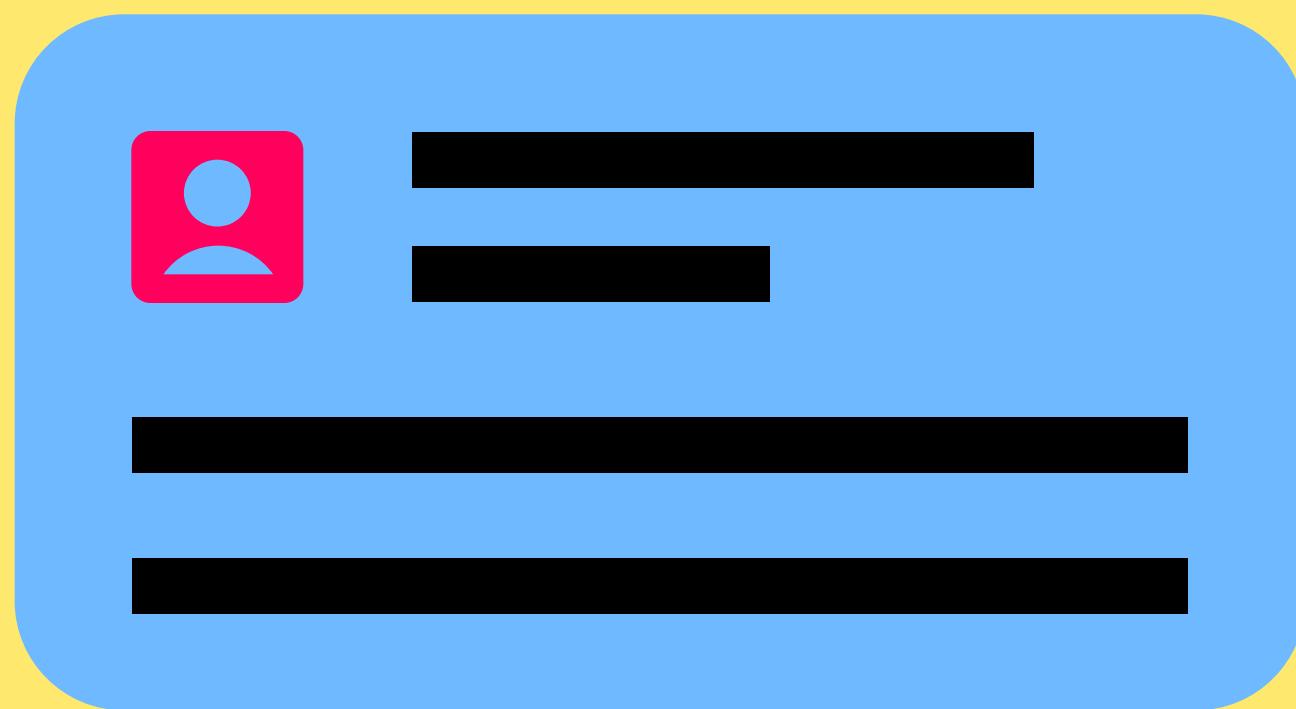
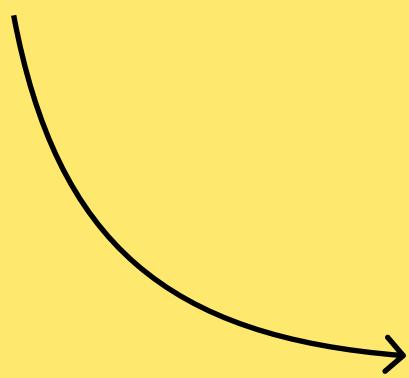


REST API Authentication Methods

**Unlock the Power of REST
APIs with Secure and
Reliable Authentication
Methods**

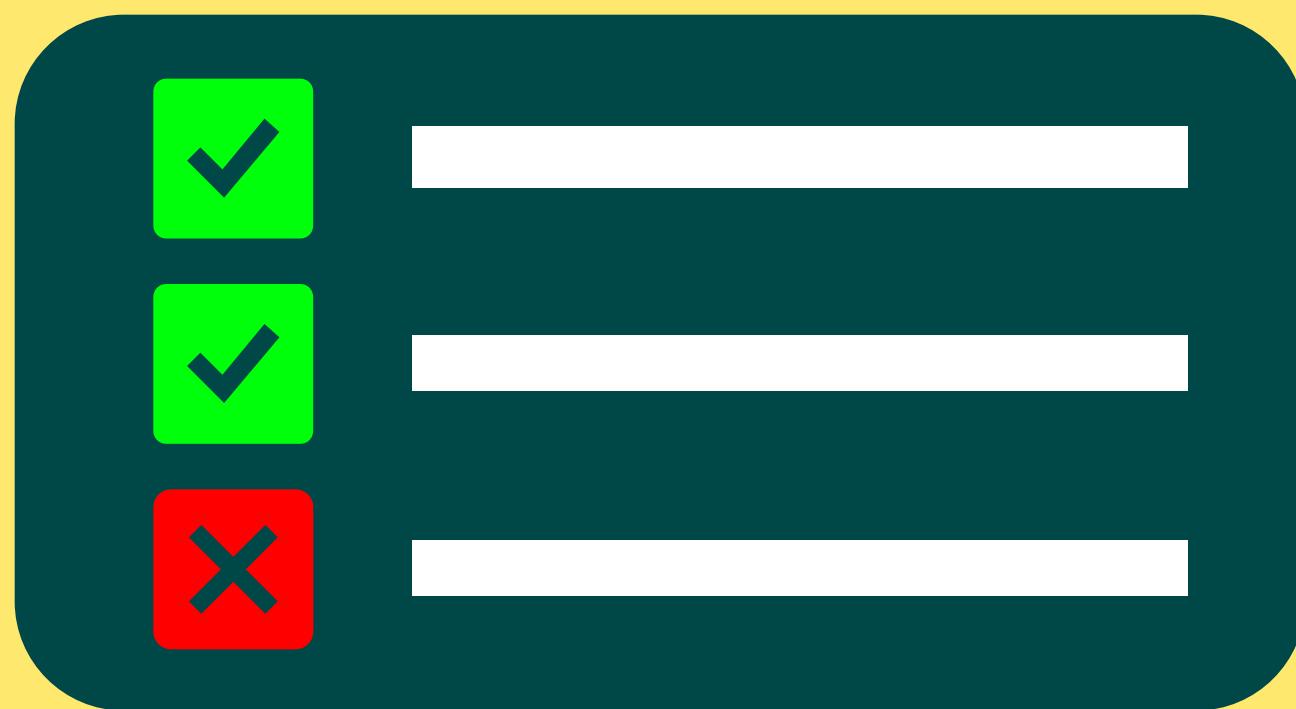


Diagram



Authentication

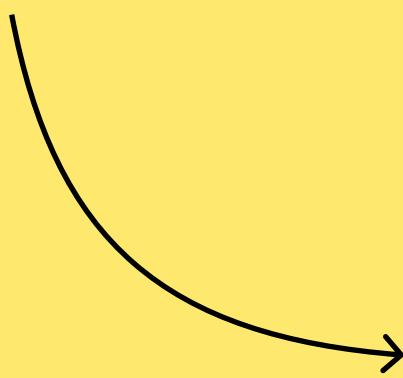
Who you are



Authorization

What you can do

Authentication

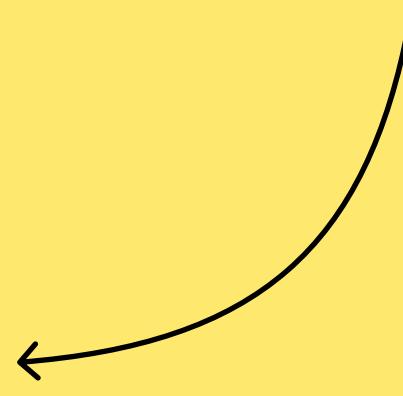


Authentication is the process of proving that you are who you say you are. It is the process of **proving your identity**.

Example

An employee can enter the office if their identity is verified using an **ID card**.

Authorization



Authorization is the process of determining **what actions you are allowed to take** based on your identity and permissions.

Example

An employee may be allowed into the office but **may not be allowed** into the server room.

In summary, **authentication** refers to proving correct identity and **authorization** refers to allowing a certain action. An **API** might authenticate you but not authorize you to make a certain request.

Authentication Methods

- Basic and Bearer
- API Keys
- OAuth (2.0)
- OpenID Connect

These are the four most common **Authentication Methods**

Let us now take a closer look at them

Basic Authentication

HTTP Basic Authentication is rarely recommended due to its inherent **security vulnerabilities**.

This is the most **straightforward** method and the **easiest**. With this method, the sender places a **username:password** into the request header.

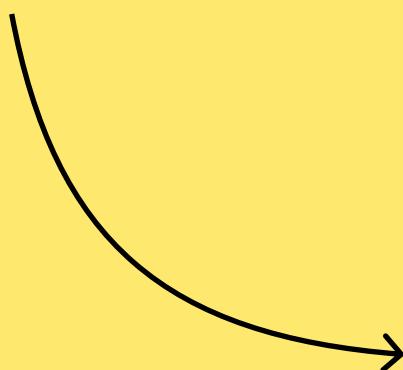
Example of a Basic Auth in a request header

Authorization: Basic bG9sOnNIY3VyZQ==

The username and password are **encoded with Base64**, which is an encoding technique that converts the username and password into a set of 64 characters to **ensure safe transmission**.

This method does not require cookies, session IDs, login pages, and other such specialty solutions, and because it **uses the HTTP header itself**, there's no need to handshakes or other complex response systems.

Bearer Authentication



Bearer authentication (also called **token authentication**) is an HTTP authentication scheme that involves **security tokens** called bearer tokens.

Bearer authentication refers to a method of granting access to a specific resource or URL by presenting a **bearer token**. This token, often a **complex string** generated by a server upon receiving a login request, serves as authorization for the bearer to access the specified resource. In other words, the name “bearer authentication” can be interpreted as granting access to whoever possesses the designated token.

The client must send this token in the Authorization header when making requests to protected resources:

Authorization: Bearer <token>

Bearer authentication was first introduced in [OAuth 2.0](#) through the [RFC-6750](#). It can also be used independently from OAuth. Like Basic authentication, Bearer authentication should only be utilized with [HTTPS \(SSL\)](#) for secure communication.

API Keys

A **unique value** is created and assigned to **each first-time user** in this method as a way of identifying them. This value serves as a **marker** that indicates the **user is known**.

API keys were introduced as a **solution to the authentication problems** that were present in earlier systems such as **HTTP Basic Authentication**.

When a user tries to access the system again, they must provide a **unique key** to prove their identity. This key may be generated based on the **user's hardware and IP data**, or it may be randomly generated by the server that recognizes the user. The purpose of this key is to confirm that the user is the **same individual** who was previously granted access to the system.

API keys are a common standard in the industry, but they should **not be considered a strong security measure**. Despite their widespread use, it is important to consider alternative measures to ensure the **security of sensitive information**.

API Keys - Diagram

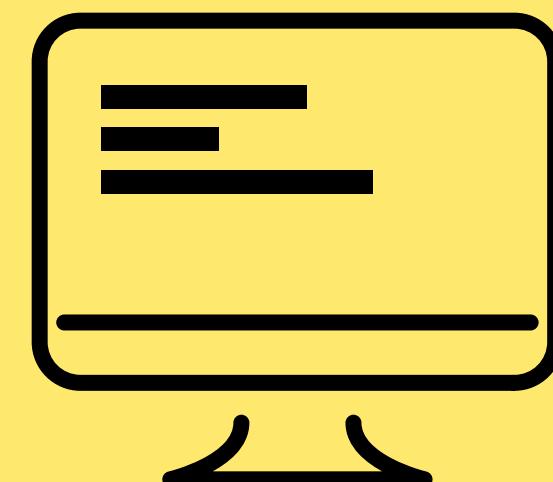
REST API



API Key

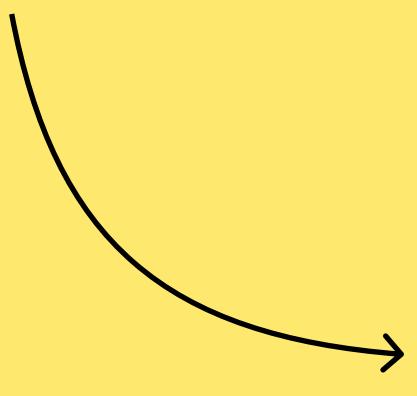
Request Header

{“api-key” : “9038-20380-9398”}



Application

API Keys



API keys are often included in the **query string of URLs**, making it easier for **unauthorized individuals** to discover and access them.

Avoid including API keys or other **sensitive information** in query string parameters. Instead, consider placing the API key in the **Authorization header** for greater security.



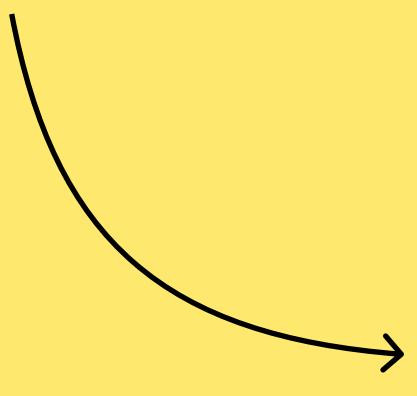
Example

Authorization: Apikey 1234567890abcdef

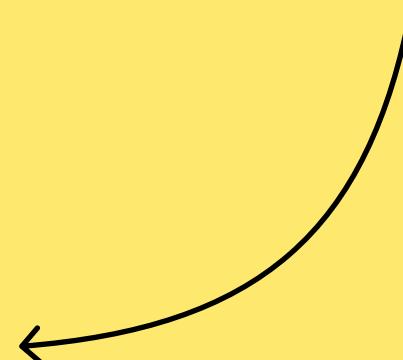
API keys often appear in various locations such as :

- Authorization Header
- Basic Auth
- Body Data
- Custom Header
- Query String

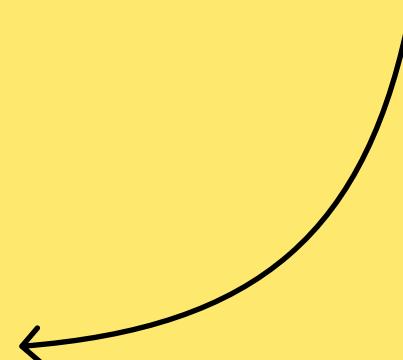
API Keys



API Keys are a useful tool because they are **easy to use**. One identifier is all that is needed, making them a **practical choice** for certain situations. Additionally, API Keys are a **valid option** for a variety of reasons.



For instance, if an API is limited specifically in functionality where “**read**” is the only possible command, an API Key can be an **adequate solution**. Without the need to edit, modify, or delete, security is a lower concern.



The **problem**, however, is that anyone who makes a request to a service, transmits their key and in theory, this key can be picked up just as easy as any network transmission, and if any point in the entire network is insecure, **the entire network is exposed**.

OAuth (2.0)

OAuth 2.0 is an **effective way** to identify individual user accounts and provide the appropriate permissions. When using this method, a user logs into a system, which prompts the request for authentication. This request is typically in the form of a **token**, which the user then sends to an **authentication server**. The server then either denies or grants the **authentication request**.

The token is given to the user and then passed on to the **requester**. The requester can use the token to validate its **authenticity** at any time and can use it within a specific time frame and scope. The token's **age of validity** is also restricted.

This is fundamentally a much more secure and powerful system than the other approaches, mainly because it allows for the **establishment of scopes** which can provide access to different parts of the API service and since the token is revoked after a certain time - makes it much **harder to re-use by attackers**.

OAuth (2.0)

OAuth 2.0 is significantly simpler compared to its predecessors, **OAuth 1.0** and **1.0a**. One of the main improvements in OAuth 2.0 is the **removal of the requirement to sign every call** with a keyed hash. This simplifies the process significantly compared to the previous versions.

One or both of the following **tokens** are often used in the implementation of **OAuth**:

Refresh Token

Access Token

An access token is a type of key that is sent to an application through an API. This key **allows the application to access a user's data**. In some cases, access tokens can have an **expiration date**.

If an access token has expired, refresh tokens **can be used to retrieve a new one** as part of an OAuth flow. OAuth2 combines both authentication and authorization, allowing for more precise control over the **scope and validity** of the token. This allows for more sophisticated security measures to be implemented in the **authorization process**.

OAuth 2.0 Popular Flows

The flows (also called **grant types**) are scenarios an API client performs to get an access token from the authorization server. OAuth 2.0 provides **several popular flows** suitable for different types of API clients.

Some of the widely used flows are:

- Authorization Code
- Implicit
- Resource Owner Password
- Client Credentials

Authorization Code

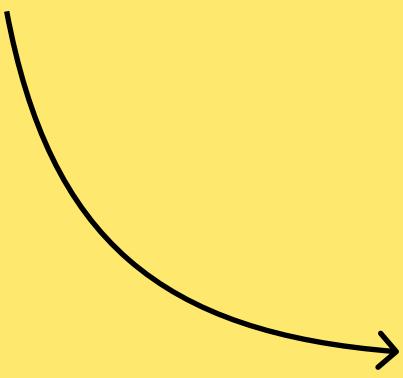
The **most common flow**, mostly used for **server-side** and **mobile web applications**. This flow is similar to how users sign up into a web application using their Facebook or Google account.

Implicit

The client must obtain an **access token on their own** in this flow. It is useful in situations where the user's credentials cannot be stored in the client code as they may be **accessible to external parties**. This prevents the risk of the user's credentials being compromised.

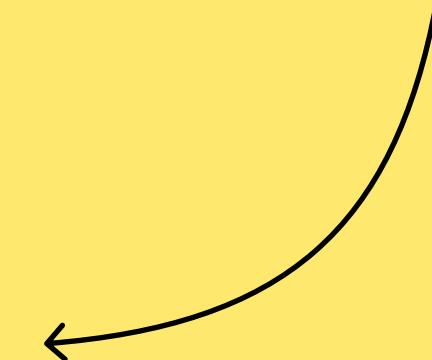
It is suitable for **web**, **desktop**, and **mobile** applications that do not include any server component.

Resource Owner Password



To access this API, a user must provide their **login credentials (username and password)** as part of the request. This method is only suitable for **trusted clients**, such as official applications released by the API provider.

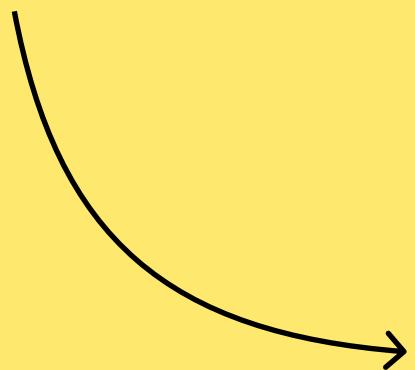
Client Credentials



This flow is designed for **server-to-server authentication** and involves the client application acting on its **own behalf**, rather than representing an individual user. It is used for authenticating communication between servers.

This flow typically enables users to enter their login information in the client app, allowing it to **access resources that are under the user's control**.

OpenID Connect



OpenID Connect is a **protocol** that allows clients to verify the identity of an end-user by **utilizing OAuth 2.0 as a base** and adding an **additional identity layer**. This process is completed through the use of an **authorization server**, which performs the necessary authentication.

It is also used to obtain **basic profile information** about the end-user in an **interoperable** and **REST-like** manner.

OpenID Connect **allows a range of clients**, including Web-based, mobile, and JavaScript clients, **to request and receive information** about authenticated sessions and end-users.

In technical terms, OpenID Connect specifies a **RESTful HTTP API**, using **JSON** as a data format.

OpenID Connect

The specification suite is **extensible**, supporting optional features such as **encryption** of identity data, the **discovery** of OpenID Providers, and **session management**.

OpenID Connect defines a **sign-in flow** that enables a client application to **authenticate a user**, and to obtain information (or “claims”) about that user, such as the **user name, email, and so on.**

User identity information is encoded in a secure **JSON Web Token (JWT)**, called ID token.

JWT

JSON Web Tokens are an open, industry-standard [RFC 7519](#) method for representing claims securely between two parties.

JWT allows you to [decode](#), [verify](#) and [generate](#) JWT. While JWT is a standard it was developed by [Auth0](#), an API driven identity, and authentication management company.

OpenID Connect defines a discovery mechanism, called [OpenID Connect Discovery](#), where an OpenID server publishes its metadata at a [well-known URL](#).

<https://server.com/.well-known/openid-configuration>

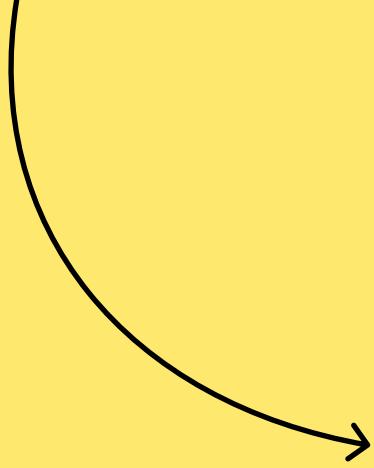
JWT

This URL returns a **JSON listing** of the OpenID/OAuth endpoints, supported scopes and claims, public keys used to sign the tokens, and other details.

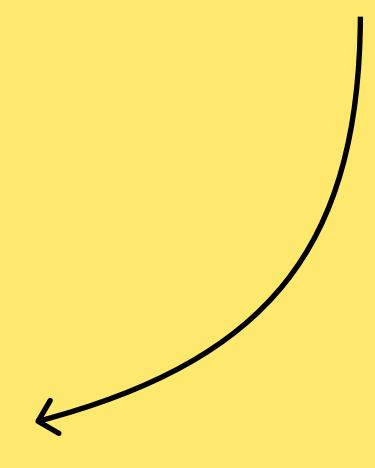
The clients can use this information to **construct a request to the OpenID server**. The field names and values are defined in the **OpenID Connect Discovery Specification**.

These are the 4 most used Authentication methods. Hope you understood their working mechanism and use cases.

Summary



OAuth 2.0 is the best choice for most situations because it is easy to use and provides strong security. It also allows for scalability, meaning it can be used by many different providers at once.



Additionally, OpenID Connect, which is based on OAuth 2.0, is becoming more popular. While API keys and HTTP Authentication may be appropriate in some cases, OAuth 2.0 offers more benefits and is a good long-term investment due to its built-in authorization capabilities.

**For More Interesting
Content**



Brij Kishore Pandey



**Follow Me On
LinkedIn**

<https://www.linkedin.com/in/brijpandeyji/>