As web developers and security professionals grapple with the challenges of authentication and data integrity, one technology stands out for its efficiency and simplicity: The JSON Web Token (JWT). This compact, URL-safe method of representing claims between two parties has revolutionized the way we implement [authentication](https://www.descope.com/learn/post/authentication) and [authorization](https://www.descope.com/learn/post/authorization) processes in web applications. Whether you're building a complex enterprise-grade application or a simple website, understanding JWT can significantly enhance your project's security and user experience.

In this post, you will learn the basics of how JWTs work and the advantages they bring to modern-day web development and authentication.

What is a JWT?

JWT (short for JSON Web Token and pronounced “jot”) is an [open standard](https://www.descope.com/learn/post/authentication-protocols) used to create compact, self-contained tokens used for securely transmitting information between different applications or services. These tokens are typically used for authentication and authorization, as they can contain information that [verifies the identity of a user](https://www.descope.com/learn/post/id-token), and their [permissions](https://www.descope.com/learn/post/access-token).

In terms of authentication, the information stored in the JWT is used to help servers establish trust between an unknown client and themselves.

Structure of a JWT

The three main components of a JWT are the:

1. Header
2. Payload
3. Signature

With these three components, JWTs allow developers to build a stateless authentication or authorization flow that is easily scalable and eliminates the need for servers to maintain session information.

All three of these parts are Base64Url encoded strings concatenated with periods ('.').

**The header** is a JSON object that typically contains two properties: The type of token (JWT) and the encryption algorithm used (e.g., HMAC SHA256, RSA, etc).

Example of JSON header:

Open menu

{

"alg": "HS256",

"typ": "JWT"

}

**The payload**is another JSON object where all of the transmitted data lives. Also called a *claim*, this data typically contains user information (username, email address), session data (IP address, time or last login), or authorization permissions (roles or groups the user belongs to).

There are four types of claims:

* **Commonly used:**Registered and public
* **Not commonly used:**Private and custom

Example of JWT payload:

Open menu

{

"drn": "DS",

"exp": 1680902696,

"rexp": "2023-05-05T21:14:56Z"

}

**The signature**is created by signing the Base64Url encoded header and payload with a secret key and an algorithm specified by the developers. It is used to verify that the sender of the JWT is who they claim to be and ensure the token's integrity.

**Here’s**[**how Descope validates JWTs**](https://docs.descope.com/knowledgebase/general/offlinejwtvalidation/)**.**

How does a JWT work?

JWTs work by encoding a set of claims into a compact, URL-safe string. This string can be easily transmitted over the network and verified by the receiver.

Here is a general overview of how JWTs work:

1. The issuer of the JWT creates a new JWT object and sets the claims that it wants to include in the token.
2. The issuer signs the JWT object using a secret key or a public/private key pair.
3. The resulting JWT is a compact, URL-safe string that can be transmitted over the network.
4. The receiver of the JWT verifies the signature of the JWT using the secret key or the public key.
5. If the signature is valid, the receiver can trust the claims in the JWT.

JWT authentication example

Here is how JWT can be used in an authentication flow:

1. A user provides their credentials (e.g., username and password) and sends them to the server.
2. The server validates the credentials. If they are correct, the server generates a JWT containing the user's information (in a claim) and signs it with a secret key.
3. The server sends the JWT back to the user.
4. The user stores the JWT (usually in the browser's local storage or as a cookie) and includes it in the Authorization header in subsequent requests to the server. *If you’re using Descope, you can change how the JWT is stored*[*here*](https://app.descope.com/settings/project)*, under Token Response Method.*
5. When the user sends a new request with the JWT, the server decodes the JWT, and verifies its signature. If the token is valid, the server processes the request and returns the appropriate response.

Pretty simple right?

**Read more:**[What Is Token-Based Authentication & How It Works](https://www.descope.com/learn/post/token-based-authentication)

Advantages of using JWTs

Along with their stateless design and scalability, there are other reasons why you should consider using JWTs in your projects:

* **Cross-domain support:** Unlike cookies, JWTs can be used across different domains and subdomains, making them ideal for [Single Sign-On](https://www.descope.com/learn/post/sso) (SSO) implementations.
* **Self-containment and extensibility:**Since JWTs already contain necessary information about the user, they reduce the need for extra queries to a database for user data. Moreover, JWTs can be extended with custom claims to include additional information as needed, allowing for greater flexibility.
* **Mobile-friendly:**JWT tokens are an excellent choice for mobile app authentication due to their compact size and stateless nature. They allow for seamless integration with APIs and can greatly reduce server overhead.
* **Enhanced security:**JWTs can be encrypted to protect sensitive data, ensuring that only intended recipients can read the token's content. Moreover, the use of digital signatures ensures that the token has not been tampered with during transmission.

Limitations and considerations of JWTs

It is important to choose the right token format for your application. While JWTs are a good choice for applications that need a compact and easy-to-use token format, it’s best to avoid using them:

* **When the payload contains sensitive information.**JWTs are not encrypted, and the payload can be read by anyone who gains access to it.
* **When the application has strict size limits on network requests.**JWTs can become large if they contain a lot of claims.
* **When the application is vulnerable to replay attacks.** JWTs can be vulnerable to replay attacks if they do not have a way to prevent them.
* **When the application is vulnerable to**[**man-in-the-middle attacks**](https://www.descope.com/learn/post/mitm-attack)**.**JWTs can be vulnerable to MITM attacks if they are not signed using a strong algorithm.

Best practices for using JWTs

To ensure the security and effectiveness of JWTs in your application, follow these best practices:

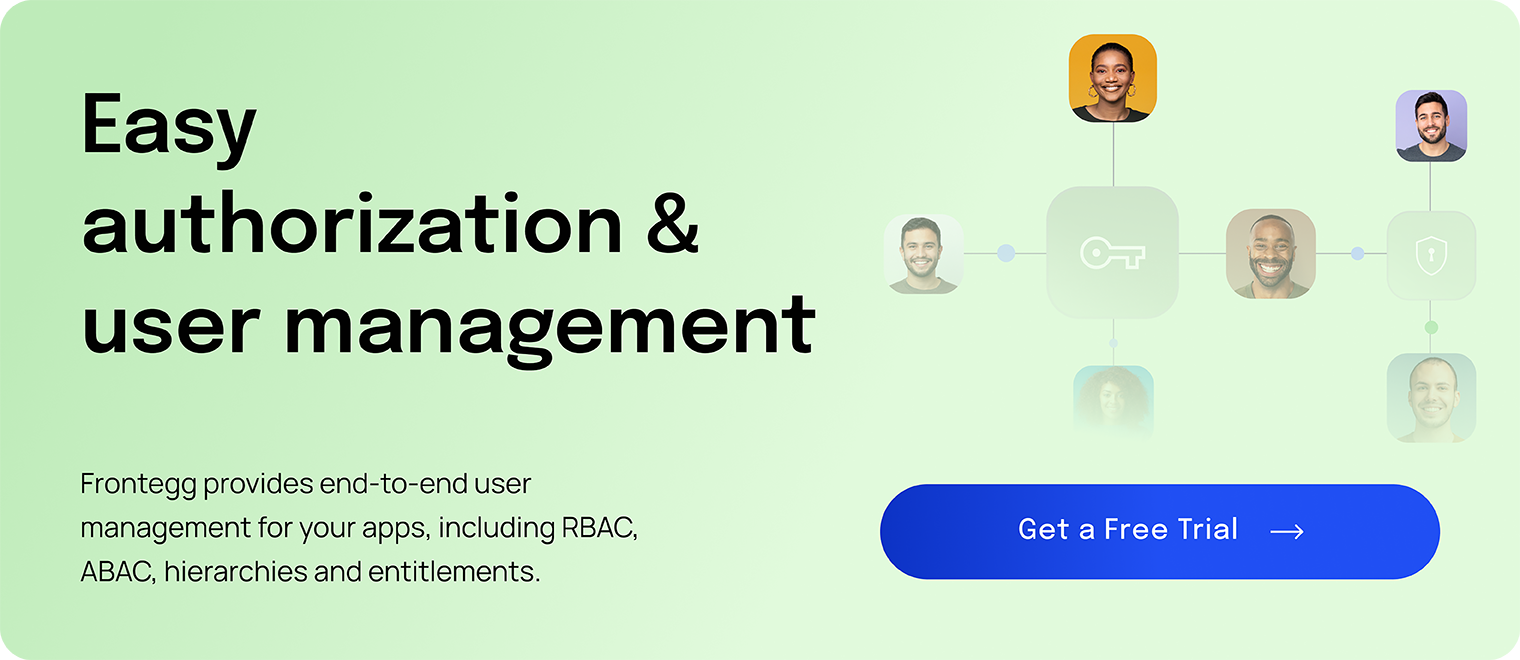
* **Secure the secret key:**Make sure that you keep the secret key used to sign the JWT confidential to prevent unauthorized access. For added security, it’s best to use a key management system or store the key in a secure environment variable.
* **Use HTTPS:**To protect JWTs from being intercepted during transmission, always use HTTPS for communication between the client and server.
* **Use appropriate algorithms:**To protect your web application, make sure that you choose a suitable signing algorithm for your JWTs. [Asymmetric algorithms](https://cryptography.io/en/latest/hazmat/primitives/asymmetric/) (like RSA or ECDSA) are generally considered the best, as they use a public/private key pair, making it difficult for an attacker to forge tokens.
* **Handle token revocation:**You should always assign a short expiration time for JWTs to minimize the risk of token theft or misuse. Many libraries will implement a mechanism for token revocation to address situations where a user's access must be immediately revoked, such as account deletion or security breaches.

JSON Web Token (JWT) is a commonly used user authentication and [authorization](https://frontegg.com/guides/authorization-a-complete-guide) standard, used to exchange data in a secure manner. Made up of three components, a header, a payload, and a signature, it’s becoming more and more commonly used. Read on to discover the best use cases for JWT authorization, learn how it works, and access best practices that can help you implement it effectively in your organization.

What Is JWT Authorization?

JWT stands for JSON Web Token. It is a compact, URL-safe means of representing claims to be transferred between two parties. It is used to securely transmit information between parties in a JSON format. JWTs can be used to authorize access to resources and services.

JWT authorization is a stateless mechanism for authentication and authorization that eliminates the need for sessions and cookies. It provides a secure means of transmitting information, because a JWT is digitally signed using a secret key known only to the server. This ensures that the information contained in the JWT is not tampered with or altered during transmission.

[](https://portal.frontegg.com/signup?optin_monster_type=inline&optin_monster_keyword=authorization)

*For more background, see our article on*[*JWT authentication*](https://frontegg.com/blog/jwt-authentication).

In this article:

* [What Is JWT Authorization?](https://frontegg.com/guides/jwt-authorization#What_Is_JWT_Authorization)
* [How Does JWT Authorization Work?](https://frontegg.com/guides/jwt-authorization#How_Does_JWT_Authorization_Work)
* [API Keys vs. JWT Authorization](https://frontegg.com/guides/jwt-authorization#API_Keys_vs_JWT_Authorization)
* [General Steps for Implementing JWT Authorization in Your Application](https://frontegg.com/guides/jwt-authorization#General_Steps_for_Implementing_JWT_Authorization_in_Your_Application)
* [Authentication and Authorization with Frontegg](https://frontegg.com/guides/jwt-authorization#Authentication_and_Authorization_with_Frontegg)

How Does JWT Authorization Work?

JWT authorization works by encoding information into a JSON web token (JWT), which is then passed between the client and server. The steps involved in a typical JWT authorization flow are as follows:

* **Authentication**: The client sends the user’s credentials to the server, which authenticates the user and generates a JWT containing information about the user.
* **Issuing the Token**: The server sends the JWT back to the client, which stores it for future use.
* **Sending the Token**: When the client wants to access a protected resource on the server, it sends the JWT in the Authorization header of the HTTP request.
* **Verifying the Token**: The server receives the request and verifies the JWT by checking its signature using the secret key that was used to sign it. If the JWT is valid, the server extracts the information contained in it and uses it to determine what actions the user is authorized to perform.
* **Authorizing the Request**: If the user is authorized to access the resource, the server returns the requested data. If the user is not authorized, the server returns an error message.

JWT authorization enables secure and efficient communication between the client and server, as the server does not need to store any session information to keep track of the user’s authentication status. This makes it ideal for use in microservice architectures and other decentralized systems, where multiple independent components need to communicate with each other in a secure manner.

Related content: Read our guide to [asp net authorization](https://frontegg.com/guides/asp-net-authorization)

API Keys vs. JWT Authorization

API keys and JWT authorization are two different mechanisms for authenticating and authorizing access to an API.

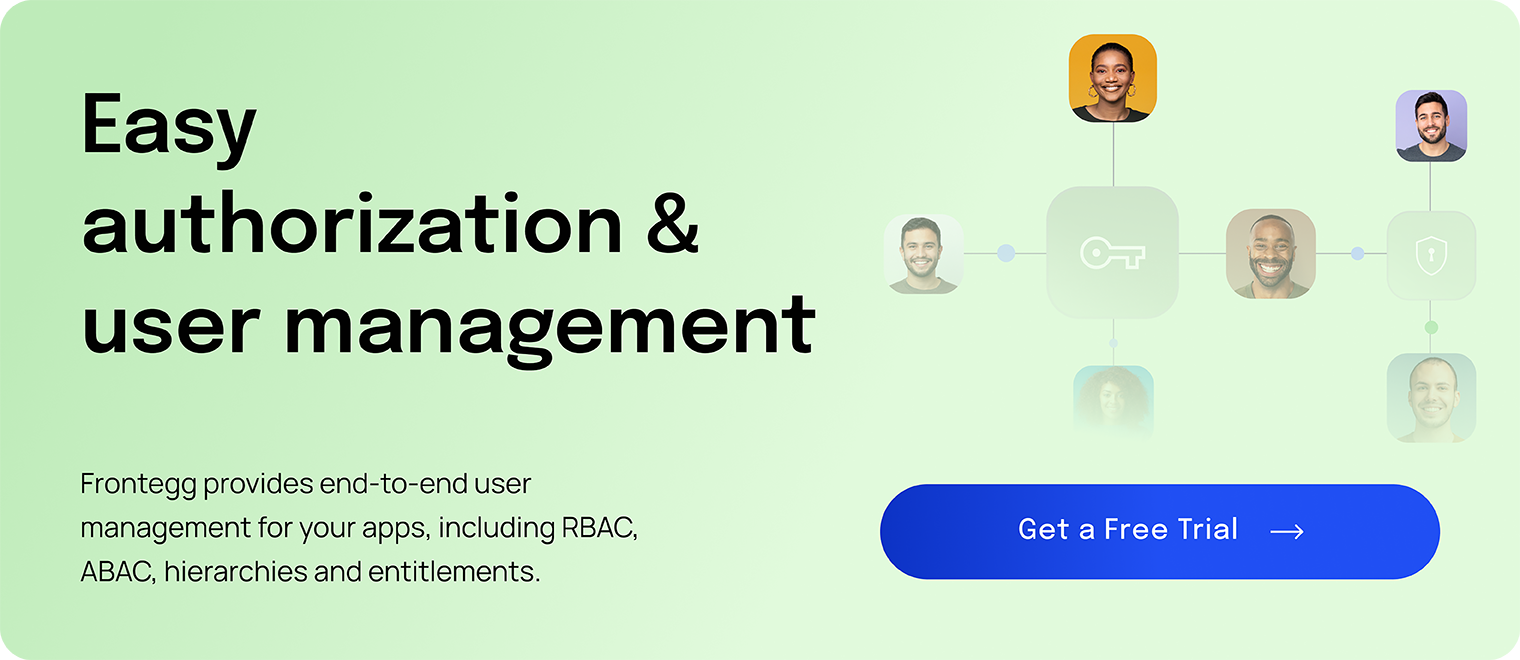
What are API keys?

API keys usually consist of a long string of characters, which are sent along with the API request as a parameter or in headers. An API key is typically generated by an API provider and is shared with a client, who needs to include it with every API request. API keys can be used to identify the client and limit the usage of the API.

What are the differences?

JWT authorization uses a JWT to represent the user’s identity and access rights. The JWT is usually generated by the authentication server after the user logs in and contains the user’s identity and access rights. The JWT is then sent with every API request as a bearer token in the authorization header.

Here is a comparison table between API keys and JWT authorization:

[](https://portal.frontegg.com/signup?optin_monster_type=inline&optin_monster_keyword=authorization)

|  |  |  |
| --- | --- | --- |
| **Feature** | **API Keys** | **JWT Authorization** |
| **Purpose** | Identifies the client, limits API usage. | Authenticates and authorizes the user. |
| **Format** | Long string of characters. | Encoded JSON object. |
| **Security** | Less secure, can be easily stolen. | More secure, digitally signed and encrypted. |
| **Usage** | Sent as a parameter or header with each request. | Sent as a bearer token in the authorization header. |
| **Authentication** | Not used for authentication. | Used for authentication. |
| **Authorization** | Not used for authorization. | Used for authorization. |
| **Flexibility** | Limited flexibility. | More flexible, supports complex access control. |
| **Ease of Use** | Simple to use. | More complex, requires token generation and verification. |
| **Standardization** | Not standardized, varies by API provider. | Standardized, based on JWT standard. |

In summary, while API Keys are simpler to use, they are less secure and less flexible than JWT authorization. JWT Authorization provides a more secure and flexible mechanism for authenticating and authorizing access to an API.

General Steps for Implementing JWT Authorization in Your Application

Here are the main steps you will need to implement JWT authorization:

1. **Set up a server-side application:** You’ll need a backend application that will generate and verify JWTs. You can use any server-side language and framework, such as Node.js and Express.
2. **Install the necessary packages:**You’ll need to install a JWT library for your server-side language. For example, if you’re using Node.js, you can install the jsonwebtoken library.
3. **Implement authentication:** Your server-side application will need to implement authentication to verify the user’s credentials. You can use methods like email/password authentication or social media authentication.
4. **Generate the JWT:** Once the user has been authenticated, your server-side application will generate a JWT that contains information about the user, such as the user’s ID, name, and roles. You can sign the JWT using a secret key or a public/private key pair.
5. **Send the JWT to the client:** The server will send the JWT to the client, which will store it for future use.
6. **Send the JWT with every request:** When the client wants to access a protected resource on the server, it will send the JWT in the Authorization header of the HTTP request.
7. **Verify the JWT on the server:** The server will receive the request and verify the JWT by checking its signature using the secret key that was used to sign it. If the JWT is valid, the server will extract the information contained in it and use it to determine what actions the user is authorized to perform.
8. **Authorize the request:** If the user is authorized to access the resource, the server will return the requested data. If the user is not authorized, the server will return an error message.

With these steps, you can implement JWT authorization in your application and secure the communication between the client and server.

Authentication and Authorization with Frontegg

The industry standard today is to use Authentication providers to “build the door”, but what about Authorization (the door knob)? Most authentication vendors don’t go the extra mile, forcing SaaS vendors to invest in expensive in-house development. This often delays core technology development and impacts developer productivity, something that negatively impacts innovation and time-to-market (TTM) metrics.

Frontegg’s end-to-end user management platform allows you to authenticate and authorize users with just a few clicks. Integration takes just a few minutes, thanks to its plug-and-play nature. It’s also multi-tenant by design.