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**

**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**. 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.

**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
* Client Credentials
* Device Code
* Refresh Token

**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

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

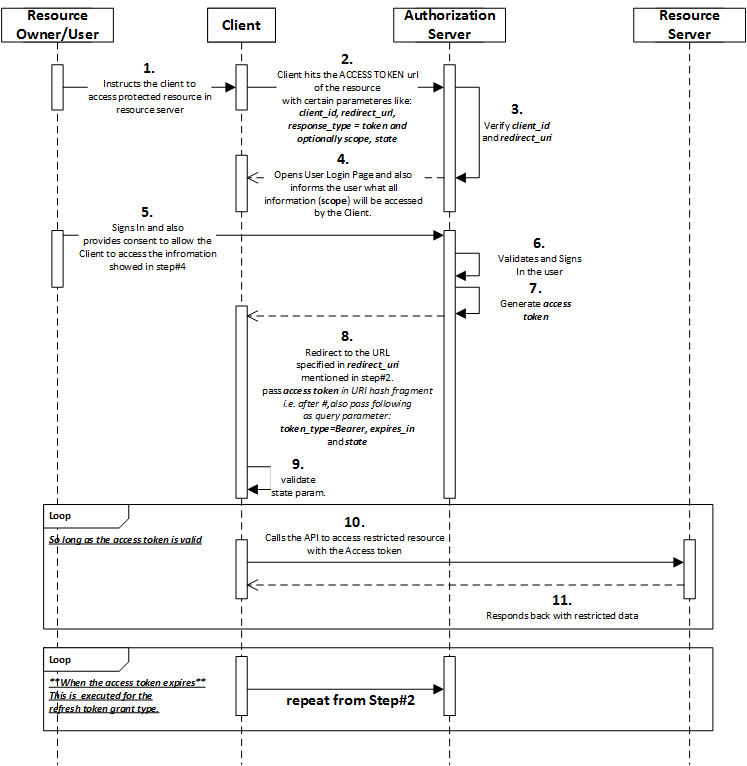
1. **Access Token**: The authorization server responds with an access token (and often a refresh token), allowing the client to interact with the resource server.
2. **API Access**: The client includes the access token in API requests as a Bearer token:

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.



**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.

References:

**What is OAuth 2.0 and How does it Work? https://fusionauth.io/articles/oauth/modern-guide-to-oauth**

**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.