

# API Keys vs OAuth Tokens vs JSON Web Tokens



Adam DuVander / March 2, 2017

For an API to be a powerful extension of a product, it almost certainly needs authentication. By building API calls that can read, write, and delete user data, you can magnify an app's influence on its users' lives. So, if authentication is a given, the method is the real choice. The industry has finally learned not to share usernames and passwords, but there's still more to figure out.

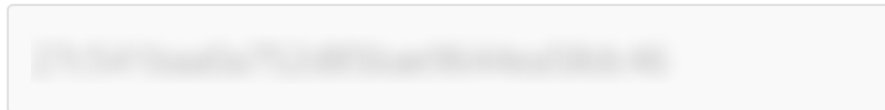
Below we'll look at three popular authentication methods: [API keys](#), [OAuth access tokens](#), and [JSON Web Tokens](#) (JWT). We'll cover how each is used and why you might choose one over the others.

## API Keys: Great for Developer Quickstart

In the earliest days of modern web APIs, the API key was all we had. It likely remains as the most common identifier, and is the first many developers consider when restricting or tracking API traffic. The best thing about an API key is its simplicity. You merely log in to a service, find your API key (often in the settings screen), and copy it to use in an application, test in the browser, or use with one of these [API request tools](#). Along with the simplicity, though, comes both security and user experience downsides to API keys.

### API TOKEN

HIDE TOKEN



This is your personal secret key to access the API  
Do not publish or share this private token



### Build Great APIs

Get new articles about API design, documentation, and success delivered to your inbox.

Subscribe

### Integrate With Zapier

Zapier connects hundreds of apps to give you the integrations you need. Easily automate tedious tasks to let Zapier do the work for you.

[Explore Developer Platform](#)

### Automate Your Work

Zapier connects hundreds of apps to give you the integrations you need. Easily automate tedious tasks to let Zapier do the work for you.

[Try it Free](#)

### Recent Posts

[Machine learning made easier with datto package](#)

[How to write great bug bounty submissions](#)

[How Zapier Reduces Churn for Clearbit](#)

[Zapier at PyCon 2019](#)



Log in

Sign up

[Track Your API Users to Gauge Integration Effectiveness](#)[API Best Practices: Webhooks, Deprecation, and Design](#)[How we used iptables to replicate UDP traffic when upgrading our Graylog cluster](#)[How We Automated Our Engineering Skills Test for Hundreds of Applicants](#)

Typically, an API key gives full access to every operation an API can perform, including writing new data or deleting existing data. If you use the same API key in multiple apps, a broken app could destroy your users' data without an easy way to stop just that one app. Some apps let users generate new API keys, or even have multiple API keys with the option to revoke one that may have gone into the wrong hands. The ability to change an API key limits the security downsides.

Many API keys are sent in the query string as part of the URL, which makes it easier to discover for someone who should not have access to it. A better option is to put the API key in the Authorization header. In fact, that's the [proposed standard](#):

```
Authorization: Apikey 1234567890abcdef
```

Yet, in practice API keys show up in all sorts of places:

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

Got others? Send them along and we'll add them to the list.

The user experience of API keys is something to consider, as well. API keys make sense when the users of an API are only developers. However, as developers created tools for themselves, they started sharing them with others. End users often find themselves fumbling through API documentation, registration, and settings just to find the API key that a tool needs—often without even knowing what an API is.

In the same way that Zapier user data showed [the poor user experience of static webhooks](#), moving out of a flow to find API keys distract users from their desired purpose. Combine that with the security concerns and there are other much better approaches to access user data with APIs.





## OAuth Tokens: Great for Accessing User Data

OAuth is the answer to accessing user data with APIs. Unlike with API keys, OAuth does not require a user to go spelunking through a developer portal. In fact, in the best cases, users simply click a button to allow an application to access their accounts. OAuth, specifically OAuth 2.0, is a standard for the

[See All Engineering Posts](#)

process that goes on behind the scenes to ensure secure handling of these permissions.

## Review permissions

	<b>Personal user data</b> Full access	▼
	<b>Repositories</b> Public and private	▼
	<b>Notifications</b> Read access	▼
	<b>Gists</b> Read and write access	▼

**Authorize application**

The previous versions of this spec, OAuth 1.0 and 1.0a, were much more complicated than OAuth 2.0. The biggest change in the latest version is that it's no longer required to sign each call with a keyed hash. The most common implementations of OAuth use one or both of these tokens instead:

- **access token:** sent like an API key, it allows the application to access a user's data; optionally, access tokens can expire.
- **refresh token:** optionally part of an OAuth flow, refresh tokens retrieve a new access token if they have expired.

Similar to API keys, you may find OAuth access tokens all over the place: in query string, headers, and elsewhere. Since an access token is like a special type of API key, the most likely place to put it is the authorization header, like so:

Authorization: Bearer 1234567890abcdef

The access and refresh tokens should not be confused with the Client ID and Client Secret. Those values, which may look like a similar random collection of characters, are used to negotiate access and refresh tokens.

Like an API key, anyone with an access token can potentially invoke harmful operations, such as deleting data. However, OAuth provides several

improvements over API keys. For starters, access tokens can be tied to particular scopes, which restrict the types of operations and data the application can access. Also, combined with refresh tokens, access tokens will expire, so the negative effects could have a limited impact. Finally, even if refresh tokens aren't used, access tokens can still be revoked.

## JWT Tokens: Great for Limiting Database Lookups

Whereas API keys and OAuth tokens are always used to access APIs, [JSON Web Tokens](#) (JWT) can be used in many different scenarios. In fact, JWT can store any type of data, which is where it excels in combination with OAuth. With a JWT access token, far fewer database lookups are needed while still not compromising security.

While a JWT is longer than most access tokens, they're still relatively compact (though this depends on how much data you store within them):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b21lcGFuZSI6Imh0dHBzOi8vemFwaWVyLmNvbS9kZXZlbG9wZXIvIiwidGFnbGluZSI6ImlphcG1lcjBtYWtlcyB5b3UgaGFwcG1lcjIsIm1lc3NhZ2UiOiJHb29kIGpvYiEgW91J3JlIGVWZdGVyIGRlY29kZXIhIPCfkY8ifQ.qti9DKAJhwoTzu511CbVJ0g2cuSGbcIILj0iQ7yXp_E
```

You're able to avoid database lookups because the JWT contains a base64 encoded version of the data you need to determine the identity and scope of access. The JWT also contains a signature calculated using the JWT data. Using the same secret you used to produce the JWT, you calculate your own version of the signature and compare. This calculation is much more efficient than looking up an access token in a database to determine who it belongs to and whether it is valid.

Like OAuth access tokens, JWT tokens should be passed in the Authorization header:

```
Authorization: Bearer  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b21lcGFuZSI6Imh0dHBzOi8vemFwaWVyLmNvbS9kZXZlbG9wZXIvIiwidGFnbGluZSI6ImlphcG1lcjBtYWtlcyB5b3UgaGFwcG1lcjIsIm1lc3NhZ2UiOiJHb29kIGpvYiEgW91J3JlIGVWZdGVyIGRlY29kZXIhIPCfkY8ifQ.qti9DKAJhwoTzu511CbVJ0g2cuSGbcIILj0iQ7yXp_E
```

The downside of not looking up access tokens with each call is that a JWT cannot be revoked. For that reason, you'll want to use JWT in combination with refresh tokens and JWT expiration. With each API call, you would need to check the JWT signature and ensure that the expiration is still in the future.

## Which Should I Use?

As you've seen, these three options aren't mutually exclusive. In fact, it's *possible* an API could use all three at once. Or, each could be used independently of the others.

- **Use API keys** if you expect developers to build internal applications that don't need to access more than a single user's data.
- **Use OAuth access tokens** if you want users to easily provide authorization to applications without needing to share private data or dig through developer documentation.
- **Use JWT** in concert with OAuth if you want to limit database lookups and you don't require the ability to immediately revoke access.

**Note:** One way to keep the simplicity of API keys while also having your API support OAuth is to create one-off tokens for internal use.

No matter which of these authentication schemes you're using, you can make your API available to Zapier users using the [Zapier Developer Platform](#). You'll want to pay special attention to the [authentication documentation](#) which describes how API keys, OAuth, and other authentication schemes can be seamlessly incorporated into your Zapier App.



## About the Author

Adam DuVander is a developer marketer working from Portland, Oregon. He loves APIs and the people who make and use them.

---

[Load Comments...](#)

Pricing  
Help  
Developer Platform  
Press  
Jobs  
Zapier for Companies

Follow us

**zapier**<sup>\*</sup> makes you happier :)

