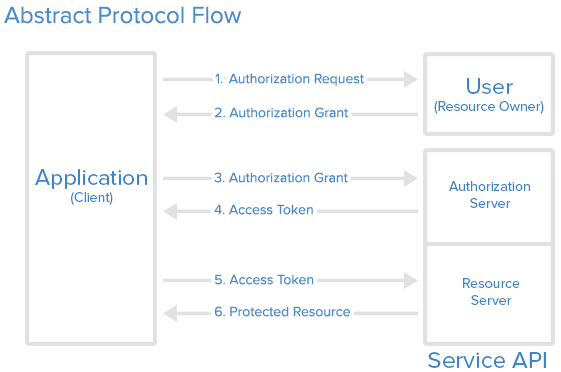
**OAuth2** **is a** **standard that describes how a third-party application can access data from an application on behalf of a user**. OAuth2 doesn’t directly handle authentication and is a more **general framework built primarily for authorization**. For example, a user might grant application access to view their calendar in order to schedule a meeting for you. This would involve an OAuth2 interaction between the user, their calendar provider, and the scheduling application.

OAuth defines four roles:

1. **Client**
   * **The client is the application that wants to access the user’s account**. Before it may do so, it must be authorized by the user, and authorization must be validated by the API.
2. **Resource Owner**
   * **The resource owner is the user who authorizes an application to access their account**. The application’s access to the user’s account is **limited to the scope of the authorization granted** (e.g. read or write access).
3. **Authorization Server**
   * **The authorization server verifies the identity of the user** **then issues access tokens to the application**.
4. **Resource Server**
   * **The resource server hosts the protected user accounts**.

Now that we have an idea of what the OAuth roles are, let’s look at a diagram of how they generally interact with each other:



Here is a more detailed explanation of the steps in the diagram:

1. The **application requests authorization to access service resources** from the user
2. **If the user authorized the request**, the **application receives an authorization grant**
3. The **application requests an access token from the authorization server** **by presenting authentication of its own identity, and the authorization grant**
4. **If the application identity is authenticated and the authorization grant is valid**, the **authorization server issues an access token** **to the application**. Authorization is complete.
5. The **application requests the resource from the resource server and presents the access token for authentication**
6. **If the access token is valid**, **the resource server serves the resource to the application**

**Application Registration**

* Before using OAuth with our application, **we must register our application with the service**. This is **done through a registration form in the developer or API portion of the service’s website**, where we **need to provide the following information**:
* **Application Name**
* **Application Website**
* **Redirect URI or Callback URL**
* The **redirect URI is where the service will redirect the user after they authorize/deny our application**, and therefore the part of our application that will **handle authorization codes or access tokens**.

**Client ID and Client Secret**

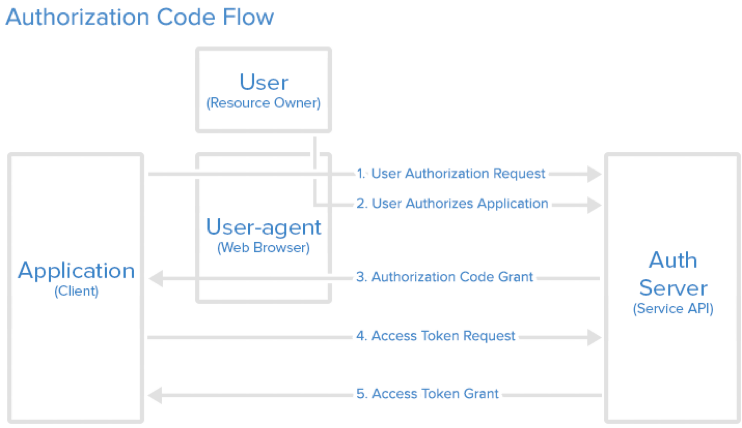
* Once your application is registered**,** the **service will issue client credentials** in the form of **a client identifier** and **a client secret**.
* The **Client ID** is a **publicly exposed string** that is **used by the service API to identify the application** and is also **used to build authorization URLs that are presented to users**.
* The **Client Secret** is **used to authenticate the identity of the application** to the service API **when the application requests to access a user’s account** and **must be kept private between the application and the API**.

**Authorization Grant**

* In the Abstract Protocol Flow outlined previously, the first four steps cover obtaining an authorization grant and access token. The **authorization grant type depends on the method used by the application to request authorization, and the grant types supported by the API**. **OAuth 2 defines three primary grant types**, each of which is useful in different cases:
* **Authorization Code**: **used with server-side applications**
* **Client Credentials**: **used with applications that have API access**
* **Device Code**: **used for devices that lack browsers or have input limitations**

**Grant Type: Authorization Code**

The **authorization code grant type** is the **most commonly used** because it is **optimized for server-side applications**, where **source code is not publicly exposed**, and **Client Secret confidentiality can be maintained**. This is a **redirection-based flow**, which means that the **application must be capable of interacting with the user-agent** (web browser) and receiving API **authorization codes** that **are routed through the user-agent**.



**Step 1 — Authorization Code Link**

First, the user is given an authorization code link that looks like:

[**https://cloud.digitalocean.com/v1/oauth/authorize?response\_type=code&client\_id=CLIENT\_ID&redirect\_uri=CALLBACK\_URL&scope=read**](https://cloud.digitalocean.com/v1/oauth/authorize?response_type=code&client_id=CLIENT_ID&redirect_uri=CALLBACK_URL&scope=read)

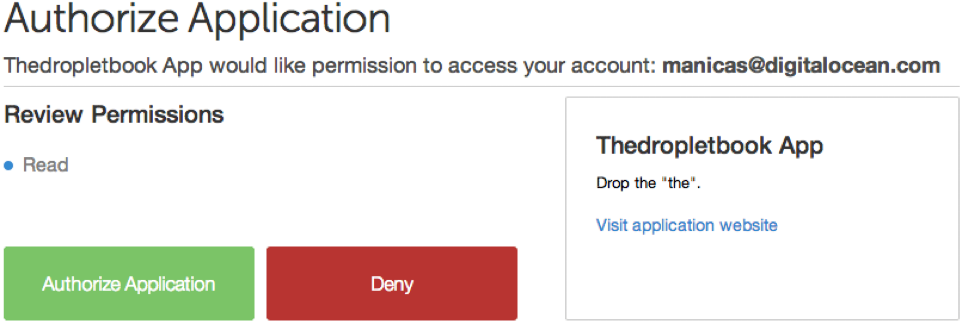
Here is an explanation of the link’s components:

* **response\_type=code**: specifies that your **application is requesting an authorization code grant**
* [**https://cloud.digitalocean.com/v1/oauth/authorize**](https://cloud.digitalocean.com/v1/oauth/authorize): the API **authorization endpoint**
* c**lient\_id=client\_id**: the **application’s client ID** (how the API identifies the application)
* **redirect\_uri=CALLBACK\_URL**: where the **service redirects the user-agent after an authorization code is granted**
* **scope=read**: specifies the **level of access that the application is requesting**

**Step 2 — User Authorizes Application**

When the user clicks the link, they must **first log in to the service to authenticate their identity** (unless they are already logged in). Then they will be **prompted by the service to authorize or deny the application access to their account**.

Here is an example authorize application prompt:



**Step 3 — Application Receives Authorization Code**

If the **user clicks Authorize Application** the service **redirects the user-agent to the application redirect URI**, which was specified during the client registration, **along with an authorization code**.

The redirect URI would look something like this (assuming the application is [dropletbook.com](http://dropletbook.com)):

[**https://dropletbook.com/callback?code=AUTHORIZATION\_CODE**](https://dropletbook.com/callback?code=AUTHORIZATION_CODE)

**Step 4 — Application Requests Access Token**

The **application requests an access token from the API by passing the authorization code along with authentication details**, **including the client secret, to the API token endpoint**.

Here is an example POST request to DigitalOcean’s token endpoint:

[**https://cloud.digitalocean.com/v1/oauth/token?client\_id=CLIENT\_ID&client\_secret=CLIENT\_SECRET&grant\_type=authorization\_code&code=AUTHORIZATION\_CODE&redirect\_uri=CALLBACK\_URL**](https://cloud.digitalocean.com/v1/oauth/token?client_id=CLIENT_ID&client_secret=CLIENT_SECRET&grant_type=authorization_code&code=AUTHORIZATION_CODE&redirect_uri=CALLBACK_URL)

**Step 5 — Application Receives Access Token**

If the **authorization is valid**, the **API will send a response containing the access token** (and **optionally, a refresh token**) to the application.

The entire response will look something like this:

{

“access\_token":"ACCESS\_TOKEN",

“token\_type":"bearer",

“expires\_in":2592000,

“refresh\_token":"REFRESH\_TOKEN",

“scope”:"read",

"uid":100101,

"info":{"name":"Mark E. Mark”,"email":"mark@thefunkybunch.com"}

}

Now the **application is authorized**. It may **use the token to access the user’s account via the service API**, **limited to the scope of access until the token expires or is revoked**. **If a refresh token was issued, it may be used to request new access tokens if the original token has expired**.

**Note Regarding Proof Key for Code Exchange**

**If a public client is using the Authorization Code grant type**, there’s **a chance that the authorization code could be intercepted**. The **Proof Key for Code Exchange** (or **PKCE**, pronounced like **“pixie”**) is an **extension to the Authorization Code flow that helps to mitigate this kind of attack**.

The **PKCE extension** involves the **client creating and recording a secret key** — known as **a code verifier** — **for every authorization request**. The client then **transforms the code verifier into a code challenge** and **sends the code verifier, the code challenge, and the transformation method to the authorization endpoint in the same authorization request**.

The **authorization endpoint records the code challenge and the transformation method** and **responds with the authorization code** as outlined previously. The **client then sends in the access token request** **which includes the code verifier it originally generated**.

**After receiving the code verifier**, the **authorization server transforms it into the code challenge using the transformation method first shared by the client**. If the code challenge derived from the code verifier sent by the client doesn’t match the one originally recorded by the authorization server, then the authorization server will deny the client access.

It’s recommended that every client use the PKCE extension for improved security.

**Grant Type: Client Credentials**

The **client credentials grant type provides an application a way to access its own service account**. Examples of when this might be useful include if an **application wants to update its registered description or redirect URI or access other data stored in its service account** via the API.

The **application requests an access token by sending its credentials, its client ID, and client secret, to the authorization server**.

An example POST request might look like the following:

<https://oauth.example.com/token?grant_type=client_credentials&client_id=CLIENT_ID&client_secret=CLIENT_SECRET>

If the application credentials check out, the **authorization server returns an access token to the application**. Now the **application is authorized to use its own account**.

**Grant Type: Device Code**

The **device code grant type** provides a means for **devices that lack a browser or have limited inputs to obtain an access token and access a user’s account**. The purpose of this grant type is to make it easier for users to easily authorize applications on such devices to access their accounts. Examples of when this might be useful include if a **user wants to sign into a video streaming app** **on a device that doesn’t have a typical keyboard input**, such as a smart television or a video game console.

The **user starts an application on their browserless or input-limited device**, such as a television or a set-top box. The **application submits a POST request to a device authorization endpoint**.

An example device code POST request might look like the following:

POST<https://oauth.example.com/device>

client\_id=CLIENT\_id

The device authorization endpoint is different from the authentication server, as **the device authorization endpoint doesn’t actually authenticate the device**. Instead, it **returns a unique device code, which is used to identify the device**; **a user code, which the user can enter on a machine on which it’s easier to authenticate**, such as a laptop or mobile device; and **the URL the user should visit to enter the user code and authenticate their device**.

Here’s what an example response from the device authorization endpoint might look like:

{

"device\_code": "IO2RUI3SAH0IQuESHAEBAeYOO8UPAI",

"user\_code": "RSIK-KRAM",

"verification\_uri": "https://example.okta.com/device",

"interval": 10,

"expires\_in": 1600

}

Note that the **device code could also be a QR code that the reader can scan on a mobile device**.

The **user then enters the user code at the specified URL** and **signs into their account**. They are then **presented with a consent screen where they can authorize the device to access their account**.

While the **user visits the verification URL and enters their code**, the **device will poll the access endpoint until it returns an error or an authentication token**. The **access endpoint will return errors** if the **device is polling too frequently** (the **slow\_down** error) if the **user hasn’t yet approved or denied the request** (the **authorization\_pending** error), if the **user has denied the request** (the **access\_denied** error), or if the **token has expired** (the **expired\_token** error).

**If the user approves the request, though, the access endpoint will return an authentication token.**

**Example Access Token Usage**

Once the application has an access token, it may use the token to access the user’s account via the API, limited to the scope of access, until the token expires or is revoked.

Here is an example of an API request, using curl. Note that it includes the access token:

curl -X POST -H "Authorization: Bearer ACCESS\_TOKEN" "https://api.digitalocean.com/v2/$OBJECT"

Assuming the access token is valid, the API will process the request according to its API specifications. If the **access token is expired or invalid**, the API will return an **invalid\_request** error.

After an access token expires, using it to make a request from the API will result in an **invalid\_token** Error. At this point, if a **refresh token** was included when the original access token was issued, it **can be used to request a fresh access token from the authorization server**.

Here is an example POST request, using a refresh token to obtain a new access token:

https://cloud.digitalocean.com/v1/oauth/token?grant\_type=refresh\_token&client\_id=CLIENT\_ID&client\_secret=CLIENT\_SECRET&refresh\_token=REFRESH\_TOKEN