The mechanisms of **OAuth 2.0**, **OpenID Connect (OIDC)**, and **API Keys** for API authentication and authorization, along with examples of how each approach works.

**1. OAuth 2.0**

[**https://fusionauth.io/articles/oauth/modern-guide-to-oauth**](https://fusionauth.io/articles/oauth/modern-guide-to-oauth)

**OAuth** (short for **open authorization**) is an open standard for **access delegation**, commonly used as a way for internet users to grant websites or applications access to their information on other websites but without giving them the passwords. This mechanism is used by companies such as Amazon, Google, Meta Platforms, Microsoft, and Twitter to permit users to share information about their accounts with third-party applications or websites. [[OAuth - Wikipedia](https://en.wikipedia.org/wiki/OAuth)]

OAuth 2.0 is an **authorization framework** that allows applications to access resources (such as APIs) on behalf of a user without exposing their credentials. OAuth in general was designed as a reaction to the [direct authentication pattern](http://guides.brucejmack.net/SOA-Patterns/WSSP/1.3DirectvsBrokeredAuthDoc.htm), which features applications requesting usernames and passwords when accessing user-specific data at other services.

When utilizing OAuth, the user still provides this information, but they log into one platform and use tokens generated by that platform to grant access to data and perform actions in one or more other applications. These applications may be owned by third parties or by the same folks who control the authentication platform. This flexibility is one reason for confusion when it comes to OAuth.

It’s commonly used when granting access to third-party applications (e.g., a travel app accessing your Google Calendar). OAuth 2.0 is built around tokens that serve as temporary access "keys" to the API.

A token has a limited lifetime and provides constrained access. Connecting multiple applications is easier for users. It is also more secure since [user credentials](https://fusionauth.io/articles/ciam/challenges-of-ciam) are not shared. Auth is also easier for developers to manage because they only need to integrate OAuth 2.0 in their application instead of having their own database to store users’ sensitive information.

OAuth and SAML are sometimes confused. SAML is primarily an authentication system used for single sign-on, while OAuth is an authorization system that has authentication layered on top of it (OpenID Connect, or OIDC). Authentication is about confirming the user’s identity. Authorization is about deciding what services, functionality or data they can access. A detailed comparison between [OAuth and SAML is found here](https://fusionauth.io/articles/oauth/saml-vs-oauth) .

The application sends the user over to an OAuth server, the user logs in, and then the user is sent back to your application. But. There are a couple of different twists and goals of this process.

**Key Components**

* **Resource Owner**: The user who owns the data and authorizes an *application* to access their account/data. The application’s access to the user’s account/data is limited to the scope of the authorization granted (e.g. read or write access)
* **Client**: The application requesting access to the resource.
* **Authorization Server**: The server that authenticates the user and issues tokens.
* **Resource Server**: The server hosting the protected resource.

OAuth framework specifies several grant types for different use cases. **Some of the most common OAuth grant types** are: (https://oauth.net/2/grant-types/)

* Authorization Code
* PKCE (Proof Key for Code Exchange - pronounced Pixy)
* Client Credentials
* Device Code
* Refresh Token

**How Does OAuth 2.0 Work?**

None of the specifications cover how OAuth is actually integrated into applications. Whoops! But as a developer, that’s what you care about. They also don’t cover the different workflows or processes that leverage OAuth. They leave almost everything up to the implementer (the person who writes the OAuth Server) and integrator (the person who integrates their application with that OAuth server).

There are eight **OAuth modes** in common use today. These real world OAuth modes are:

1. Local login and registration
2. Third-party login and registration *(federated identity)*
3. First-party login and registration *(reverse federated identity)*
4. Enterprise login and registration *(federated identity with a twist)*
5. Third-party service authorization
6. First-party service authorization
7. Machine-to-machine authentication and authorization
8. Device login and registration

**Which OAuth Mode Is Right for You? (**[**https://fusionauth.io/articles/oauth/modern-guide-to-oauth**](https://fusionauth.io/articles/oauth/modern-guide-to-oauth)**)**

Wow, that’s a lot of different ways you can use OAuth. That’s the power and the danger of OAuth, to be honest with you. It is so flexible that people new to it can be overwhelmed. So, here’s a handy set of questions for you to ask yourself.

* Are you looking to outsource your authentication and authorization to a safe, secure and standards-friendly auth system? You’ll want [Local login and registration](https://fusionauth.io/articles/oauth/modern-guide-to-oauth#local-login-and-registration) in that case.
* Trying to avoid storing any credentials because you don’t want responsibility for passwords? [Third-party login and registration](https://fusionauth.io/articles/oauth/modern-guide-to-oauth#third-party-login-and-registration) is where it’s at.
* Are you selling to Enterprise customers? Folks who hear terms like SAML and SOC2 and are comforted, rather than disturbed? Scoot on over to [Enterprise login and registration](https://fusionauth.io/articles/oauth/modern-guide-to-oauth#enterprise-login-and-registration).
* Are you building service to service communication with no user involved? If so, you are looking for [Machine-to-machine authorization](https://fusionauth.io/articles/oauth/modern-guide-to-oauth#machine-to-machine-authorization).
* Are you trying to let a user log in from a separate device? That is, from a TV or similar device without a friendly typing interface? If this is so, check out [Device login and registration](https://fusionauth.io/articles/oauth/modern-guide-to-oauth#device-login-and-registration).
* Are you building a platform and want to allow other developers to ask for permissions to make calls to APIs or services on your platform? Put on your hoodie and review [First-party login and registration](https://fusionauth.io/articles/oauth/modern-guide-to-oauth#first-party-login-and-registration) and [First-party service authorization](https://fusionauth.io/articles/oauth/modern-guide-to-oauth#first-party-service-authorization).
* Do you have a user store already integrated and only need to access a third party service on your users’ behalf? Read up on [Third-party service authorization](https://fusionauth.io/articles/oauth/modern-guide-to-oauth#third-party-service-authorization).

With that overview done, let’s examine each of these modes in more detail.

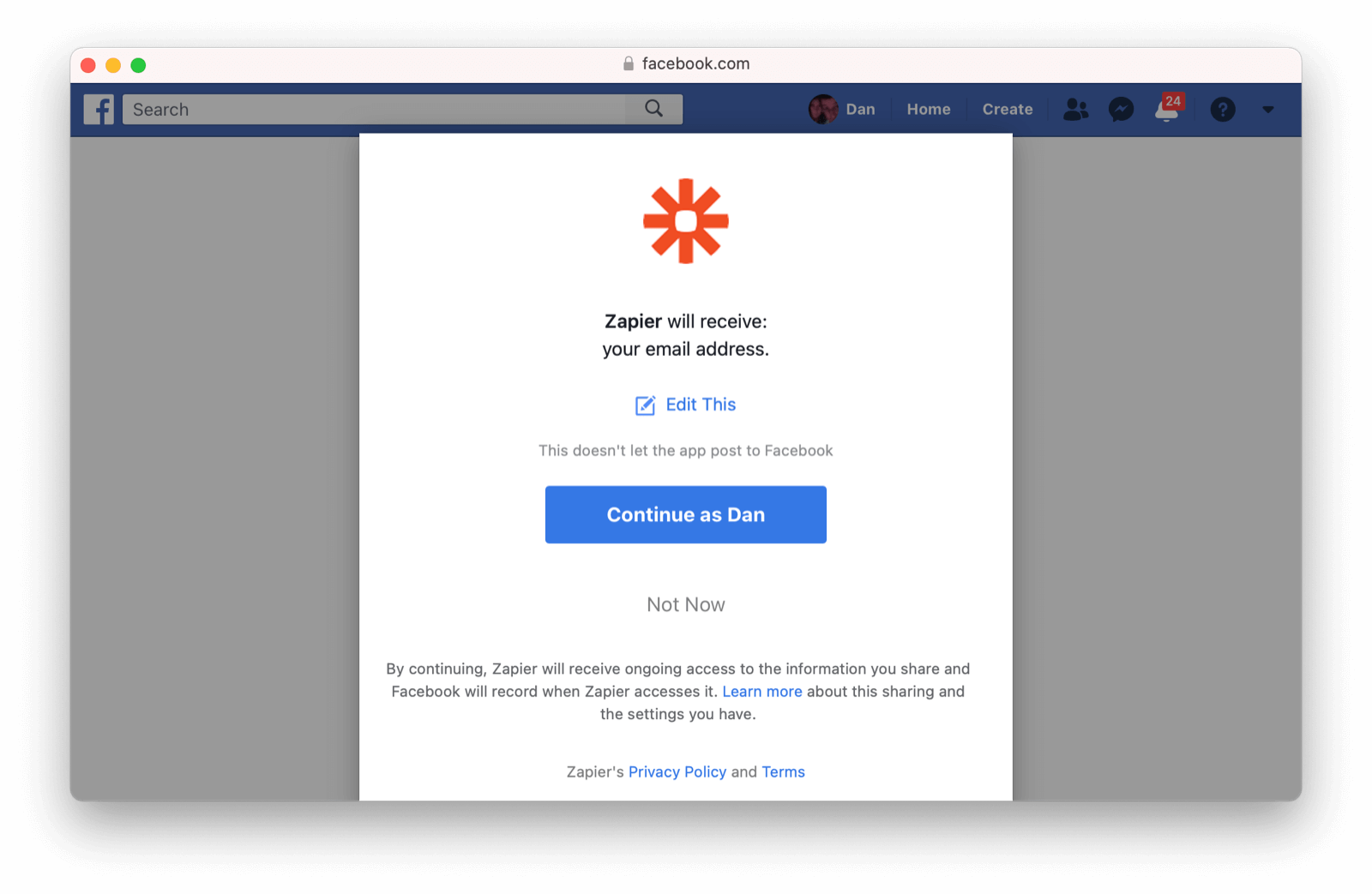
**Third-party Login and Registration**

The **Third-party login and registration** mode is typically implemented with the classic “**Login with …**” buttons you see in many applications. These buttons let users sign up or log in to your application by logging into one of their other accounts (i.e. Facebook or Google). Here, your application sends the user over to Facebook or Google to log in.

Let’s use Facebook as an example OAuth provider. In most cases, your application will need to use one or more APIs from the OAuth provider in order to retrieve information about the user or do things on behalf of the user (for example sending a message on behalf of the user). In order to use those APIs, the user has to grant your application permissions. To accomplish this, the third-party service usually shows the user a screen that asks for certain permissions. We’ll refer to these screens as the “permission grant screen” throughout the rest of the guide.

For example, Facebook will present a screen asking the user to share their email address with your application. Once the user grants these permissions, your application can call the Facebook APIs using an access token (which we will cover [later in this guide](https://fusionauth.io/articles/oauth/modern-guide-to-oauth#third-party-login-and-registration-also-enterprise-login-and-registration-with-the-authorization-code-grant)).

Here’s an example of the Facebook permission grant screen, where Zapier would like to access a user’s email address:



After the user has logged into the third-party OAuth server and granted your application permissions, they are redirected back to your application and logged into it.

This mode is different from the previous mode because the user logged in but also granted your application permissions to the service (Facebook). This is one reason so many applications leverage “Login with Facebook” or other social integrations. It not only logs the user in, but also gives them access to call the Facebook APIs on the user’s behalf.

This mode is a good example of federated identity. Here, the user’s identity (username and password) is stored in the third-party system. They are using that system to register or log in to your application.

So, how does this work in practice? Let’s take a look at the steps for our TWGTL application if we want to use Facebook to register and log users in:

1. A user visits TWGTL and wants to sign up and manage their ToDos.
2. They click the “Sign Up” button on the homepage.
3. On the login and registration screen, the user clicks the “Login with Facebook” button.
4. This button takes them over to Facebook’s OAuth server.
5. They log in to Facebook (if they aren’t already logged in).
6. Facebook presents the user with the permission grant screen based on the permissions TWGTL needs. This is done using OAuth scopes, covered above.
7. Facebook redirects the browser back to TWGTL, which logs the user in. TWGTL also calls Facebook APIs to retrieve the user’s information.
8. The user begins using TWGTL and adds their current ToDos.
9. The user stops using TWGTL; they head off and do some ToDos.
10. Later, the user comes back to TWGTL and needs to log in to check off some of their ToDos. They click the **My Account** link at the top of the page.
11. This takes the user to the TWGTL login screen that contains the “Login with Facebook” button.
12. Clicking this takes the user back to Facebook and they repeat the same process as above.

**OAuth 2.0 Flow (Authorization Code Grant)**

Let's look at the Authorization Code grant type, which is commonly used for server-side apps:

1. **Authorization Request**: The client (app) sends a request to the authorization server with:
   * client\_id: the app's unique identifier.
   * redirect\_uri: the callback URL to receive the authorization code.
   * response\_type=code: specifies the response should be an authorization code.
   * scope: permissions the app is requesting.
   * Example URL:

https://authorization-server.com/auth?client\_id=abc123&redirect\_uri=https://myapp.com/callback&response\_type=code&scope=email profile

**The parameters defined in the OAuth specifications** are:

* **client\_id** - this identifies the application you are logging into. In OAuth, this is referred to as the **client**. This value will be provided to you by the OAuth server.
* **redirect\_uri** - this is the URL in your application to which the OAuth server will redirect the user to after they log in. This URL must be registered with the OAuth server and it must point to a controller in your app (rather than a static page), because your app must do additional work after this URL is called.
* **state** - technically this parameter is optional, but it is useful for preventing various security issues. This parameter is echoed back to your application by the OAuth server. It can be anything you might need to be persisted across the OAuth workflow. If you have no other need for this parameter, I suggest setting it to a large random string. If you need to have data persisted across the workflow, I suggest URL encoding the data and appending a random string as well.
* **response\_type** - this should always be set to **code** for this grant. This tells the OAuth server you are using the Authorization Code grant.
* **scope** - this is also an optional parameter, but in some of the above modes, this will be required by the OAuth server. This parameter is a space separated list of strings. You might also need to include the **offline** scope in this list if you plan on using refresh tokens in your application (we’ll refresh tokens later).

Here are definitions of the standard scopes in the OpenID Connect specification:

* **openid** - tells the OAuth server to use OpenID Connect for the handling of the OAuth workflow. This additionally will tell the OAuth server to return an Id token from the Token endpoint (covered below).
* **offline\_access** - tells the OAuth server to generate and return a refresh token from the Token endpoint (covered below).
* **profile** - tells the OAuth server to include all of the standard OpenID Connect claims in the returned tokens (access and/or Id tokens).
* **email** - tells the OAuth server to include the user’s email in the returned tokens (access and/or Id tokens).
* **address** - tells the OAuth server to include the user’s address in the returned tokens (access and/or Id tokens).
* **phone** - tells the OAuth server to include the user’s phone number in the returned tokens (access and/or Id tokens).
* **code\_challenge** - this an optional parameter, but provides support for PKCE. This is useful when there is not a backend that can handle the final steps of the Authorization Code grant. This is known as a “public client”. There aren’t many cases of applications that don’t have backends, but if you have something like a mobile application and you aren’t able to leverage a server-side backend for OAuth, you must implement PKCE to protect your application from security issues. The security issues surrounding PKCE are out of the scope of this guide, but you can find numerous articles online about them. PKCE is also recommended by the [OAuth 2.1 draft](https://fusionauth.io/articles/oauth/differences-between-oauth-2-oauth-2-1).
* **code\_challenge\_method** - this is an optional parameter, but if you implement PKCE, you must specify how your PKCE **code\_challenge** parameter was created. It can either be **plain** or **S256**. We never recommend using anything except **S256** which uses SHA-256 secure hashing for PKCE.
* **nonce** - this is an optional parameter and is used for OpenID Connect. We don’t go into much detail of OpenID Connect in this guide, but we will cover a few aspects including Id tokens and the **nonce** parameter. The **nonce** parameter will be included in the Id token that the OAuth server generates. We can verify that when we retrieve the Id token.

1. **Authorization Grant**: The user is redirected to a login page (on the authorization server) and, upon successful login, grants the requested permissions. They are then redirected back to the redirect\_uri with an authorization code.
2. **Token Exchange**: The client exchanges the authorization code for an access token by making a POST request to the authorization server’s token endpoint. The request includes:
   * client\_id and client\_secret: the app’s credentials.
   * grant\_type=authorization\_code.
   * Example Request:

POST /token

Host: authorization-server.com

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

grant\_type=authorization\_code&code=AUTH\_CODE&redirect\_uri=https://myapp.com/callback&client\_id=abc123&client\_secret=shh\_its\_a\_secret

1. **Access Token**: The authorization server responds with an access token (and often a refresh token), allowing the client to interact with the resource server.
2. **API Access**: The client includes the access token in API requests as a Bearer token:

GET /userinfo

Host: api.example.com

Authorization: Bearer ACCESS\_TOKEN

1. **Refresh Token (Optional)**: If the access token expires, the refresh token can be used to request a new access token without re-prompting the user.

**access\_token**: This is a token that contains information about the user including their id, permissions, and anything else we might need from the OAuth server. It is often, but not always, a JWT.

**id\_token**: This is a JWT that contains public information about the user such as their name. This token is usually safe to store in non-secure cookies or local storage because it can’t be used to call APIs on behalf of the user.

**refresh\_token**: This is an opaque token (not a JWT) that can be used to create new access tokens. Access tokens expire and might need to be renewed, depending on your requirements (for example how long you want access tokens to last versus how long you want users to stay logged in).

Since two of the tokens we have are JWTs, let’s quickly cover that technology here. A full coverage of JWTs is outside of the scope of this guide, but there are a couple of good JWT guides in our [Token Expert Advice section](https://fusionauth.io/articles/tokens/).

JWTs are JSON objects that contain information about users and can also be signed. The keys of the JSON object are called “claims”. JWTs expire, but until then they can be presented to APIs and other resources to obtain access. Keep their lifetimes short and protect them as you would other credentials such as an API key. Because they are signed, a JWT can be verified to ensure it hasn’t been tampered with. JWTs have a couple of standard claims. These claims are:

* aud: The intended audience of the JWT. This is usually an identifier and your applications should verify this value is as expected.
* exp: The expiration instant of the JWT. This is stored as the number of seconds since Epoch (January 1, 1970 UTC).
* iss: An identifier for that system which created the JWT. This is normally a value configured in the OAuth server. Your application should verify that this claim is correct.
* nbf: The instant after which the JWT is valid. It stands for “not before”. This is stored as the number of seconds since Epoch (January 1, 1970 UTC).
* sub: The subject of this JWT. Normally, this is the user’s id.

JWTs have other standard claims that you should be aware of. You can review these specifications for a list of additional standard claims:

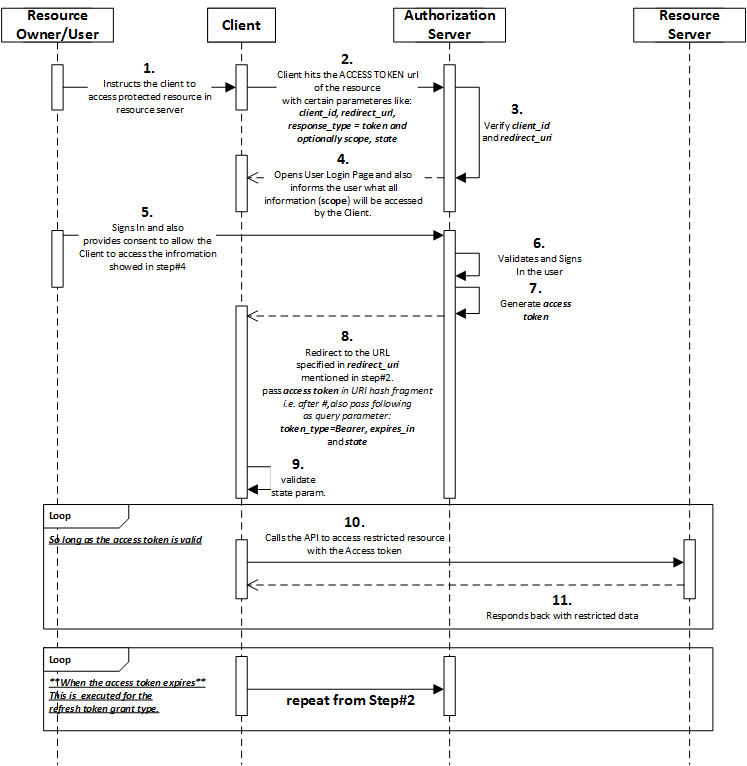
* [JWT Claims in the JSON Web Token RFC](https://tools.ietf.org/html/rfc7519#section-4)
* [Claims in the OpenID Connect 1.0 spec](https://openid.net/specs/openid-connect-core-1_0.html#Claims)

User And Token Information

Before we cover how the Authorization Code grant is used for each of the OAuth modes, let’s discuss two additional OAuth endpoints used to retrieve information about your users and their tokens. These endpoints are:

* Introspection - this endpoint is an extension to the OAuth 2.0 specification and returns information about the token using the standard JWT claims from the previous section.
* UserInfo - this endpoint is defined as part of the OIDC specification and returns information about the user.

These two endpoints are quite different and serve different purposes. Though they might return similar values, the purpose of the Introspection endpoint is to return information about the access token itself. The UserInfo endpoint returns information about the user for whom the access token was granted.



**In Step #9, *Client* will compare the value of *state* parameter to ensure that it’s the same value that client sent in Step#2. This is to avoid any CSRF attack.**

**The above diagram is from** [**Tutorial on understanding oAuth2 Implicit Grant Flow**](https://iteritory.com/tutorial-on-oauth2-implicit-grant-flow/)

**How the Bearer Token is verified at resource server**

When a client includes a Bearer access token in an API request, the resource server needs a way to verify if the token is valid, not expired, and issued by a trusted authorization server. Here are the common methods used to verify the access token:

**1. Token Introspection Endpoint (OAuth 2.0 Introspection)**

* The resource server can call an introspection endpoint provided by the authorization server to validate the access token in real time.
* This endpoint checks if the token is active (not expired or revoked) and may return metadata such as user ID, scope, and expiry.

**Example Request**

POST /introspect

Host: authorization-server.com

Authorization: Basic base64(client\_id:client\_secret)

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

token=ACCESS\_TOKEN

**Example Response**

{

"active": true,

"scope": "read write",

"client\_id": "abc123",

"username": "jdoe",

"exp": 1619030047

}

* **Active Field**: Indicates if the token is valid (e.g., true or false).
* **Scope, Username, Expiration**: Provides additional information about the token and can help enforce authorization.

**2. Self-Contained Tokens (e.g., JWTs)**

* Access tokens are often structured as JSON Web Tokens (JWTs), which are self-contained and include all the necessary data, like user claims, scope, and expiration.
* Since JWTs are signed by the authorization server (typically using an asymmetric key like RSA or HMAC for symmetric keys), the resource server can verify the token locally by:
  1. **Validating the Signature**: The resource server checks the JWT signature using the authorization server’s public key (for RSA) or a shared secret (for HMAC).
  2. **Checking Expiry and Claims**: It ensures the exp (expiration) claim is in the future and that the iss (issuer) claim matches the trusted authorization server.

**Example JWT Structure**

A JWT is structured as three parts: header.payload.signature. Here’s an example of a JWT payload decoded:

{

"sub": "1234567890",

"name": "John Doe",

"iat": 1516239022,

"exp": 1616239022,

"iss": "https://authorization-server.com"

}

* **sub**: Subject, usually the user ID.
* **iat**: Issued-at time.
* **exp**: Expiry timestamp.
* **iss**: Issuer.

The resource server verifies the exp to ensure the token isn’t expired and validates the iss to confirm it came from a trusted source.

**3. Cached Introspection or JWT Verification**

* To reduce the frequency of requests to the authorization server, resource servers can cache the validation result (for introspected tokens) or cache the public keys used for JWT signature verification.
* The cached data is refreshed periodically, allowing the resource server to validate tokens quickly without querying the authorization server for each request.

**Example Flow**

Suppose a request is received:

GET /userinfo

Host: api.example.com

Authorization: Bearer ACCESS\_TOKEN

1. The resource server extracts the ACCESS\_TOKEN from the header.
2. Depending on the token type:
   * **Introspection Endpoint**: It calls the authorization server’s introspection endpoint to verify the token.
   * **JWT Verification**: If it’s a JWT, the resource server checks the signature and claims directly.
3. If the token is valid, the request proceeds. Otherwise, it returns a 401 Unauthorized response.

**2. OpenID Connect (OIDC)**

OIDC is an identity layer on top of OAuth 2.0, primarily used for user authentication. While OAuth 2.0 provides authorization, OIDC adds user authentication by returning an ID token, allowing the client to confirm the user’s identity.

**OIDC Flow Example**

The flow is similar to OAuth 2.0 but with additional ID-specific scopes and an ID token in the response.

1. **Authorization Request**: The client sends a request similar to OAuth 2.0, with scope set to include openid (e.g., scope=openid email profile).
2. **Token Exchange**: During token exchange, the response includes both an access\_token and an id\_token.
3. **ID Token**: The id\_token is a JWT (JSON Web Token) containing user identity information. It can be decoded to access information like:
   * sub: the unique user ID.
   * name, email, etc., based on the requested scopes.
4. **API Access and Authentication**:
   * The client uses the access\_token for API access.
   * The id\_token can be used for authenticating the user, verifying their identity information in the JWT payload.

OIDC is widely used in Single Sign-On (SSO) scenarios. For example, logging in with Google often involves OIDC, where Google acts as the authorization and authentication provider.

**3. API Keys**

API keys are simple tokens that are often used to authenticate applications, particularly when there’s no user context (e.g., server-to-server communication).

**Example Usage**

1. **API Key Generation**: An API key is generated by the server and shared with the client. It’s typically a long, random string like 12345abcdef67890.
2. **API Request with API Key**: The client includes the API key in every request, usually in the headers or as a query parameter:
   * In Headers:

http

Copy code

GET /data

Host: api.example.com

X-API-Key: 12345abcdef67890

* + As a Query Parameter:

bash

Copy code

GET /data?api\_key=12345abcdef67890

**Pros and Cons of API Keys**

* **Pros**:
  + Simple to implement.
  + Useful for low-risk or anonymous access.
* **Cons**:
  + Limited security; keys can be shared, compromised, and do not inherently verify user identity.
  + Often lack scope or permission granularity.

**Summary**

* **OAuth 2.0** is ideal for granular, secure access to resources, especially with third-party applications.
* **OpenID Connect (OIDC)** adds an identity layer to OAuth, allowing user authentication along with authorization.
* **API Keys** are simple to implement for server-to-server or low-risk API access but lack user-specific context.