Differences in flows of Oauth2 and when to use them.

Veronika N. Valeva

Fontys University of Applied Science

Versioning table

Version	Date	Author	Changes	State
0.1	Nov 1 2021	Veronika Valeva	Initial state of the research document	Sprint 3 deliverable
0.2	Nov 22 2021	Veronika Valeva	Refactoring on the introduction and the recommendation.	Sprint 4 deliverable
0.3	Jan 6 2022	Veronika Valeva	APA style Bibliography	Sprint 6 deliverable

Abstract

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. The OAuth 2.0 authorization framework holds different roles, scopes and grant types. Different grant types are structured differently and used in various ways, each grant type vary in level of complexity and security consideration.

Research points

Introduction

The vast usage of smart devices in our daily routine and our constantly growing involvement and urge to be up to date with the latest news and social/virtual life updates brings a high demand of accessing remote data on several different services. That leads to the nuisance of trying to maintain different accounts on every single service. It is time consuming and inconvenient. ***The question is how to spare the user some time and nuisance of constantly having to maintain his/her services and make the usage fast, easy, secure and reliable.*** A known solution, which is the main topic of this research and we are going to look at, to that problem is the Open AUTH 2.0 framework. This framework makes use of a single account to identify users throughout the different services, without exposing their vulnerable data. This research uses an evaluation approach. This research analyses existing information about the subject of OAuth 2.0 to arrive at objective research outcomes or reach informed decisions.

Common examples of its usage

To begin to understand what problems OAuth2.0 solves, let's investigate some practical examples of OAuth2.0 in use:

- Each time a site offers a user to log into his/her profile using his/her Google account OAuth 2.0 is in use.
- The functionality to share a post to Pinterest and Instagram at the same time is done using OAuth 2.0.

At a high level, the OAuth 2.0 protocol allows two parties to exchange information securely and reliably. In more practical terms, you'll find that the most common uses of OAuth 2.0 involve two things: Allowing a user to log into an application with another account. For example, Pinterest allowing users to log in with their Twitter accounts. This is known as *federated identity*. Allowing one service to access resources on another service on behalf of the user. For example, Adobe accessing the user’s Facebook photos on his/her behalf. This is known as *delegated authority*.

Federated identity - It refers to the concept that allows one service provider to allow authentication of a user using their identity with another service provider. For instance, imagine a user that logs into Foursquare and Amazon with their Facebook credentials. In this example, the user only needs to maintain a single user account, their Facebook account, which gives them access to several service providers. In this case, Facebook itself, plus Foursquare, and Amazon. They don't need to create individual accounts on Foursquare or Amazon, and therefore, don't need to maintain three separate passwords. In this sense, the user's identities across these sites are federated, as in, they are made to act as one.

Delegated authority - It refers to the ability for a service or application to gain access to a user's resources on their behalf. Take, for instance, LinkedIn, which can suggest contacts for you to add by looking at your Google contact list. In this example, LinkedIn will be able to view your Google contact list on your behalf. Permission to access your Google contacts has been delegated to LinkedIn.

Some common, day to day examples of OAuth2.0 usage:

- StackOverflow allowing you to log in with your Google account.
- Posting a status update from your phone using the Facebook mobile application.
- LinkedIn suggesting contacts for you to add by looking at your Google contacts.
- Pinterest allowing you to pin something from a WordPress blog.
- Sharing an article to your Facebook feed from the article itself.

Research main question

- **How to implement OAuth2.0?**

Research sub questions

- What is OAuth2.0?
- How does OAuth2.0 work?
- What are the different scopes, roles and principles of OAuth 2.0?
- What are the different flows of OAuth 2.0? How are they implemented and how do they work?
- When should the different flows of OAuth2.0 be implemented?

The three levels of the Development Oriented Triangulation Framework are used for defining the main questions of this research. The "What" of the research - Explaining the base of the research and the basics of OAuth2.0. The "Why" of the research - Explaining the usage of the topic of the research. The "How" of the research - Explaining the implementation of the topic of the research.

Table of Contents

Contents

OAuth 2.0 terminology

1. **OAuth 2.0:** the industry-standard protocol for authorization.
2. **Resource Owner:** Entity that can grant access to a protected resource. Typically, this is the end-user.
3. **Client:** Application requesting access to a protected resource on behalf of the Resource Owner.
4. **Resource Server:** Server hosting the protected resources. This is the API you want to access.
5. **Authorization Server:** Server that authenticates the Resource Owner and issues Access Tokens after getting proper authorization. In this case, Auth0.
6. **User Agent:** Agent used by the Resource Owner to interact with the Client (for example, a browser or a native application).
7. **Access token:** It is a piece of data that represents the authorization to access resources on behalf of the end-user.
8. **JSON Web Token (JWT):** It is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.
9. **Cross site request forgery (CSRF):** It is an attack vector that tricks a web browser into executing an unwanted action in an application to which a user is logged in.
10. **application/x-www-form-urlencoded:** Represents a URL encoded form.
11. **UTF-8 (UCS Transformation Format 8):** It is the World Wide Web's most common character encoding.

12. **HTTP entity:** It is the majority of an HTTP request or response, consisting of some of the headers and the body, if present

OAuth 2.0

OAuth 2.0 stands for “Open Authorization”. It is an authorization framework, which enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. It replaced OAuth 1.0 in 2012 and is now the de facto industry standard for online authorization. OAuth 2.0 provides consented access and restricts actions of what the client app can perform on resources on behalf of the user, without ever sharing the user's credentials.

Principles of OAuth2.0

OAuth 2.0 is an *authorization* protocol and not an *authentication* protocol. As such, it is designed primarily as a means of granting access to a set of resources.

OAuth 2.0 uses Access Tokens. OAuth 2.0 doesn’t define a specific format for Access Tokens. However, in some contexts, the JSON Web Token (JWT) format is often used. This enables token issuers to include data in the token itself. Also, for security reasons, Access Tokens may have an expiration date.

OAuth2.0 Roles

The idea of roles is part of the core specification of the OAuth2.0 authorization framework. These define the essential components of an OAuth 2.0 system, and are as follows:

- Resource owner is the user or system that owns the protected resources and can grant access to them.

- The client is the system that requires access to the protected resources. To access resources, the client must hold the appropriate access token.
- The authorization server receives requests from the client for access tokens and issues them upon successful authentication and consent by the resource owner. The authorization server exposes two endpoints: the authorization endpoint, which handles the interactive authentication and consent of the user, and the Token endpoint, which is involved in a machine-to-machine interaction.
- The resource server protects the user’s resources and receives access requests from the client. It accepts and validates an access token from the client and returns the appropriate resources to it.

OAuth 2.0 Scopes

Scopes are an important concept in OAuth 2.0. Scope is a mechanism in OAuth 2.0 to limit an application's access to a user's account. An application can request one or more scopes, this information is then presented to the user in the consent screen, and the access token issued to the application will be limited to the scopes granted.

Grant Types in OAuth 2.0

The OAuth framework specifies several grant types for different use cases, as well as a framework for creating new grant types. The authorization framework provides several grant types to address different scenarios. The most common OAuth grant types are listed below.

- Authorization Code grant.
- Implicit Grant.
- Authorization Code Grant with Proof Key for Code Exchange (PKCE).

- Client Credentials Grant Type.
- Device Authorization Flow.
- Refresh Token Grant.

Authorization Code Grant

How does Authorization Code Grant work.

The Authorization Code Grant Type is probably the most common of the OAuth 2.0 grant types. It is used by both web apps and native apps to get an access token after a user authorizes an app. The Authorization Code grant type is used by web and mobile apps. It differs from most of the other grant types by first requiring the app launch a browser to begin the flow. At a high level, the flow has the following steps:

- The application opens a browser to send the user to the OAuth server.
- The user sees the authorization prompt and approves the app’s request.
- The user is redirected back to the application with an authorization code in the query string.
- The application exchanges the authorization code for an access token.

When the user visits this URL, the authorization server will present them with a prompt asking if they would like to authorize this application’s request. If the user approves the request, the authorization server will redirect the browser back to the redirect uri specified by the application, adding a code and state to the query string. The state value will be the same value that the application initially set in the request. The application is expected to check that the state in the redirect matches the state it originally set. This protects against CSRF and other related attacks. The code is the authorization code generated by the authorization server. This code is relatively short-lived, typically lasting between 1 to 10 minutes depending on the OAuth service. Now that the application has the authorization code, it can use that to get an access token. The application makes a POST request to the service’s token endpoint with the following parameters:

grant type = authorization code - This tells the token endpoint that the application is using the Authorization Code grant type.

code - The application includes the authorization code it was given in the redirect.

redirect uri - The same redirect URI that was used when requesting the code.

client id - The application’s client ID.

client secret - The application’s client secret. This ensures that the request to get the access token is made only from the application, and not from a potential attacker that may have intercepted the authorization code. The token endpoint will verify all the parameters in the request, ensuring the code hasn’t expired and that the client ID and secret match. If everything checks out, it will generate an access token and return it in the response.

When to use the Authorization Code Grant.

The Authorization Code flow is best used in web and mobile apps. Since the Authorization Code grant has the extra step of exchanging the authorization code for the access token, it provides an additional layer of security not present in the Implicit grant type. Using this the client 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 client, without going through the user's web browser and risking exposure.

Implicit Grant

How does the Implicit Grant work.

The Implicit Grant Type is a way for a single-page JavaScript app to get an access token without an intermediate code exchange step. It was originally created for use by JavaScript apps (which don't have a way to safely store secrets). Like the Authorization Code Grant Type, the Implicit Grant starts out by building a link and directing the user's browser to that URL. At a high level, the flow has the following steps:

- The application opens a browser to send the user to the OAuth server.
- The user sees the authorization prompt and approves the app's request.
- The user is redirected back to the application with an access token in the URL fragment.

To begin the Implicit flow, the application constructs a URL with parameters described below and directs the browser to that URL.

response type = token - This tells the authorization server that the application is initiating the Implicit flow.

client id - The application's client ID.

redirect uri - Tells the authorization server where to send the user back to after they approve the request.

scope - One or more space-separated strings indicating which permissions the application is requesting.

state - The application generates a random string and includes it in the request. It should then check that the same value is returned after the user authorizes the app. This is used to prevent CSRF attacks.

If the user approves the request, the authorization server will redirect the browser back to the redirect uri specified by the application, adding a token and state to the fragment part of the URL.

When to use the Implicit Grant.

The Implicit grant type was created for JavaScript apps while trying to also be easier to use than the Authorization Code grant. The main downside to the Implicit grant type is that the access token is returned in the URL directly, rather than being returned via a trusted back channel like in the Authorization Code flow. One of the historical reasons that the Implicit flow used the URL fragment is that browsers could manipulate the fragment part of the URL without triggering a page reload. It is not recommended to use the implicit flow (and some servers prohibit this flow entirely) due to the inherent risks of returning access tokens in an HTTP redirect without any confirmation that it has been received by the client.

Authorization Code Grant with Proof Key for Code Exchange (PKCE)

How does the Authorization Code Grant with Proof Key for Code Exchange work.

Because the PKCE-enhanced Authorization Code Flow builds upon the standard Authorization Code Flow, the steps are very similar. PKCE extension provides protections against other attacks where the authorization code may be intercepted. The technique involves the client first creating a secret, and then using that secret again when exchanging the authorization code for an access token. This way if the code is intercepted, it will not be useful since the token request relies on the initial secret.

When to use Authorization Code Flow with Proof Key for Code Exchange (PKCE).

This flow is considered best practice when using Single Page Apps (SPA) or Mobile Apps.

Client Credentials Grant Type

How does the Client Credentials Grant Type work.

The client can request an access token using only its client credentials (or other supported means of authentication) when the client is requesting access to the protected resources under its control, or those of another resource owner that have been previously arranged with the authorization server. The flow includes the following steps:

- The client authenticates with the authorization server and requests an access token from the token endpoint.
- The authorization server authenticates the client, and if valid, issues an access token.

Since the client authentication is used as the authorization grant, no additional authorization request is needed. The client makes a request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body:

grant type – This parameter is required. Value must be set to "client_credentials".

scope – This parameter is optional.

Next the client authenticates with the authorization server. If the access token request is valid and authorized, the authorization server issues an access token.

When to use Client Credentials Grant Type.

In the case of machine-to-machine authorization, the Client is also the Resource Owner, so no end-user authorization is needed. An example is a cron job (command-line utility) that uses an API to import information to a database. In this example, the cron job is the Client 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.

Device Authorization Flow

How does Device Authorization Flow work.

With input-constrained devices that connect to the internet, rather than authenticate the user directly, the device asks the user to go to a link on their computer or smartphone and authorize the device. This avoids a poor user experience for devices that do not have an easy way to enter text. To do this, device apps use the Device Authorization Flow, in which they pass along their Client ID to initiate the authorization process and get a token. The device authorization flow includes the following steps:

- The client requests access from the authorization server and includes its client identifier in the request.
- The authorization server issues a device code and an end-user code and provides the end-user verification URI.
- The client instructs the end user to use a user agent on another device and visit the provided end-user verification URI. The client provides the user with the end-user code to enter in order to review the authorization request.
- The authorization server authenticates the end user (via the user agent) and prompts the user to input the user code provided by the device client. The authorization server validates the user code provided by the user and prompts the user to accept or decline the request.
- While the end user reviews the client's request the client repeatedly polls the authorization server to find out if the user completed the user authorization step. The client includes the device code and its client identifier.

- The authorization server validates the device code provided by the client and responds with the access token if the client is granted access, an error if they are denied access, or an indication that the client should continue to poll.

When to use Device Authorization Flow

The Device Authorization feature is an OAuth 2.0 grant type. It allows users to sign in to input-constrained devices, such as smart TVs, digital picture frames, and printers, as well as devices with no browser. Device Authorization enables you to use a secondary device, such as a laptop or mobile phone, to complete sign-in to applications that run on such devices.

Refresh Token Grant

How does Refresh Token Grant work and how to use it.

A refresh token is a special kind of token used to obtain a renewed access token. For native applications, refresh tokens improve the authentication experience significantly. The user has to authenticate only once, through the web authentication process. Subsequent re-authentication can take place without user interaction, using the refresh token. You can increase security by using refresh token rotation which issues a new refresh token and invalidates the predecessor token with each request made to Auth0 for a new access token. Rotating the refresh token reduces the risk of a compromised refresh token.

Conclusion

In the modern world of social media, each of us uses dozens of applications and websites every day. OAuth 2.0 identity provider is designed to simplify the authorization process and, as a result, make the lives of users easier and safer. OAuth 2.0 authentication protocol is a simple solution based on HTTP, which makes it possible to use it on almost any platform. The observation gathered from the research point to one conclusion. OAuth managed to establish an inter-operable framework which enables a third-party application to obtain access. It is not dependent on the actual implementation or platform as it provides modules for diverse scripting and programming languages.

Recommendation

After a thorough look into the topic of OAuth 2.0 and its different flows and implementations, its safe to say that the usage of this framework is a perfect solution to the query of how to access multiple web sites and services by maintaining a single account. This framework boosts simplicity and efficiency. The OAuth 2.0 framework is widely adopted, this research confirms and supports the adoption of the proposed mitigation techniques to tackle each know issue.

Bibliography

(IETF), I. E. (2012, 10). *The OAuth 2.0 Authorization Framework*. Retrieved from https://datatracker.ietf.org/doc/html/rfc6749

Andrukhanenko, O. (2020, 3 17). *Oauth 2.0 Basic Understanding*. Retrieved from stfalcon: https://stfalcon.com/en/blog/post/oauth-2.0

auth0, d. (2013-2022). *Authentication and Authorization Flows*. Retrieved from auth0 docs: https://auth0.com/docs/get-started/authentication-and-authorization-flow

Denniss, W., Bradley, J., Jones, M., & Tschofenig, H. (n.d.). *OAuth 2.0 Device Authorization Grant*. Retrieved from Internet Engineering Task Force (IETF) : https://datatracker.ietf.org/doc/html/rfc8628#page-5

Hughes, A. (2021, 5 5). *How to Use Client Credentials Flow with Spring Security*. Retrieved from okta Developer: https://developer.okta.com/blog/2021/05/05/client-credentials-spring-security

Parecki, A. (2018, 4 10). *What is the OAuth 2.0 Authorization Code Grant Type?* Retrieved from okta Developer: https://developer.okta.com/blog/2018/04/10/oauth-authorization-code-grant-type