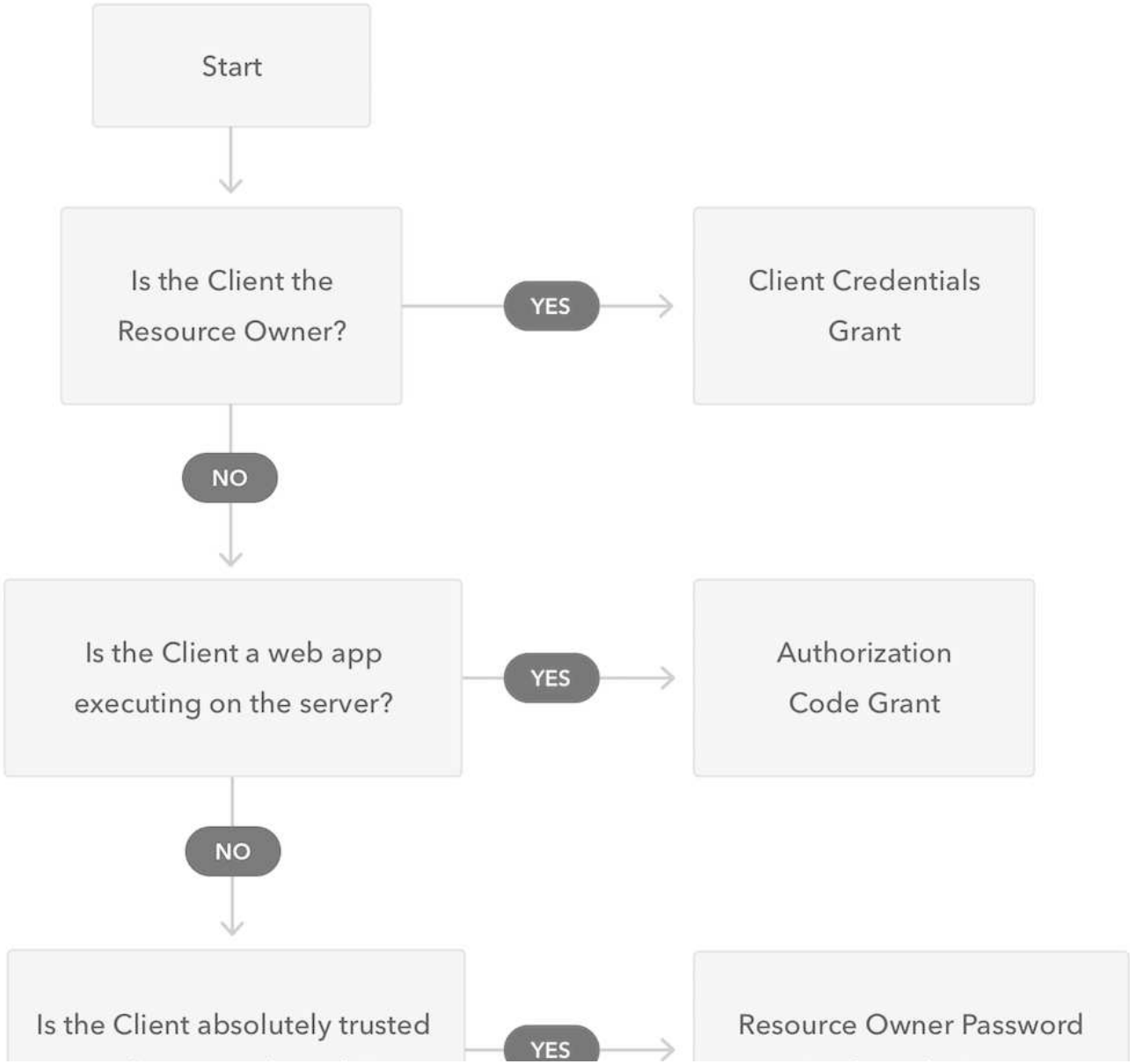
Which OAuth 2.0 flow should I use?

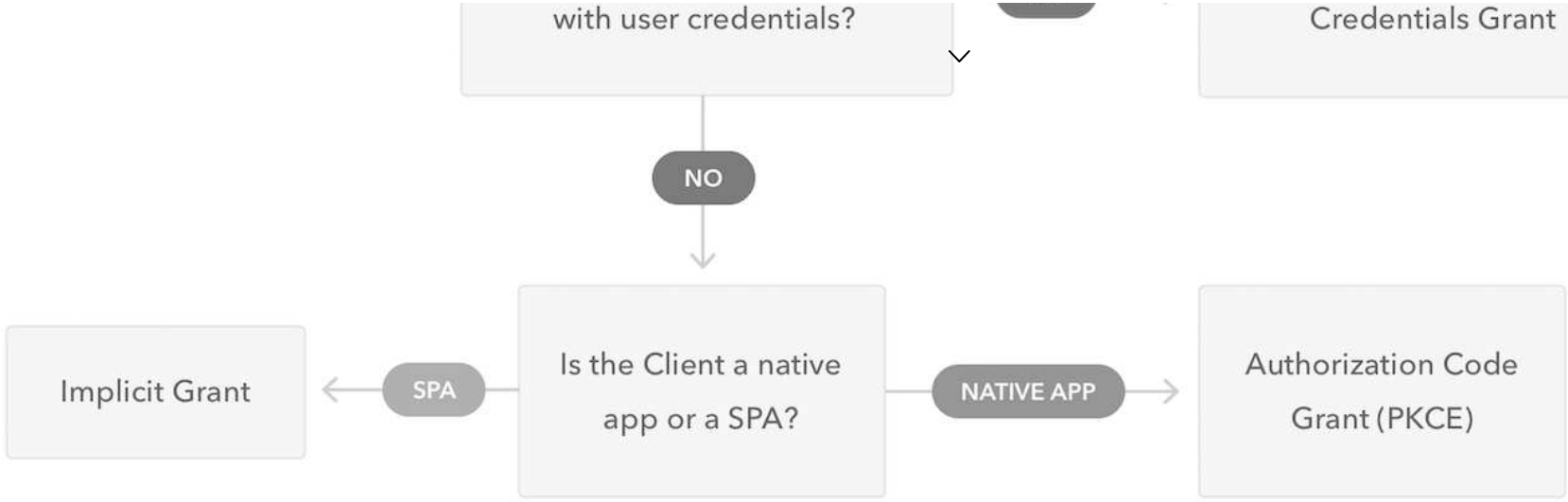
In this article

Heads up! As part of our efforts to improve security and standards-based interoperability, we have implemented several new features in our authentication flows and made changes to existing ones. For an overview of these changes, and details on how you adopt them, refer to Introducing OIDC Conformant Authentication.

OAuth 2.0 supports several different **grants.** By grants we mean ways of retrieving an Access Token. Deciding which one is suited for your case depends mostly on your Application's type, but other parameters weigh in as well, like the level of trust for the Application, or the experience you want your users to have.

Follow this flow to identify the grant that best matches your case.





Quick refresher - OAuth 2.0 terminology

- **Resource Owner**: the entity that can grant access to a protected resource. Typically this is the end-user.
- **Application**: an application requesting access to a protected resource on behalf of the Resource Owner.
- **Resource Server**: the server hosting the protected resources. This is the API you want to access.
- **Authorization Server**: the server that authenticates the Resource Owner, and issues Access Tokens after getting proper authorization. In this case, Auth0.
- **User Agent**: the agent used by the Resource Owner to interact with the Application, for example a browser or a native application.

Is the Application the Resource Owner?

The first decision point is about whether the party that requires access to resources is a machine. In this case of machine to machine authorization, the Application is also the Resource Owner. No end-user authorization is needed in this case. An example is a cron job that uses an API to import information to a database. In this example the cron job is the Application and the Resource Owner since it holds the Client ID and Client Secret and uses them to get an Access Token from the Authorization Server.

If this case matches your needs, then for more information on how this flow works and how to implement it refer to Calling APIs from a service.

Is the Application a web app executing on the server?

If the Application is a regular web app executing on a server then the **Authorization Code Grant** is the flow you should use. Using this the Application can retrieve an Access Token and, optionally, a Refresh Token. It's considered the safest choice since the Access Token is passed directly to the web server hosting the Application, without going through the user's web browser and risk exposure.

If this case matches your needs, then for more information on how this flow works and how to implement it refer to Calling APIs from server-side web apps.

Is the Application absolutely trusted with user credentials?

This decision point may result to suggesting the **Resource Owner Password Credentials Grant**. In this flow the end-user is asked to fill in credentials (username/password) typically using an interactive form. This information is sent to the backend and from there to Auth0. It is therefore imperative that the Application is absolutely trusted with this information.

This grant should **only** be used when redirect-based flows (like the Authorization Code Grant) are not possible. If this is your case, then for more information on how this flow works and how to implement it refer to Call APIs from Highly Trusted Applications.

Is the Application a native app or a SPA?

If the Application is a Single Page Application (meaning an application running in a browser using a scripting language such as Javascript) then the **Implicit Grant** should be used. In this case, instead of getting an authorization code that needs to be exchanged for an Access Token, the Application retrieves directly an Access Token. On the plus side, this is more efficient since it reduces the number of round trips required to get an Access Token. However, a security consideration is that the Access Token is exposed on the client side. Also, it should be noted that **Implicit Grant** does not return a Refresh Token because the browser cannot keep it private (read the **SPAs and Refresh Tokens** panel for a workaround).

For more information on how this flow works and how to implement it, refer to Call APIs from client-side web apps.

SPAs and Refresh Tokens

While SPAs cannot use Refresh Tokens, they can take advantage of other mechanics that provide the same function. A workaround to improve user experience is to use `prompt=none` when you invoke the /authorize endpoint. This will not display the login dialog or the consent dialog. For more information on this, refer to Silent Authentication. In addition to that if you call `/authorize` from a hidden iframe and extract the new Access Token from the parent frame, then the user will not see the redirects happening.

If the Application is a native app then the **Authorization Code Grant using Proof Key for Code Exchange** should be used. This grant adds the concept of a `code_verifier` to the Authorization Code Grant. When at first the application asks for an **Authorization Code** it generates a `code_verifier` and its transformed value called `code_challenge`. The `code_challenge` is sent along with the request. A `code_challenge_method` is also sent. Afterwards, when the application wants to exchange the Authorization Code for an Access Token, it also sends along the `code_verifier`. The Authorization Server transforms this and if it matches the originally sent `code challenge` it returns an Access Token.

For more information on how this flow works and how to implement it, refer to Calling APIs from Mobile Apps.

Pricing

Why Auth0

How It Works

Lock

COMPANY

About Us

Blog

Jobs

Press

LEARN

Availability & Trust

Security

White Hat

API Explorer

MORE

Help & Support

Documentation

Open Source

WordPress

CONTACT

10900 NE 8th Street
Suite 700
Bellevue, WA 98004

+1 (888) 235-2699
+1 (425) 312-6521
+44 (0) 33-3234-1966