## **JSON Web Tokens (JWT)**

JWT are signed, and consist of three parts: **the header, payload and signature**, **each base64-encoded and then concatenated**. The result is a long string of base62 characters.

JWT fill different roles; they can be "**Identity tokens**", which are tokens issued by an identity provider to a client application that provide information about an authenticated user (like name, email address, and other claims or attributes).

JWT can also be used as "**access tokens**"; in this usage, the client application presents a JWT to a service as credentials with a service request. The service provider can evaluate whether to provide service based on the claims in the token. The distinction between ID token and access token is primarily one of intent, not of form.

A JWT token is composed of three parts: a header, payload, and signature.

The problem is that people can put sensitive info in the payload.

None of it is encrypted, it's only signed with HMAC.

Unless you're keeping track of the tokens, once a token is issued **it's valid until it expires**, due to it's stateless nature.

You can use a JWT as a Bearer token**, but since it's only base64 encoded, you can pull out that payload data.**

JWT, in contrast, are not **opaque**. JWT actually contains **meta data** that can be extracted and interpreted by any bearer that has the token. JWT usually contains real information so it can be of variable size depending on the claims contained within it and the algorithm used to sign it.

Any holder of the JWT can inspect it, validate it and then optionally make authorization decisions based on the claims presented in it. Validating the JWT means usually involves following:

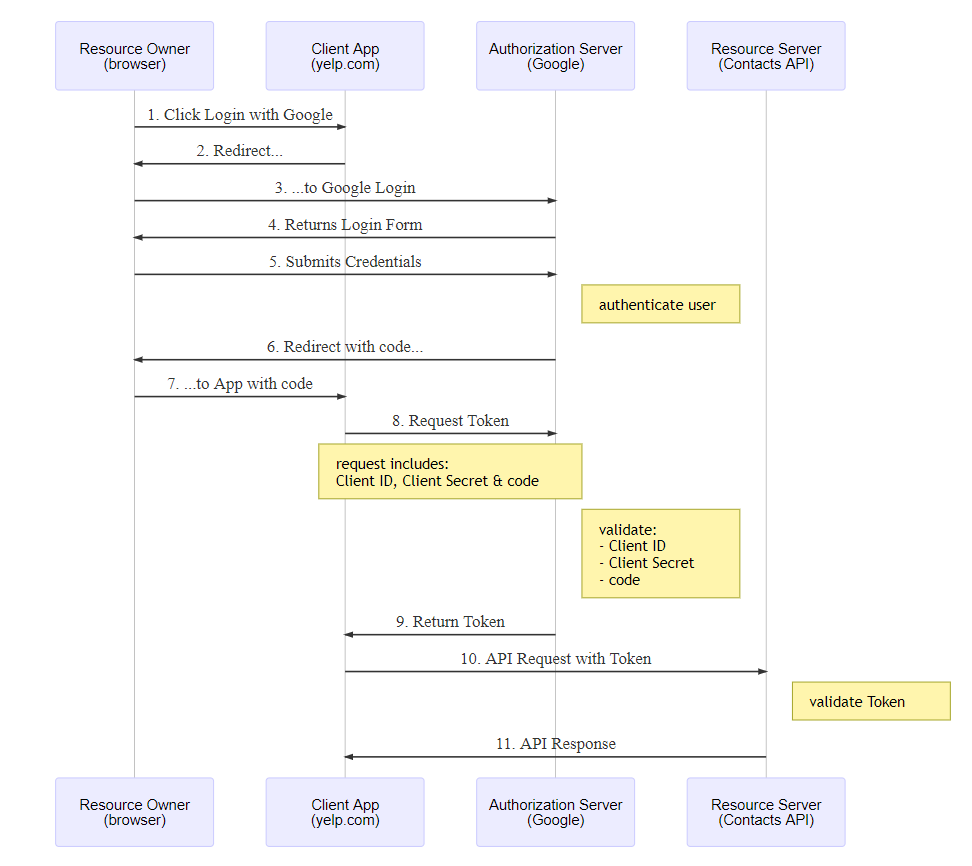
* verifying JWT structure
* decoding the base64 encoding
* verifying the key is correct
* verifying the signature
* verifying the required claims are present in the token
* **verifying the expiry of the JWT**
* Sites like **Yelp** started wanting access to the **contact information** you had in your **Google Contacts**. Yelp naturally collected **your Google username and password** so that it could access your contacts. You gave Yelp your permission by giving yelp your google username/passwd.
* With your google username and password, Yelp could access your email, your docs - everything you had in Google - not just your contacts.
* And, worse, Yelp had to store your password in a way that it could use it in plaintext and there was no standard way to revoke your consent to Yelp to access your Google account.

The world needed an authorization framework that would allow you to grant access to specific information **without you sharing your password**. Cue OAuth.

* Three revisions later, we’re at OAuth 2.0 (there was 1.0 and 1.0a before it) and all’s right with the world.
* Now, an application like Yelp (a **Client Application**) can request an **Access Token** from a service like Google (an **Authorization Server**). **You** (the **Resource Owner**) **log into Google with your credentials and give your Consent to Yelp** to access your contacts (and only your contacts).
* **Access Token** in hand, Yelp makes a request of the Google Contacts API (the **Resource Server**) and gets your contacts. **Yelp never sees your password** and never has access to anything more than you’ve consented to. And, you can withdraw your consent at any time.
* So Auth 2.0 solves the problem with resource owner has to share credentials (username/passwd) with client application; instead provide access token or grant to client application.
* In this new world of consent and authorization, only one thing was missing: **identity**. Cue **OpenID Connect**. OIDC is a thin layer on top of OAuth 2.0 that introduces a new type of token: the **Identity Token**. Encoded within these cryptographically signed tokens in [JWT](https://developer.okta.com/docs/api/resources/oidc#access-token) format, is information about the authenticated user. This opened the door to a new level of interoperability and **single sign-on**.

OAuth (and by extension OIDC) uses a number of defined Flows to manage the interactions between the Client App, the Authorization Server and the Resource Server. **The most secure** of these is the **Authorization Code Flow**. This flow is meant to be kicked off from your browser and goes like this:

1. Yelp wants access to your contacts. It presents a button to link your Google Contacts.
2. When you click the button, you’re **redirected to Google where you login** with your username and password (if you’re not already logged in).
3. Google shows you a screen telling you that Yelp would like read-only access to your contacts.
4. Once you give your consent, Google redirects back to Yelp, via your browser, with a temporary code (called an authorization code)
5. **Using this code along with a secret**, Yelp contacts Google to trade it for an Access Token
6. Google validates the code and if all checks out, **issues an Access Token** with limited capabilities (read-only access to your contacts) to Yelp
7. **Yelp** then presents the **Access Token** to the **Google Contacts API**
8. Google Contacts API validates the token and, if the request matches the capabilities identified by the token, returns your contact list to Yelp



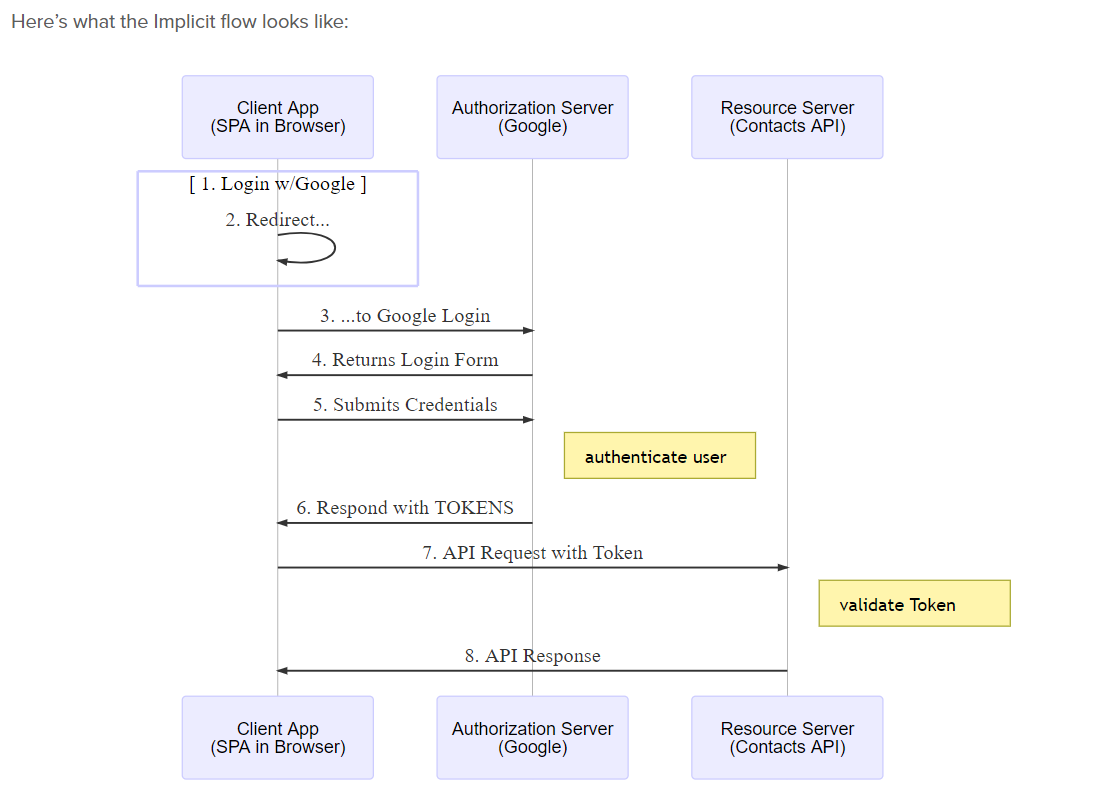
Notice step 8 in the diagram. In addition to the code, Yelp must present **a secret** **that has been assigned by Google**, **which is how Google validates Yelp as a client.**

**This flow is great for web apps**, but it’s not safe to store a secret in a SPA app, since anyone can view source code in the browser and gain access to that secret. In the early days of OAuth 2.0, without better options, the Implicit flow provided a mechanism to get ID and Access tokens from the Authorization server. PKCE represents a better option now, but let’s first visit the Implicit flow to see why it’s less secure.

[**Why You Should Never Use the Implicit Flow Again**](https://developer.okta.com/blog/2019/08/22/okta-authjs-pkce#why-you-should-never-use-the-implicit-flow-again)

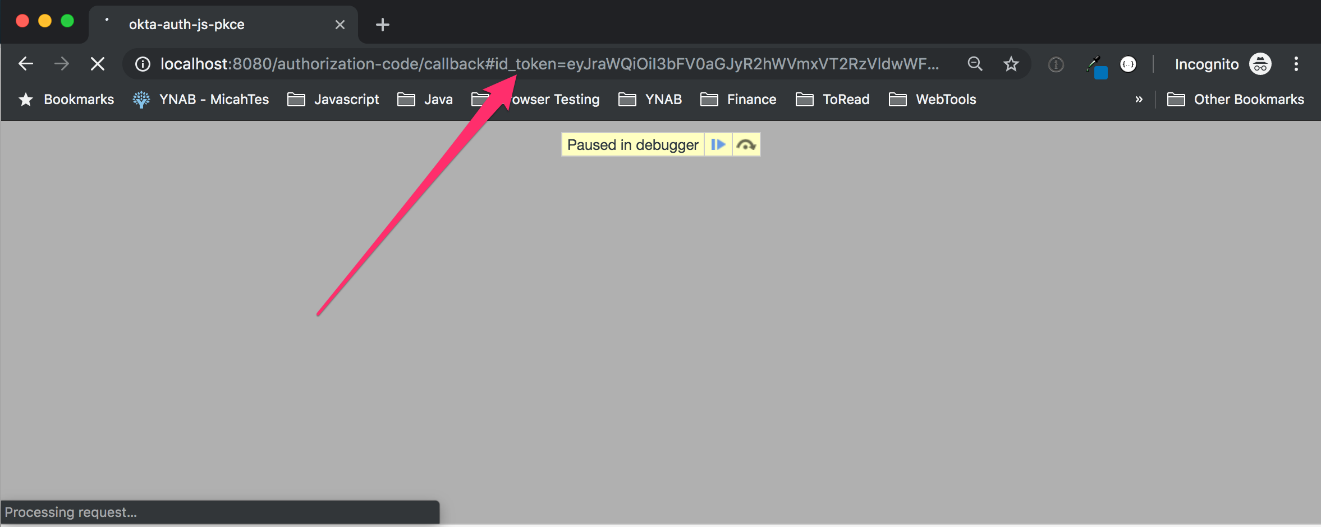
The OAuth 2.0 specification included the Implicit Flow at a time when browser support for SPAs was much more limited. In particular, JavaScript did not have access to browser history or local storage. Also, most providers did not allow cross-site POST requests to a /token endpoint, which is a requirement of the Authorization Code flow.

Here’s what the Implicit flow looks like:



Notice that after you authenticate**, the Authorization Server (like Google) responds directly with tokens**. This means that the **tokens are in your browser’s address bar as a result of the redirect**. That’s problematic since Google can’t definitively know that your browser (the intended recipient) actually received the response. It’s also problematic because modern browsers can do browser history syncing and they support browser extensions that could be actively scanning for tokens in the browser address bar. Leaking tokens is a big security risk.

In the screenshot below, you can see that the execution is paused to capture the id\_token in the browser address bar:



These security issues led to a reassessment of the value of the Implicit flow, and in November of 2018, new guidance was released that effectively deprecated this flow. Additional specs that speak to [updated guidelines for security with OAuth 2.0](https://oauth.net/2/oauth-best-practice/) in general and [security for web apps](https://oauth.net/2/browser-based-apps/) in particular were put forward this year as well.

If you can’t (or shouldn’t) use the Implicit flow, then what? It turns out there’s an extension to the Authorization Code flow that’s been in use for some time with Mobile and Native apps. That’s Proof Key for Code Exchange or PKCE (pronounced “pixie”).

**The OAuth 2.0 Authorization Framework**

Access tokens CANNOT be invalidated. Once issued they are forever valid until the expiration is reached or the token is tampered with. Revoking refresh tokens does not influence access tokens.

Access token are ***bearer* tokens**. Meaning **no other identification is required, and the access token is all that is needed to impersonate you**. Because of this, they should always remain short lived. On the other hand **refresh tokens are not *bearer* tokens**. When you send a refresh token to YouTube to get a new access token, you also have to send a **client\_id** and **client\_secret**. Because of this, refresh token can remain longer lived because it is much less likely that both the refresh token and the client\_secret would be compromised.

The idea of refresh tokens is that if an access token is compromised, because it is short-lived, the attacker has a limited window in which to abuse it.

**Refresh tokens**, if compromised, are useless because the attacker requires the client id and secret in addition to the refresh token in order to gain an access token.

**Having said that**, because every call to both the authorization server and the resource server is done over **SSL** - including the original **client id and secret** when they request the access/refresh tokens - **I am unsure as to how the access token is any more "compromisable" than the long-lived refresh token and clientid/secret combination.**

This of course is different to implementations where you don't control both the authorization and resource servers.

Refresh tokens... mitigates the risk of a long-lived access\_token leaking (query param in a log file on an insecure resource server, beta or poorly coded resource server app, JS SDK client on a non https site that puts the access\_token in a cookie, etc)

the **client ID** and **secret** are credentials for the **OAuth client**, not the user. When talking about **OAuth the "client" is usually a server** (for example the stackoverflow web server) which interfaces with an **authorization** or **resource API server** (for example the facebook auth provider). The **user's credentials** are only passed between the user and the OAuth API server, and never known to the client. The **client secret** is only passed from the client to the OAuth API server, and is never known to the user.

**How the system with long-lived access tokens should work**

The server allows the Client to get access to User's data within a pre-defined set of scopes by issuing a token. As we want to keep the token revocable, we must store in the database the token along with the flag "revoked" being set or unset (otherwise, how would you do that with self-contained token?) Database can contain as much as len(users) x len(registered clients) x len(scopes combination) records. Every API request then must hit the database. Although it's quite trivial to make queries to such database performing O(1), the single point of failure itself can have negative impact on the scalability and performance of the system.

**How the system with long-lived refresh token and short-lived access token should work**

Here we issue two keys: random refresh token with the corresponding record in the database, and signed self-contained access token, containing among others the expiration timestamp field.

**As the access token is self-contained, we don't have to hit the database at all to check its validity. All we have to do is to decode the token and to validate the signature and the timestamp.**

Nonetheless, we still have to keep the **database of refresh tokens**, but the number of requests to this database is generally defined by the lifespan of the access token (the longer the lifespan, the lower the access rate).

In order to revoke the access of Client from a particular User, **we should mark the corresponding refresh token as "revoked"** (or remove it completely) and **stop issuing new access tokens**. It's obvious though that there is a window during which the refresh token has been revoked, but its access token may still be valid.

**Tradeoffs**

Refresh tokens partially eliminate the SPoF (Single Point of Failure) of Access Token database, yet they have some obvious drawbacks.

1. The "window". A timeframe between events "user revokes the access" and "access is guaranteed to be revoked".
2. The complication of the Client logic.

**without** refresh token

* + send API request with access token
  + if access token is invalid, fail and ask user to **re-authenticate**

**with** refresh token

* + send API request with access token
  + If access token is invalid, try to update it using refresh token
  + if refresh request passes, update the access token and re-send the initial API request
  + If **refresh request fails**, ask user to **re-authenticate**

I hope this answer does make sense and helps somebody to make more thoughtful decision. I'd like to note also that some well-known OAuth2 providers, including github and foursquare adopt protocol without refresh tokens, and seem happy with that.

 the short version of my post is, if you save the access token in the database, you hit the database on every request to your API (which may or may not be a problem in your particular case). If you save refresh tokens and keep access tokens "self-contained", you hit the database only when the client decides to refresh the access token.

To clear up some confusion you have to understand the roles of the [*client secret*](https://salesforce.stackexchange.com/questions/14009/whats-the-benefit-of-the-client-secret-in-oauth2) and the *user password*, which are very different.

**The *client*** is an app/website/program/..., backed by a server, that wants **to *authenticate* a *user*** by using a third-party authentication service. The **client secret** is a (random) string that is known to **both this client and the authentication server**. Using this secret the client can identify itself with the authentication server, receiving *authorization* to request access tokens.

**To get the initial access token and refresh token**, what is required is:

* **The user ID**
* **The user password**
* **The client ID**
* **The client secret**

To get a **refreshed access token** however the *client* uses the following information:

* **The client ID**
* **The client secret**
* **The refresh token**

This clearly shows the difference: when refreshing, the client receives authorization to refresh access tokens by using its client secret, and can thus re-authenticate the user using the refresh token *instead* of the user ID + password. **This effectively prevents the user from having to re-enter his/her password.**

This also shows that losing a refresh token is no problem because the client ID and secret are not known. It also shows that keeping the client ID and client secret secret is *vital*.

A server would verify an access token based on credentials and signing of (typically) a JWT.

An access token leaking is bad, but once it expires it is no longer useful to an attacker. A refresh token leaking is far worse, but presumably it is less likely. (I think there is room to question whether the likelihood of a refresh token leaking is much lower than that of an access token leaking, but that’s the idea.)

Point is that the access token is added to every request you make, whereas a refresh token is only used during the refresh flow So less chance of a MITM seeing the token

In addition, if the authorization server is separate from the application server processing other client requests then that application server will never see refresh tokens. It will only see access tokens that will not live for much longer.

**The most basic form of API authentication is typically known as**[**HTTP Basic Authentication**](https://en.wikipedia.org/wiki/Basic_access_authentication).

**API key is Basic Authentication**

The way it works is pretty simple for both the people writing API services, and the developers that consume them:

* A developer is given an **API key (typically an ID and Secret).** This API key usually looks something like this: **3bb743bbd45d4eb8ae31e16b9f83c9ba**:**ffb7d6369eb84580ad2e52ca3fc06c9d**.
* A developer is responsible for storing this API key in a secure place on their server, in a place that nobody else can access it.
* The developer makes API requests to the API service by putting the API key he was given into the **HTTP Authorization header** along with the word **Basic** (which is used by the API server to properly decode the authorization credentials). Here’s how a developer might specify his API key when authenticating to an API service via the command line tool cURL:

|  |  |
| --- | --- |
| 1  2 | $ curl --user **3bb743bbd45d4eb8ae31e16b9f83c9ba**:ffb7d6369eb84580ad2e52ca3fc06c9d https://api.example.com/v1/test |

The cURL tool will take the API credentials**,**[**base64**](https://en.wikipedia.org/wiki/Base64)**encode** them, and create an HTTP Authorization header that looks like this: Basic  
M2JiNzQzYmJkNDVkNGViOGFlMzFlMTZiOWY4M2M5YmE6ZmZiN2Q2MzY5ZWI4NDU4MGFkMmU1MmNhM2ZjMDZjOWQ=.

The API server will then reverse this process. When it finds the HTTP Authorization header, it will base64 decode the result, grab the API key ID and Secret, then validate these tokens before allowing the request to continue being processed.

HTTP Basic Authentication is great because it’s simple. A developer can request an API key and easily authenticate to the API service using this key.

What makes HTTP Basic Authentication **a bad option** for mobile apps is that **you need to actually store the API key securely** in order for things to work. In addition to this, HTTP Basic Authentication requires that **your raw API keys be sent over the wire for every request**, thereby increasing the chance of exploitation in the long run (the less you use your credentials, the better).

Introducing OAuth2 for Mobile API Security

You’ve probably heard of [OAuth](https://oauth.net/) before, and the debate about what it is and is not good for. Let’s be clear: OAuth2 is an excellent protocol for securing API services from untrusted devices, and it provides a nice way to authenticate mobile users via what is called ***token authentication***.

Here’s how OAuth2 token authentication works from a user perspective (*OAuth2 calls this the* ***password grant*** *flow*):

1. A user opens up your mobile app and is prompted for their *username or email* and *password*.
2. You send a POST request from your mobile app to your API service with the user’s *username or email* and *password* data included (*OVER SSL!*).
3. You **validate the user credentials**, and **create an access token for the user** that expires after a certain amount of time.
4. You store this access token on the mobile device, treating it like an API key which lets you access your API service.
5. Once the access token expires and no longer works, **you re-prompt the user for their *username or email* and *password*U**.

What makes OAuth2 great for securing APIs is that it doesn’t require you to store API keys in an unsafe environment. Instead, it will generate *access tokens* that can be stored in an untrusted environment *temporarily*.

This is great because even **if an attacker somehow manages to get a hold of your temporary access token, it will expire**! This reduces damage potential (*we’ll cover this in more depth in our*[*next article*](https://stormpath.com/blog/manage-authentication-lifecycle-mobile)).

Now, when your API service generates an Oauth2 access token that your mobile app needs, of course you’ll need to store this in your mobile app somewhere.

**BUT WHERE?!**

Well, there are different places this token should be stored depending on what platform you’re developing against. If you’re writing an Android app, for instance, you’ll want to store all access tokens in [SharedPreferences](https://developer.android.com/reference/android/content/SharedPreferences.html) (*here’s the API docs you need to make it work*). If you’re an iOS developer, you will want to store your access tokens in the [Keychain](https://developer.apple.com/library/content/documentation/Security/Conceptual/keychainServConcepts/02concepts/concepts.html).

If you still have questions, the following two StackOverflow posts will be very useful — they explain not only how you should store access tokens a specific way, but why as well:

* [Where should I store access tokens on Android?](https://stackoverflow.com/questions/10161266/how-to-securely-store-access-token-and-secret-in-android)
* [Where should I store access tokens on iOS?](https://stackoverflow.com/questions/5793128/access-tokens-persistence-best-practices-ios)
* Should access tokens for services like Twitter and Facebook be encrypted? In particular, should tokens be stored on the the device's Keychain vs. UserDefaults? What are some possible security issues that could arise if a user's device is stolen/taken
* This is what I have come up with so far.
* **Pros of Keychain:** Encrypted
* Cons: No way to clean up when user removed app
* **Pros of UserDefaults:** Kept inside the app.
* Cons: No encryption.

## **Access Tokens**

Here’s the short answer: an access token can technically be anything you want:

* A random number
* A random string
* A UUID
* etc.

As long as you can:

* Issue it to a client,
* Verify that it was created by you (using a strong signature),
* And assign it an expiration time…

You’re golden!

**BUT…** With that said, there are some conventions you’ll probably want to follow.

Instead of handling all this stuff yourself, you can instead **create an access token that’s a**[**JWT**](http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html)**(JSON Web Token)**. It’s a relatively new specification that allows you to generate access tokens that:

* Can be issues to clients.
* Can be verified as being created by you (more on this later).
* Can expire automatically at a specific time.
* Can hold variable JSON information.
* Can reduce the amount of API calls to your service by allowing users to validate / verify their API credentials LOCALLY, without querying your service!

JWTs also look like a **randomly generated string**: so you can always store them as strings when using them. This makes them really convenient to use in place of a traditional access token as they’re basically the same thing, except with way more benefits.

JWTs are almost always cryptographically signed. The way they work is like so:

* You store a **secure random string somewhere on your API server**. This should ideally just be a long random string (40 characters or so is fine).
* When you **create a new JWT**, you’ll pass this **random string** into your JWT library to **sign the token**, along with any JSON data you want to store: a user account ID, an email address, what permissions this user has, etc.
* This token will be generated, and it’ll look something like this: header.claims.signature — where header, claims, and signature are just long [base64](https://en.wikipedia.org/wiki/Base64) encoded strings. This isn’t really important to know though.
* **You give this token to your user**: likely an API client (eg: your mobile app).

Now, from the mobile client, you can view whatever is stored in the JWT. So if I have a JWT, I can easily check to see what JSON data is inside it. Usually it’ll be something like:

{

  "user\_id": "e3457285-b604-4990-b902-960bcadb0693",

  "scope": "can-read can-write"

}

Now, this is a 100% fictional example, of course, but you get the idea: if I have a copy of this JWT token, I can see the JSON data above, yey!

But I can also verify that it is still valid because the JWT spec supports expiring tokens automatically. So when you’re using your JWT library in whatever language you’re writing, you’ll be able to verify that the JWT you have is valid and hasn’t yet expired (cool).

This means that if you use a JWT to access an API service, you’ll be able to tell whether or not your API call will work by simply **validating the JWT**! No API call required!

Now, once you’ve got a valid JWT, you can also do cool stuff with it on the server-side.

Let’s say you’ve given out a JWT to a mobile app that contains the following data:

|  |  |
| --- | --- |
| 1  2  3  4  5 | {    "user\_id": "e3457285-b604-4990-b902-960bcadb0693",    "scope": "can-read can-write"  } |

But let’s say some malicious program on the mobile app is able to modify your JWT so that it says:

|  |  |
| --- | --- |
| 1  2  3  4  5 | {    "user\_id": "e3457285-b604-4990-b902-960bcadb0693",    "scope": "can-read can-write can-delete"  } |

See how I added in the **can-delete** permission there? What will happen if this modified token is sent to our API server? Will it work? Will our server accept this modified JWT?

**NOPE!!**

When your **API service receives this JWT and validates it**, it’ll do a few things:

* It’ll check the token to make sure **it wasn’t tampered with**. It does this by using the secret randomly generated string that only the server knows. If the JWT was modified at all, this check will fail, and you’ll know someone is trying to do something nasty.
* It’ll also check the **expires time** of this JWT to make sure it’s actually valid. So if you assigned a JWT a long time ago that has long since expired, you’ll know that your client is trying to use an old token — so you can just reject their API request.

This is nice functionality, as it makes handling verification / expiration / security a lot simpler.

The only thing you need to keep in mind when working with JWTs is this: **you should only store stuff you don’t mind exposing publicly**.

As long as you follow the rule above, you really can’t go wrong with using JWTs.

The two pieces of information you’ll typically store inside of a JWT are:

* A **user account’s unique ID** of some sort. This way you can look this user up from your user database when you receive this JWT for authentication.
* **A user’s permissions**: what they can and can’t do. If you’re building a simple API service where all users are the same, this may not be necessary. But this way, users won’t need to hit your API to figure out what they can do all the time: instead, they can just look at their JWT.

So, that just about sums up JWTs. Hopefully, you now know why you should be using them as your OAuth access tokens — they provide:

* A nice way to validate tokens.
* A nice way to pass publicly-viewable information to a client.
* A simple interface for developers to work with.
* Built-in expiration.

Now, moving on — let’s talk about how this all works together…



**1**. User Opens App

The user opens the app! *Next!*

**2**. App Asks for Credentials

Since we’re going to be using the **OAuth2 password grant type scheme** to authenticate users against our API service, your app needs to ask the user for their *username or email* and *password*.

Almost *all* mobile apps ask for this nowadays, so users are used to typing their information in.

**3**. User Enters their Credentials

Next, the user enters their credentials into your app. Bam. Done. *Next!*

**4.** App Sends POST Requests to API Service

This is where the **initial OAuth2 flow** begins. What you’ll be doing is essentially making a simple HTTP POST request from your mobile app to your API service.

Here’s a command line POST request example using [cURL](https://curl.haxx.se/):

|  |  |
| --- | --- |
| 1  2 | $ curl --form **'grant\_type=password**&username=USERNAMEOREMAIL&password=PASSWORD' https://api.example.com/v1/oauth |

What we’re doing here is *POST’ing* the *username or email* and *password* to our API service using the **OAuth2 *password grant type***: (*there are several grant types, but this is the one we’ll be talking about here as it’s the only relevant one when discussing* ***building your own mobile-accessible API***).

**NOTE**: See how we’re sending the body of our POST request as form content? That is, **application/www-x-form-urlencoded**? This is what the OAuth2 spec wants =)

**5.** API Server Authenticates the User

What happens next is that your API service retrieves the incoming *username or email and password* data and validates the user’s credentials.

This step is very platform specific, but typically works like so:

1. You retrieve the user account from your database by *username or email*.
2. You compare the password hash from your database to the password received from the incoming API request. **NOTE**: Hopefully you store your passwords with [bcrypt](https://en.wikipedia.org/wiki/Bcrypt) ([password-hashing function](https://en.wikipedia.org/wiki/Password-hashing_function))
3. If the credentials are valid (*the user exists, and the password matches*), then you can move onto the next step. If not, you’ll return an error response to the app, letting it know that either the *username or email* and *password* are invalid.

**6.** API Server Generates a JWT that the App Stores

Now that you’ve authenticated the app’s OAuth2 request, you need to **generate an access token for the app**. To do this, you’ll use a JWT library to generate a useful access token, then return it to the app.

Here’s how you’ll do it:

1. Using whatever JWT library is available for your language, you’ll create a JWT that includes JSON data which holds the **user ID** (*from your database, typically*), all **user permissions** (*if you have any*), and any other data you need the app to immediately access.

Once you’ve generated a JWT, you’ll return a JSON response to the app that looks something like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | {    "**access\_token**": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJEUExSSTVUTEVNMjFTQzNER0xHUjBJOFpYIiwiaXNzIjoiaHR0cHM6Ly9hcGkuc3Rvcm1wYXRoLmNvbS92MS9hcHBsaWNhdGlvbnMvNWpvQVVKdFZONHNkT3dUVVJEc0VDNSIsImlhdCI6MTQwNjY1OTkxMCwiZXhwIjoxNDA2NjYzNTEwLCJzY29wZSI6IiJ9.ypDMDMMCRCtDhWPMMc9l\_Q-O-rj5LATalHYa3droYkY",  **"token\_type": "bearer",**    "**expires\_in": 3600**  } |

As you can see above, our JSON response contains 3 fields. The first field **access\_token,** is **the actual OAuth2 access token** that the mobile app will be using from this point forward in order to make authenticated API requests.

The second field, **token\_type**, simply tells the mobile app what type of access token we’re providing — in this case, we’re providing an [OAuth2 Bearer token](https://tools.ietf.org/html/rfc6750). I’ll talk about this more later on.

Lastly, the third field provided is the **expires\_in** field. This is basically the number of seconds for which the supplied access token is valid.

In the example above, what we’re saying is that we’re giving this mobile app an access token which can be used to access our private API for up to 1 hour — no more. After 1 hour (*3600 seconds*) this access token will expire, and any future API calls we make using that access token will fail.

On the mobile app side of things, you’ll **retrieve this JSON response**, **parse out the access token** that was provided by the API server, and then store it locally in a secure location. On Android, this means [SharedPreferences](https://developer.android.com/reference/android/content/SharedPreferences.html), on iOS, this means [Keychain](https://developer.apple.com/library/ios/documentation/Security/Conceptual/keychainServConcepts/02concepts/concepts.html).

Now that you’ve got an access token securely stored on the mobile device, you can use it for making all subsequent API requests to your API server.

**7.** App Makes Authenticated Requests to API Server

All that’s left to do now is to make secure API requests from your mobile app to your API service. The way you do this is simple.

In the last step, your mobile app was given an OAuth2 access token, which it then stored locally on the device.

In order to successfully make API requests using this token, you’ll need to create an [HTTP Authorization](https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#14.8) header that uses this token to identify your user.

To do this, what you’ll do is insert your access token along with the word Bearer into the HTTP Authorization header. Here’s how this might look using cURL:

|  |  |
| --- | --- |
| 1  2 | $ curl -H "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJEUExSSTVUTEVNMjFTQzNER0xHUjBJOFpYIiwiaXNzIjoiaHR0cHM6Ly9hcGkuc3Rvcm1wYXRoLmNvbS92MS9hcHBsaWNhdGlvbnMvNWpvQVVKdFZONHNkT3dUVVJEc0VDNSIsImlhdCI6MTQwNjY1OTkxMCwiZXhwIjoxNDA2NjYzNTEwLCJzY29wZSI6IiJ9.ypDMDMMCRCtDhWPMMc9l\_Q-O-rj5LATalHYa3droYkY" https://api.example.com/v1/test |

In the end, your Authorization header will look like this:

|  |  |
| --- | --- |
| 1  2 | Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJEUExSSTVUTEVNMjFTQzNER0xHUjBJOFpYIiwiaXNzIjoiaHR0cHM6Ly9hcGkuc3Rvcm1wYXRoLmNvbS92MS9hcHBsaWNhdGlvbnMvNWpvQVVKdFZONHNkT3dUVVJEc0VDNSIsImlhdCI6MTQwNjY1OTkxMCwiZXhwIjoxNDA2NjYzNTEwLCJzY29wZSI6IiJ9.ypDMDMMCRCtDhWPMMc9l\_Q-O-rj5LATalHYa3droYkY |

When your API service receives the HTTP request, what it will do is this:

1. Inspect the HTTP Authorization header value, and see that it starts with the word Bearer.
2. Next, it’ll grab the following string value, referring to this as the access token.
3. It’ll then validate this access token (*JWT*) using a JWT library. This step ensures the token is valid, untampered with, and not yet expired.
4. **It’ll then retrieve the user’s ID and permissions out of the token** (*permissions are optional, of course*).
5. **It’ll then retrieve the user account from the user database. (Jian: this shouldn’t be the case;**

**Since bearer token is self-contained; there shouldn’t be any need for database lookup)**

1. Lastly, it will ensure that what the user is trying to do is allowed, eg: the user must be allowed to do what they’re trying to do. After this is done, the API server will simply process the API request and return the result  
   normally.

Nothing anything familiar about this flow? You should! It’s almost the exact same way [HTTP Basic Authentication](https://en.wikipedia.org/wiki/Basic_access_authentication) works, with one main difference in execution: the HTTP Authorization header is slightly different (Bearer vs Basic).

Token Expiration

One of the most common questions we get here at [Stormpath](https://stormpath.com/" \o "Stormpath - A Simple User Management API), when talking about [token authentication for mobile devices](https://stormpath.com/blog/the-ultimate-guide-to-mobile-api-security), is about token expiration.

Developers typically ask us this:

*“This OAuth2 stuff with JSON Web Tokens sounds good, but how long should I allow my access tokens to exist before expiring them? I don’t want to force my users to re-authenticate every hour. That would suck.”*

This is an excellent question. The answer is a bit tricky though. Here are some general rules:

If you’re dealing with any form of sensitive data (*money, banking data, etc.*), don’t bother storing access tokens on the mobile device at all. When you authenticate the user and get an access token, just keep it in **memory**. When a users closes your app, your memory will be cleaned up, and the token will be gone. This will force users to log into your app every time they open  
it, but that’s a good thing.

For extra security, make sure your tokens themselves expire after a short period of time (*eg: 1 hour*) — this way, even if an attacker somehow compromises your access token, it’ll still expire fairly quickly.

If you’re building an app that holds sensitive data, that’s not related to money, you’re probably fine forcing tokens to expire somewhere around the range of every month. For instance, if I was building a mobile app that allowed users to take fitness progress photos of themselves to review at a later time, I’d use a 1 month setting.

The above setting is a good idea as it doesn’t annoy users, requiring them to re-input their credentials every time the open the app, but also doesn’t expose them to unnecessary risk. In the worst case scenario above, if a user’s access token is compromised, an attacker might be able to view this person’s progress photos for up to one month.

If you’re building a massive consumer application, like a game or social application, you should probably use a much more liberal expiration time: anywhere from 6 months to 1 year.

For these sorts of applications, there is very little risk storing an access token for a long period of time, as the service contains only low-value content that can’t really hurt a user much if leaked. If a token is compromised, it’s not the end of the world.

This strategy also has the benefit of not annoying users by prompting them to re-authenticate very frequently. For many mass consumer applications, signing in is considered a big pain, so you don’t want to do anything to break down your user experience.

Token Revocation

Let’s now talk about token revocation. What do you do if an access token is compromised?

Firstly, let’s discuss the odds of this happening. In general: they are very low. Using the recommended data stores for Android and iOS will greatly reduce the risk of your tokens being compromised, as the operating system provides a lot of built-in protections for storing sensitive data like access tokens.

But, let’s assume for this exercise that a user using your mobile app lost their phone, a saavy hacker grabbed it, broke through the OS-level protections, and was able to extract your API service’s access token.

*What do you do?*

This is where token revocation comes into play.

It is, in general, a good idea to support token revocation for your API service. What this means is that you should have a way to strategically invalidate tokens after issuing them.

**NOTE**: Many API services do *not* support token revocation, and as such, simply rely on token expiration times to handle abuse issues.

Supporting token revocation means you’ll have to go through an extra few steps when building this stuff out:

1. You’ll need to store all access tokens (*JWTs*) that you generate for clients in a database. This way, you can see what tokens you’ve previously assigned, and which ones are valid.
2. You’ll need to write an API endpoint which accepts an access token (*or user credentials*) and removes either the specific access token or *all* access tokens from a user’s account.

For those of you wondering how this works, the official  
[OAuth2 Revocation Spec](https://tools.ietf.org/html/rfc7009) actually talks about it in very simple terms. The gist of it is that you write an endpoint like /revoke that accepts POST requests, with the token or credentials in the body of the request.

The idea is basically this though: once you know that a given access token, or user account has been compromised, you’ll issue the appropriate revocation API request to your private API service. You’ll either revoke:

* The single access token, or
* All access tokens for a given user account.

We will cover the basics of JSON Web Tokens (JWT) vs. OAuth, token storage in cookies vs. HTML5 web storage (localStorage or sessionStorage), and basic security information about cross-site scripting (XSS) and cross-site request forgery (CSRF).

Let’s get started…

## **JSON Web Tokens (JWT): A Crash Course**

**The most implemented solutions for API authentication and authorization are the OAuth 2.0 and JWT specifications**, which are fairly dense. Cliff’s Notes Time! Here’s what you need to know about JWT vs OAuth:

* JWTs are a great authentication mechanism. They give you a structured and stateless way to declare [a user and what they can access](https://stormpath.com/product/authorization/). They can be cryptographically signed and encrypted to prevent tampering on the client side.
* JWTs are a great way to declare information about the token and authentication. You have a ton of freedom to decide what makes sense for your application because you are working with JSON.
* The concept behind scopes is powerful yet incredibly simple: you have the freedom to design your own access control language because, again, you are working with JSON.

If you encounter a token in the wild, it looks like this:

|  |  |
| --- | --- |
| 1  2 | "dBjftJeZ4CVP.mB92K27uhbUJU1p1r.wW1gFWFOEjXk…" |

This is a Base64 encoded string. If you break it apart you’ll actually find three separate sections:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9  .  eyJpc3MiOiJodHRwOi8vZ2FsYXhpZXMuY29tIiwiZXhwIjoxMzAwODE5MzgwLCJzY29wZXMiOlsiZXhwbG9yZXIiLCJzb2xhci1oYXJ2ZXN0ZXIiXSwic3ViIjoic3RhbmxleUBhbmRyb21lZGEuY29tIn0  .  edK9cpfKKlGtT6BqaVy4bHnk5QUsbnbYCWjBEE7wcuY |

The first section is a header that describes the token. The second section is a payload which contains the juicy bits, and the third section is a signature hash that can be used to verify the integrity of the token (if you have the secret key that was used to sign it).

When we magically decode the second section, the payload, we get this nice JSON object:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | {    "iss": "http://galaxies.com",    "exp": 1300819380,    "scopes": ["explorer", "solar-harvester", "seller"],    "sub": "tom@andromeda.com"  } |

This is the payload of your token. It allows you to know the following:

* Who this person is (sub, short for subject)
* What this person can access with this token (scope)
* When the token expires (exp)
* Who issued the token (iss, short for issuer)

These declarations are called ‘claims’ because the token creator claims a set of assertions that can be used to ‘know’ things about the subject. Because the token is signed with a secret key, you can verify its signature and implicitly trust what is claimed.

**Tokens are given to your users after they present some credentials, typically a username and password, but they can also provide API keys,** or even tokens from another service. This is important because it is better to pass a token (that can expire, and have limited scope) to your API than a username and password. If the username and password are compromised in a man-in-the-middle attack, it is like giving an attacker keys to the castle.

Stormpath’s [API Key Authentication Feature](http://docs.stormpath.com/guides/api-key-management/) is an example of this. The idea is that you present your hard credentials once, and then get a token to use in place of the hard credentials.

The JSON Web Token (JWT) specification is quickly gaining traction. Recommended highly by Stormpath, it provides structure and security, but with the flexibility to modify it for your application. Here is a longer article on it: [Use JWT the Right Way!](https://stormpath.com/blog/jwt-the-right-way/)

## **Where to Store Your JWTs**

So now that you have a good understanding of what a JWT is, the next step is to figure out how to store these tokens. **If you are building a web application, you have a couple of options**:

* HTML5 Web Storage (localStorage or sessionStorage)
* Cookies

To compare these two, let’s say we have a fictitious AngularJS or single page app (SPA) called galaxies.com with a login route (/token) to authenticate users to return a JWT. To access the other APIs endpoints that serve your SPA, the client needs to pass a valid JWT.

The **request for a token** that the single page app makes would resemble:

|  |  |
| --- | --- |
|  | HTTP/1.1  POST /token  Host: galaxies.com  Content-Type: application/x-www-form-urlencoded  username=tom@galaxies.com&password=andromedaisheadingstraightforusomg&grant\_type=password |

Your server’s response will vary based on whether you are using cookies or Web Storage. For comparison, let’s take a look at how you would do both.

### JWT localStorage or sessionStorage (Web Storage)

Exchanging a username and password for a JWT to store it in browser storage (sessionStorage or localStorage) is rather simple. The **response body for issuing Token would contain the JWT as an access token**:

|  |  |
| --- | --- |
|  | HTTP/1.1 200 OK    {    "**access\_token**": "eyJhbGciOiJIUzI1NiIsI.eyJpc3MiOiJodHRwczotcGxlL.mFrs3Zo8eaSNcxiNfvRh9dqKP4F1cB",         "**expires\_in**":3600     } |

On the client side, you would store the token in HTML5 Web Storage (assuming that we have a success callback):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function tokenSuccess(err, response) {      if(err){          throw err;      }      $window.sessionStorage.accessToken = response.body.access\_token;  } |

To pass the access token back to your protected APIs, you would use the HTTP Authorization Header and the Bearer scheme. The **request** that your SPA would make would resemble:

|  |  |
| --- | --- |
|  | HTTP/1.1  GET /stars/pollux  Host: galaxies.com  **Authorization: Bearer** eyJhbGciOiJIUzI1NiIsI.eyJpc3MiOiJodHRwczotcGxlL.mFrs3Zo8eaSNcxiNfvRh9dqKP4F1cB |

### JWT Cookie Storage

Exchanging a username and password for a JWT to store it in a cookie is simple as well. The **response would use the Set-Cookie HTTP header**:

|  |  |
| --- | --- |
|  | HTTP/1.1 200 OK  **Set-Cookie**: access\_token=eyJhbGciOiJIUzI1NiIsI.eyJpc3MiOiJodHRwczotcGxlL.mFrs3Zo8eaSNcxiNfvRh9dqKP4F1cB; Secure; HttpOnly; |

To pass the access token back to your protected APIs on the same domain, the browser would automatically include the cookie value. The request to your protected API would resemble:

|  |  |
| --- | --- |
|  | GET /stars/pollux  Host: galaxies.com    **Cookie**: access\_token=eyJhbGciOiJIUzI1NiIsI.eyJpc3MiOiJodHRwczotcGxlL.mFrs3Zo8eaSNcxiNfvRh9dqKP4F1cB; |

## **So, What’s the difference?**

If you compare these approaches, both receive a JWT down to the browser. Both are stateless because all the information your API needs is in the JWT. Both are simple to pass back up to your protected APIs. The difference is in the medium.

### JWT sessionStorage and localStorage Security

**Web Storage (localStorage/sessionStorage) is accessible through JavaScript on the same domain**. This means that any JavaScript running on your site will have access to web storage, and because of this can be vulnerable to cross-site scripting (XSS) attacks. XSS, in a nutshell, is a type of vulnerability where an attacker can inject JavaScript that will run on your page. Basic XSS attacks attempt to inject JavaScript through form inputs, where the attacker puts <script>alert('You are Hacked');</script> into a form to see if it is run by the browser and can be viewed by other users.

**To prevent XSS, the common response is to escape and encode all untrusted data**. But this is far from the full story. In 2015, modern web apps use JavaScript hosted on CDNs or outside infrastructure. Modern web apps include 3rd party JavaScript libraries for A/B testing, funnel/market analysis, and ads. We use package managers like Bower to import other peoples’ code into our apps.

What if only one of the scripts you use is compromised? Malicious JavaScript can be embedded on the page, and Web Storage is compromised. These types of XSS attacks can get **everyone’s** Web Storage that visits your site, without their knowledge. This is probably why a [bunch](https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet#Local_Storage) of [organizations](https://blog.whitehatsec.com/web-storage-security/) advise not to store anything of value or trust any information in web storage. This includes session identifiers and tokens.

As a storage mechanism, Web Storage does not enforce any secure standards during transfer. Whoever reads Web Storage and uses it must do their due diligence to ensure they always send the JWT over HTTPS and never HTTP.

### JWT Cookie Storage Security

Cookies, when used with the HttpOnly cookie flag, are not accessible through JavaScript, and are immune to XSS. You can also set the Secure cookie flag to guarantee the cookie is only sent over HTTPS. This is one of the main reasons that cookies have been leveraged in the past to store tokens or session data. **Modern developers are hesitant to use cookies because they traditionally required state to be stored on the server, thus breaking RESTful best practices**.

Cookies as a storage mechanism do not require state to be stored on the server if you are storing a JWT in the cookie. This is because the JWT encapsulates everything the server needs to serve the request.

However, cookies are vulnerable to a different type of attack: **cross-site request forgery (CSRF)**. A CSRF attack is a type of attack that occurs when a malicious web site, email, or blog causes a user’s web browser to perform an unwanted action on a trusted site on which the user is currently authenticated. This is an exploit of how the browser handles cookies. A cookie can only be sent to the domains in which it is allowed. By default, this is the domain that originally set the cookie. The cookie will be sent for a request regardless of whether you are on galaxies.com or hahagonnahackyou.com.

CSRF works by attempting to lure you to hahagonnahackyou.com. That site will have either an img tag or JavaScript to emulate a form post to galaxies.com and attempt to hijack your session, if it is still valid, and modify your account.

For example:

|  |  |
| --- | --- |
| 18  19  20  21 | <body>      <!– CSRF with an img tag –>      <img href="http://galaxies.com/stars/pollux?transferTo=tom@stealingstars.com" />      <!– or with a hidden form post –>      <script type="text/javascript">    $(document).ready(function() {      window.document.forms[0].submit();    });    </script>      <div style="display:none;">      <form action="http://galaxies.com/stars/pollux" method="POST">        <input name="transferTo" value="tom@stealingstars.com" />      <form>    </div>  </body> |

Both would send the cookie for galaxies.com and could potentially cause an unauthorized state change. CSRF can be prevented by using synchronized token patterns. This sounds complicated, but all modern web frameworks have [support](https://docs.angularjs.org/api/ng/service/$http#cross-site-request-forgery-xsrf-protection) [for](https://github.com/expressjs/csurf) [this](https://spring.io/blog/2013/08/21/spring-security-3-2-0-rc1-highlights-csrf-protection/).

For example, AngularJS has a solution to validate that the cookie is accessible by only your domain. Straight from [AngularJS docs](https://docs.angularjs.org/api/ng/service/$http):

When performing XHR requests, the $http service reads a token from a cookie (by default, XSRF-TOKEN) and sets it as an HTTP header (X-XSRF-TOKEN). Since only JavaScript that runs on your domain can read the cookie, your server can be assured that the XHR came from JavaScript running on your domain.

You can make this CSRF protection stateless by including a xsrfToken JWT claim:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {    "iss": "http://galaxies.com",    "exp": 1300819380,    "scopes": ["explorer", "solar-harvester", "seller"],    "sub": "tom@andromeda.com",    "xsrfToken": "d9b9714c-7ac0-42e0-8696-2dae95dbc33e"  } |

If you are using the [Stormpath SDK for AngularJS](https://github.com/stormpath/stormpath-sdk-angularjs), you get stateless CSRF protection with no development effort.

Leveraging your web app framework’s CSRF protection makes cookies rock solid for storing a JWT. CSRF can also be partially prevented by checking the HTTP Referer and Origin header from your API. CSRF attacks will have Referer and Origin headers that are unrelated to your application.

Even though they are more secure to store your JWT, cookies can cause some developer headaches, depending on if your applications require cross-domain access to work. Just be aware that cookies have additional properties (Domain/Path) that can be modified to allow you to specify where the cookie is allowed to be sent. Using AJAX, your server side can also notify browsers whether credentials (including Cookies) should be sent with requests with CORS.

### Conclusion

JWTs are an awesome authentication mechanism. They give you a structured way to declare users and what they can access. They can be encrypted and signed for to prevent tampering on the client side, but the devil is in the details and where you store them. Stormpath recommends that you store your JWT in cookies for web applications, because of the additional security they provide, and the simplicity of protecting against CSRF with modern web frameworks. HTML5 Web Storage is vulnerable to XSS, has a larger attack surface area, and can impact all application users on a successful attack

**JWT is an encoding standard** for tokens that contains a JSON data payload that can be signed and encrypted.

**JWT can be used for many things, among those are bearer tokens**, i.e. a piece of information that you can present to some service that by virtue of you having it (you being the "bearer") grants you access to something.

A Bearer token is just string, potentially arbitrary, that is used for authorization.

**Bearer tokens can be included in an HTTP request in different ways**, one of them (probably the preferred one) being the **Authorization header**. But **you could also put it into a request parameter, a cookie or the request body. That is mostly between you and the server you are trying to access**.

A few years ago, before the JWT revolution, a <token> was just a string with no intrinsic meaning, e.g. 2pWS6RQmdZpE0TQ93X. That token was then looked-up in a database, which held the **claims** for that token. The downside of this approach is that **DB access** (or a cache) is required everytime the token is used.

JWTs *encode* and *verify* (via signing) their own **claims**. This allows folks to issue short-lived JWTs that are stateless (read: self-contained, don't depend on anybody else). **They do not need to hit the DB**. This reduces DB load and simplifies application architecture because **only the service that issues the JWTs needs to worry about hitting the DB/persistence layer** (the **refresh\_token** you've probably come across).

**JWT (JSON Web Tokens)**- It is just a token format. JWT tokens are JSON encoded data structures contains information about issuer, subject (claims), expiration time etc. It is signed for tamper proof and authenticity and it can be encrypted to protect the token information using symmetric or asymmetric approach. JWT is simpler than SAML 1.1/2.0 and supported by all devices and it is more powerful than SWT(Simple Web Token).

**OAuth2** - OAuth2 solve a problem that user wants to access the data using client software like browse based web apps, native mobile apps or desktop apps. OAuth2 is just for authorization, client software can be authorized to access the resources on-behalf of end user using access token.

**OpenID Connect** - OpenID Connect builds on top of OAuth2 and add authentication. OpenID Connect add some constraint to OAuth2 like UserInfo Endpoint, ID Token, discovery and dynamic registration of OpenID Connect providers and session management. JWT is the mandatory format for the token.

**CSRF protection** - You don't need implement the CSRF protection if you do not store token in the browser's cookie.

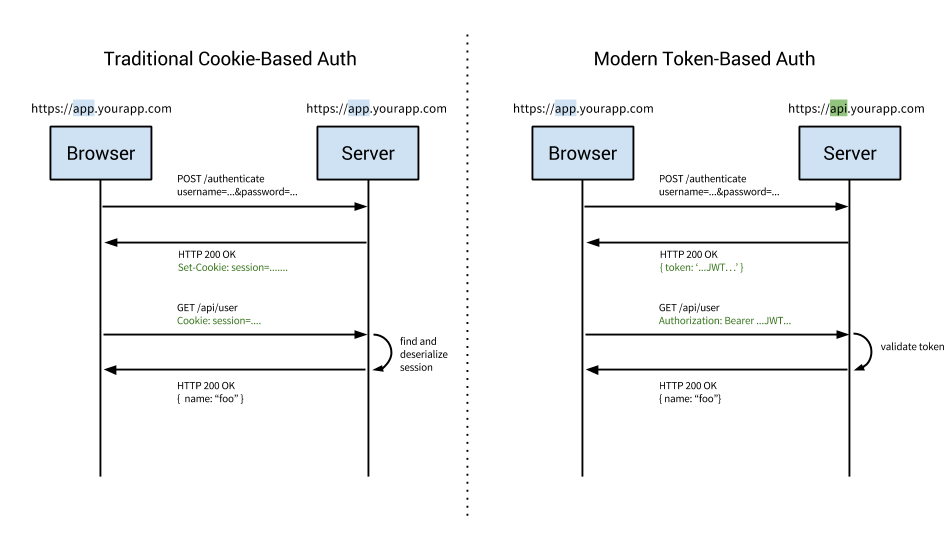
**find the main differences between JWT & OAuth**

1. OAuth 2.0 defines a protocol & JWT defines a token format.
2. OAuth can use either JWT as a token format or access token which is a bearer token.
3. OpenID connect mostly use JWT as a token format.

## Cookie vs. Token Authentication - Recap

Before we dive further, let's quickly recap how these two authentication systems work. If you are already familiar with how cookie and token authentication works, feel free to skip this section, otherwise read on for an in-depth overview.

This diagram is a great introduction and simplified overview of the difference between cookie and token approaches to authentication.



### Cookie-Based Authentication

Cookie-based authentication has been the default, tried-and-true method for handling user authentication for a long time.

Cookie-based authentication is **stateful**. This means that an authentication record or session must be kept both server and client-side. The server needs to keep track of active sessions in a database, while on the front-end a cookie is created that holds a session identifier, thus the name cookie based authentication. Let's look at the flow of traditional cookie-based authentication:

1. User enters their login credentials.
2. Server verifies the credentials are correct and creates a session which is then stored in a database.
3. A cookie with the session ID is placed in the users browser.
4. On subsequent requests, the session ID is verified against the database and if valid the request processed.
5. Once a user logs out of the app, the session is destroyed both client-side and server-side.

### Token-Based Authentication

Token-based authentication has gained prevalence over the last few years due to the rise of single page applications, web APIs, and the Internet of Things (IoT). When we talk about authentication with tokens, we generally talk about authentication with [JSON Web Tokens](https://jwt.io/introduction) (JWTs). While there are different ways to implement tokens, JWTs have become the de-facto standard. With this context in mind, the rest of the article will use tokens and JWTs interchangeably.

Token-based authentication is **stateless**. The server does not keep a record of which users are logged in or which JWTs have been issued. Instead, every request to the server is accompanied by a token which the server uses to verify the authenticity of the request. The token is generally sent as an addition Authorization header in the form of Bearer {JWT}, but can additionally be sent in the body of a POST request or even as a query parameter. Let's see how this flow works:

1. User enters their login credentials.
2. Server verifies the credentials are correct and returns a signed token.
3. This token is stored client-side, most commonly in local storage - but can be stored in session storage or a cookie as well.
4. Subsequent requests to the server include this token as an additional Authorization header or through one of the other methods mentioned above.
5. The server decodes the JWT and if the token is valid processes the request.
6. Once a user logs out, the token is destroyed client-side, no interaction with the server is necessary.

## Advantages of Token-Based Authentication

Understanding **how** something works is only half the battle. Next, we'll cover the reasons **why** token authentication is preferable over the traditional cookie-based approach.

### Stateless, Scalable, and Decoupled

Perhaps the biggest advantage to using tokens over cookies is the fact that token authentication is **stateless**. The back-end does not need to keep a record of tokens. **Each token is self-contained**, containing all the data required to check it's validity as well as convey user information through claims.

The server's only job, then, becomes to **sign tokens** **on a successful login request** and verify that incoming tokens are valid. In fact, the server does not even need to sign tokens. Third party services such as Auth0 can handle the issuing of tokens and then the server only needs to verify the validity of the token.

### Cross Domain and CORS

Cookies work well with singular domains and sub-domains, but when it comes to managing cookies across different domains, it can get hairy. In contrast, a token-based approach with CORS enabled makes it trivial to expose APIs to different services and domains. Since the JWT is required and checked with each and every call to the back-end, as long as there is a valid token, requests can be processed. There are a few caveats to this and we'll address those in the Common Questions and Concerns section below.

### Store Data in the JWT

With a cookie based approach, you simply store the session id in a cookie. JWT's, on the other hand, allow you to store any type of metadata, as long as it's valid JSON. The [JWT spec](https://tools.ietf.org/html/rfc7519) specifies different types of claims that can be included such as reserved, public and private. You can learn more about the specifics and the differences between the types of claims on the [jwt.io](https://jwt.io/introduction/) website.

In practice, what this means is that a JWT can contain any type of data. Depending on your use case you may choose to make the minimal amount of claims such as the user id and expiration of the token, or you may decide to include additional claims such as the user's email address, who issued the token, scopes or permissions for the user, and more.

### Performance

When using the **cookie-based authentication**, the back-end has to do a **lookup**, whether that be a traditional **SQL database** or a NoSQL alternative, and the round trip is likely to take longer compared to decoding a token. Additionally, since you can store additional data inside the JWT, such as the user's permission level, you can save yourself additional lookup calls to get and process the requested data.

For example, say you had an API resource /api/orders that retrieves the latest orders placed via your app, but only users with the role of **admin** have access to view this data. **In a cookie based approach**, once the request is made, you'd have one call to the database to **verify that the session is valid**, another to **get the user data and verify that the user has the role of admin**, and finally a third call to get the data. On the other hand, with a JWT approach, you can store the user role in the JWT, so once the request is made and the JWT verified, you can make a single call to the database to retrieve the orders.

### Mobile Ready

**Modern APIs do not only interact with the browser**. Written properly a single API can serve both the browser and native mobile platforms like iOS and Android. **Native mobile platforms and cookies do not mix well. While possible, there are many limitations and considerations to using cookies with mobile platforms**. Tokens, on the other hand, are much easier to implement on both iOS and Android. Tokens are also easier to implement for Internet of Things applications and services that do not have a concept of a cookie store.

## Common Questions and Concerns

In this section, we'll take a look at some common questions and concerns that frequently arise when the topic of token authentication comes up. The key focus here will be security but we'll examine use cases concerning token size, storage and encryption.

### JWT Size

The biggest disadvantage of token authentication is the **size of JWTs**. **A session cookie is relatively tiny compared to even the smallest JWT**. Depending on your use case, the size of the token could become problematic if you add many claims to it. Remember, each request to the server must include the JWT along with it.

### Where to Store Tokens?

With token-based auth, you are given the choice of where to store the JWT. Commonly, **the JWT is placed in the browser's local storage** and this works well for most use cases. There are some issues with storing JWTs in local storage to be aware of. Unlike cookies, local storage is sandboxed to a **specific domain** and **its data cannot be accessed** by any other domain including sub-domains.

**You can store the token in a cookie** instead, but the max size of a cookie is only 4kb so that may be problematic if you have many claims attached to the token. Additionally, you can store the token in session storage which is similar to local storage but is cleared as soon as the user closes the browser.

### XSS and XSRF Protection

Protecting your users and servers is always a top priority. One of the most common concerns developers have when deciding on whether to use token-based authentication is the security implications. Two of the most common attack vectors facing websites are Cross Site Scripting (XSS) and Cross-Site Request Forgery (XSRF or CSRF).

[**Cross Site Scripting**](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)) attacks occur when an outside entity is able to execute code within your website or app. The most common attack vector here is if your website allows inputs that are not properly sanitized. If an attacker can execute code on your domain, your JWT tokens are vulnerable. Our CTO has [argued](https://auth0.com/blog/2014/01/27/ten-things-you-should-know-about-tokens-and-cookies/#xss-xsrf) in the past that XSS attacks are much easier to deal with compared to XSRF attacks because they are generally better understood. Many frameworks, including Angular, automatically sanitize inputs and prevent arbitrary code execution. If you are not using a framework that sanitizes input/output out-of-the-box, you can look at plugins like [caja](https://github.com/google/caja) developed by Google to assist. Sanitizing inputs is a solved issue in many frameworks and languages and I would recommend using a framework or plugin vs building your own.

**Cross Site Request Forgery** attacks are not an issue if you are using JWT with local storage. On the other hand, if your use case requires you to store the JWT in a cookie, you will need to protect against XSRF. XSRF are not as easily understood as XSS attacks. Explaining how XSRF attacks work can be time-consuming, so instead, check out [this](http://www.gnucitizen.org/blog/csrf-demystified/) really good guide that explains in-depth how XSRF attacks work. Luckily, preventing XSRF attacks is a fairly simple matter. To over-simplify, protecting against an XSRF attack, your server, upon establishing a session with a client will generate a unique token (note this is not a JWT). Then, anytime data is submitted to your server, a hidden input field will contain this token and the server will check to make sure the tokens match. Again, as our recommendation is to store the JWT in local storage, you probably will not have to worry about XSRF attacks.

One of the best ways to protect your users and servers is to have a short expiration time for tokens. That way, even if a token is compromised, it will quickly become useless. Additionally, you may maintain a [**blacklist**](https://auth0.com/blog/2015/03/10/blacklist-json-web-token-api-keys/)of compromised tokens and not allow those tokens access to the system. Finally, the nuclear approach would be to change the signing algorithm, which would invalidate all active tokens and require all of your users to log in again. This approach is not easily recommended, but is available in the event of a severe breach.

### Tokens Are Signed, Not Encrypted

A JSON Web Token is comprised of three parts: the header, payload, and signature. The format of a JWT is header.payload.signature. If we were to **sign** a JWT with the **HMACSHA256 algorithm**, the **secret 'shhhh'** and the payload of:

1

{

2

"sub": "1234567890",

3

"name": "Ado Kukic",

4

"admin": true

5

}

The JWT generated would be:

1

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkFkbyBLdWtpYyIsImFkbWluIjp0cnVlLCJpYXQiOjE0NjQyOTc4ODV9.Y47kJvnHzU9qeJIN48\_bVna6O0EDFiMiQ9LpNVDFymM

The **very** important thing to note here is that **this token is signed by the HMACSHA256 algorithm**, and the **header and payload are Base64URL encoded, it is not encrypted**. If I go to [jwt.io](https://jwt.io/), paste this token and select the HMACSHA256 algorithm, I could decode the token and read its contents. Therefore, it should go without saying that sensitive data, such as passwords, should never be stored in the payload.

If you must store sensitive data in the payload or your use case calls for the JWT to be obscured, you can use JSON Web Encryption (JWE). JWE allows you to encrypt the contents of a JWT so that it is not readable by anyone but the server. [JOSE](http://jose.readthedocs.io/en/latest/) provides a great framework and different options for JWE and has SDKs for many popular frameworks including [NodeJS](https://github.com/cisco/node-jose) and [Java](https://bitbucket.org/connect2id/nimbus-jose-jwt/wiki/Home).

**The contents are entirely viewable**, but **they're signed using a secret key by the server** so it can tell if they've been tampered with.

Since everything is in the JWT, and the client can present it to whomever they want, you can use it for Single Sign On **as long as the different servers share the same secret** so they can **verify the signature**.

Like a ticket, a JWT has an expiry date. As long as it hasn't expired, it's valid. This means you can't revoke them before that. For this reason JWTs often have short expiry times (30 mins or so) and the client is also issued a refresh token in order to renew the JWT quickly when it expires.

**JWTs**

* **Not stored on the server**
* Great for SSO
* Can't be revoked prematurely

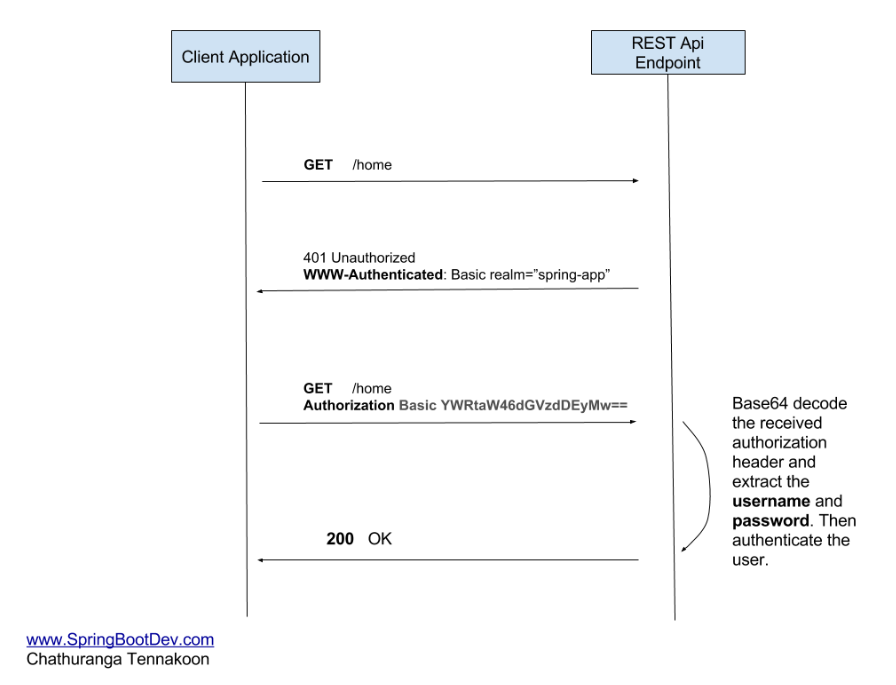
**Bearer tokens** are like a guest list. The server puts the client on the guest list, then provides a pass code to identify it when it wants something. When the client provides the code, the server looks it up on the list and checks that it's allowed to do whatever it's asking.

The server has to have the list available to it so if you want to share access across servers, they either all need to be able to access the list (database), or talk to some authority that has it (auth server).

On the other hand, since they have the guest list, they can take you off it whenever they want.

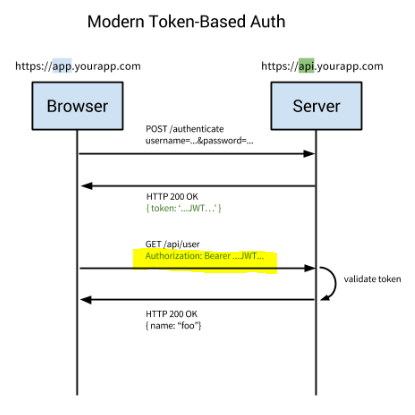
***Basic authentication*** transmits credentials as user ID/password pairs, encoded using base64. The client sends HTTP requests with the Authorization header that contains the word Basic word followed by a space and a base64-encoded string username:password.

Authorization: Basic ZGVtbzpwQDU1dzByZA==

[](https://i.stack.imgur.com/9ts46.png) Note: For basic authentication, as the user ID and password are passed over the network as clear text (it is base64 encoded, but base64 is a reversible encoding), the basic authentication scheme is not secure. **HTTPS / TLS should be used in conjunction with basic authentication.**

***Bearer authentication*** (also called **token authentication**) has security tokens called bearer tokens. The name “Bearer authentication” can be understood as “**give access to the bearer of this token**.” The bearer token is a cryptic string, usually generated by the server in response to a login request. The client must send this token in the Authorization header when making requests to protected resources:

Authorization: Bearer < token >

[](https://i.stack.imgur.com/HQF1L.png)

**Note:** Similarly to Basic authentication, Bearer authentication **should only be used over HTTPS (SSL)**.

**JWT Tokens: Great for Limiting Database Lookups**

Whereas API keys and OAuth tokens are always used to access APIs, [JSON Web Tokens](https://jwt.io/?utm_source=zapier.com&utm_medium=referral&utm_campaign=zapier) (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.eyJob21lcGFnZSI6Imh0dHBzOi8vemFwaWVyLmNvbS9kZXZlbG9wZXIvIiwidGFnbGluZSI6IlphcGllciBtYWtlcyB5b3UgaGFwcGllciIsIm1lc3NhZ2UiOiJHb29kIGpvYiEgWW91J3JlIGEgbWFzdGVyIGRlY29kZXIhIPCfkY8ifQ.qti9DKAJhwoTzu511CbVJ0g2cuSGbcIILjOiQ7yXp\_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.eyJob21lcGFnZSI6Imh0dHBzOi8vemFwaWVyLmNvbS9kZXZlbG9wZXIvIiwidGFnbGluZSI6IlphcGllciBtYWtlcyB5b3UgaGFwcGllciIsIm1lc3NhZ2UiOiJHb29kIGpvYiEgWW91J3JlIGEgbWFzdGVyIGRlY29kZXIhIPCfkY8ifQ.qti9DKAJhwoTzu511CbVJ0g2cuSGbcIILjOiQ7yXp\_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.

JSON Web Tokens (JWT) as OAuth 2.0 Bearer Access Tokens

Decentralized and Stateless API Security Architecture

[](https://johann-nallathamby.medium.com/?source=post_page-----89120c94c082--------------------------------)

[Johann Dilantha Nallathamby](https://johann-nallathamby.medium.com/?source=post_page-----89120c94c082--------------------------------)

Follow

[May 7, 2020](https://medium.com/identity-beyond-borders/json-web-tokens-jwt-as-oauth-2-0-bearer-access-tokens-89120c94c082?source=post_page-----89120c94c082--------------------------------) · 6 min read

Introduction

JSON Web Tokens (JWTs) [1] are rapidly becoming a popular choice for security tokens due to the following properties:

1. Their representation is compact, and they are small in size, which makes the transmission of a JWT efficient using the HTTP protocol.
2. They provide a structured way to declare users and entitlements.
3. They provide extensibility to the structure so that you can modify it to suit your application needs.
4. They can be signed to prevent tampering on the client side.
5. They can be encrypted to prevent sensitive information leakage on the client side.

Given these interesting and useful properties, it is not surprising that JWT bearer access tokens are rapidly catching up with the popularity of opaque bearer access tokens [2]. This is made possible due to the provision of extensible token types in OAuth 2.0.

**Benefits of JWT Bearer Access Tokens**

One of the major benefits of JWTs are that they are very portable. Modern applications tend to have many faces: the logged in view, the logged out view, or the restricted view. It’s all about fine-grained entitlements and access control. All the users get to use the same application, but they may have different levels of access based on fine-grained entitlements. **You’ll find yourself building authorization logic for your frontend and your backend**. JWTs instantly become a very suitable candidate for this purpose. **They can be used by your frontend and your backend for authorization**. Using the same token improves performance and developer experience. Typically, in addition to the JWT metadata, **the JWT bearer access tokens contain OAuth 2.0 metadata such as granted scopes and expiry time.** It could also contain user entitlements represented as roles and permissions depending on the specific implementations. This kind of information allows the application to do things like authorization and access token renewal interval calculation, entirely on the client-side.

Another major benefit of using JWTs **as bearer access tokens**, is that it makes the API security layer decentralized and stateless because all the information that your API security layer needs is in the JWT. What does this mean? With standard opaque bearer access tokens, the API Server needs to introspect the access tokens with the OAuth 2.0 authorization server by invoking its introspection endpoint. This means that the OAuth 2.0 authorization server holds information related to the validity of the access token which makes the API security layer **centralized and stateful**. In contrast, with JWT bearer access tokens, which is a kind of *self-contained* and *self-verifiable* access token, the **API Server can introspect the access token by itself**. In order to introspect the JWT bearer access token, **the API server could use the X509 public certificate to validate its signature, and establish the authenticity and integrity of the token**. It could also use the *exp* attribute to make sure that the access token is still active. Likewise there could be more standard and non-standard information in the JWT that can be used to validate and authorize the OAuth 2.0 client’s access to APIs. This greatly improves access patterns against the authorization server and reduces the load on it, and improves speed of token introspection during resource access. This decentralized and stateless solution has better scalability than a centralized and stateful system using opaque access tokens.

Caveats and Solutions

The decentralized and stateless architecture of JWT bearer access tokens however, does not come without drawbacks. One of the main drawbacks is the difficulty in **communicating changes to the JWT’s access rights**, due to changes in access rights of the user or client, or **revocation of the token**, across to the various components like the resource servers, unlike in the centralized and stateful architecture.

1. In a fully distributed and stateless implementation, the OAuth 2.0 authorization server will not store any metadata related to the issued JWT bearer access tokens. Hence there is no way of revoking an issued token at the OAuth 2.0 authorization server.
2. Even in some stateful implementations, only certain metadata information about the access token may be stored for audit purposes. The JWT’s access rights may be updated in this entry in the OAuth 2.0 authorization server. But still, in the decentralized architecture, since the API Server will not be directly interacting with OAuth 2.0 authorization server, the OAuth 2.0 authorization server cannot intercept the access token introspection process.

**So as an acceptable optimization, most vendors recommend having a low expiry time for access tokens and a long expiry time for refresh tokens.** Though not perfect, some organizations with non-critical use cases and relatively relaxed security policies, would be mostly fine with this optimization. This split way of handling tokens has other benefits too:

1. The shorter lifetime for leaked access tokens reduces the chance of a leaked token allowing access to a protected resource.
2. Since **refresh tokens** will continue to be **opaque tokens** and **not self-contained tokens**, they can continue to be used as an identifier to revoke the authorization grant.
3. Opaque refresh tokens don’t impact performance as much as opaque access tokens, since they are used more sparsely.
4. Even if refresh tokens are compromised, they cannot be used to renew the access token without the **client authentication** in the case of confidential clients.
5. Sliding-sessions (sessions that expire after a period of inactivity, generally considered a security best practice) can be easily implemented using shorter lifetime access tokens and longer lifetime refresh tokens.

Still for some organizations and business use cases this compromise may not be acceptable. One of the commonly considered solutions for this problem is changing the JWT signing key. By changing the JWT signing key, all the JWTs issued against the old key will immediately get revoked, as they cannot be verified using the new key. However, is a situation where the OAuth 2.0 authorization and resource servers are decoupled and distributed, implementing key rotation becomes much harder to the point that it is *not worth the candle*.

If a perfect solution needs to be sought for this problem, we need to maintain partial token state including the ‘*jti’* and *‘exp’* attributes of the JWT bearer access token in the OAuth 2.0 authorization server. When a JWT bearer access token is revoked in the OAuth 2.0 authorization server, the API servers have to be notified with the *‘jti’* that has been revoked and its *‘exp’* value. The API servers can maintain a revocation list of *‘jti’s*, until their expiry time is reached. With this we will lose the holy grail of stateless authentication check — but sometimes security requires tradeoffs.

Another place where many developers go wrong is, to use the **JWT access token payload to transfer logged-in user’s information other than entitlements**. It is highly recommended NOT to do this, since this information could contain sensitive information about the user, and you may inadvertently expose them to attackers. Also you’re probably going to want to cache it in the client-side for future reloads of the application, which would increase the attack surface even further. If the logged-in user’s information is needed for the application, it is recommended to use APIs over HTTPS, such as the OIDC Userinfo API or any other custom Rest API, which are also secured using OAuth 2.0. Another option is to use the OAuth 2.0 token response’s JSON payload to send this additional information to the application. You may use the JWT access token to transfer logged-in user information to the application, ONLY IF the information is deemed non-sensitive.

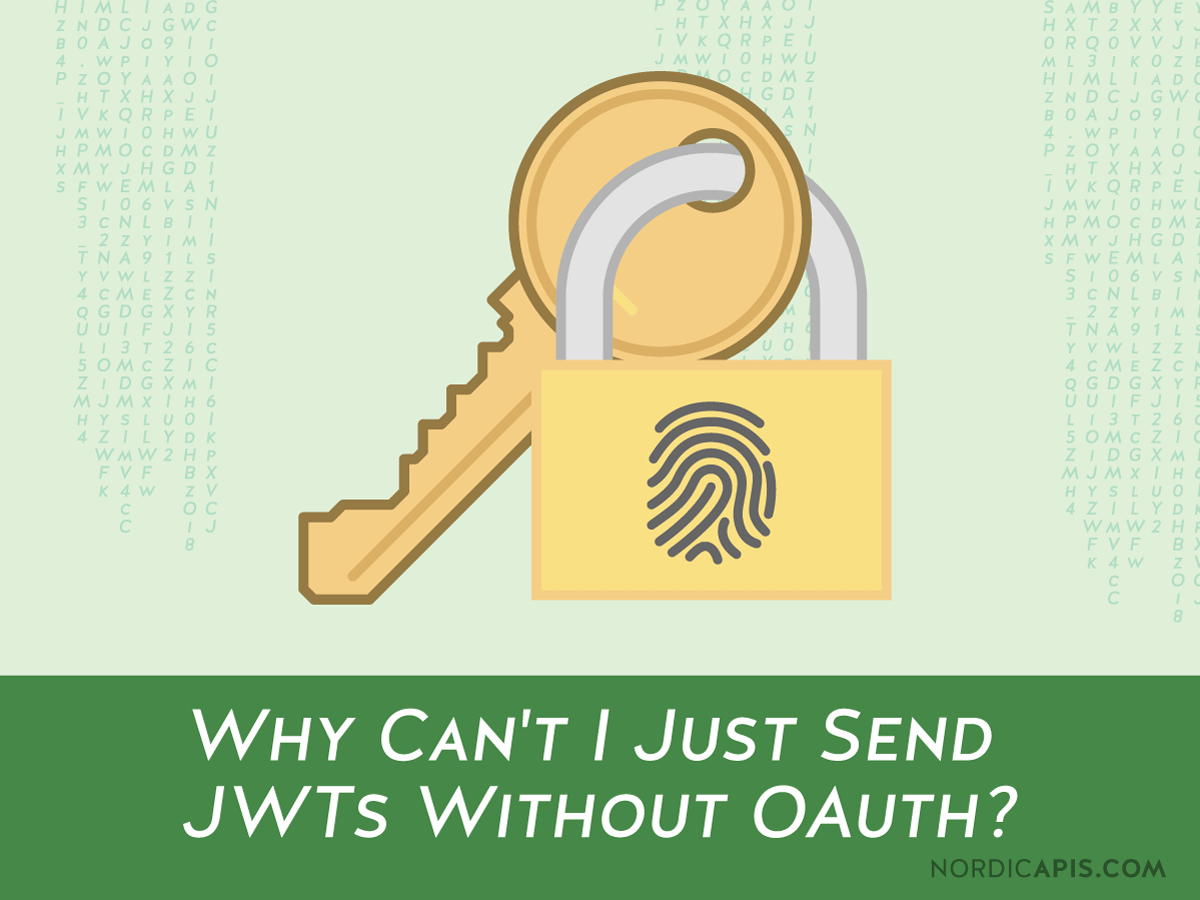
One more caution when using JWTs is, they might become a disadvantage if you store a lot of data inside it. Since it is sent back and forth with every request, it can slow down the communication, whereas with opaque tokens you are just sending a reference and looking up the data on the server side, which makes the transfer over the network faster.

# [**Why Can’t I Just Send JWTs Without OAuth?**](https://nordicapis.com/why-cant-i-just-send-jwts-without-oauth/)

[[](https://nordicapis.com/author/sandovaleffect/)](https://nordicapis.com/author/sandovaleffect/)

[Kristopher Sandoval](https://nordicapis.com/author/sandovaleffect/)

November 14, 2017



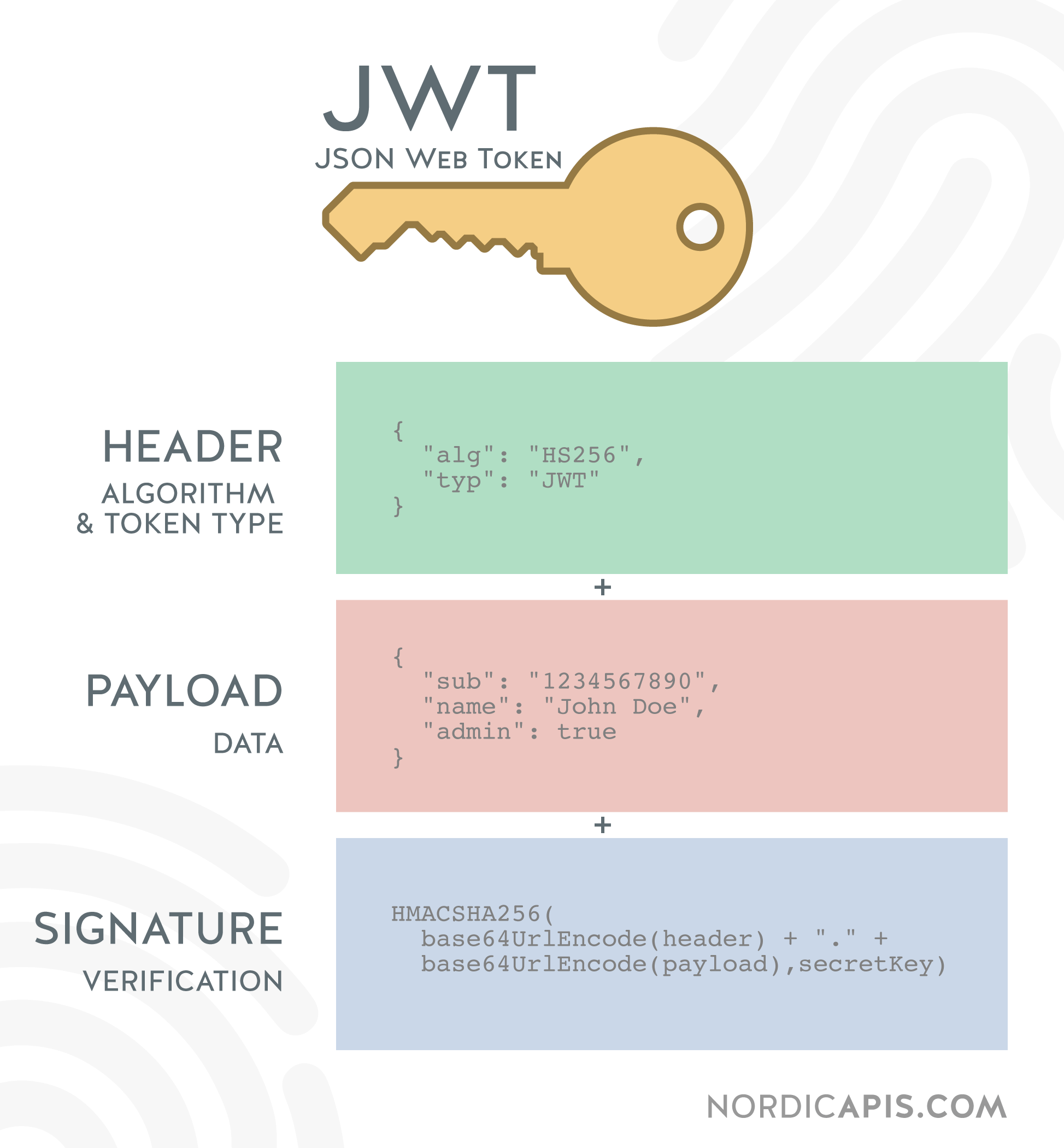
A **JSON Web Token** or **JWT** is an extremely powerful standard. It’s a signed JSON object; a compact token format often exchanged in HTTP headers to encrypt web communications.

Because of its power, JWTs can be found driving some of the largest modern API implementations. For many, the JWT represents a great solution that balances weight with efficiency, and as such, it’s often a very attractive standard to adopt for **API security**.

However, **a JWT should not be viewed as a complete solution**. Unfortunately, it seems that there are some significant misunderstandings as to what a JWT is, and how exactly it functions. In many situations, depending on JWTs alone can be extremely dangerous.

One of the most common questions about using JWTs is: Why can’t I send JWTs without OAuth? Today, we’re going to answer that very question. We’ll define what a JWT actually is, how it functions, and why adopting it in isolation is dangerous.

## **What is a JWT?**

[](http://19yw4b240vb03ws8qm25h366-wpengine.netdna-ssl.com/wp-content/uploads/Why-Cant-I-Just-Send-JWTs-Without-OAuth-JWT.png)Before we address why utilizing JWTs alone is insecure, we must define **what a JWT actually is**. This is because JWTs are often conflated with the additional protocols and systems surrounding them, meaning that the JWT design concept has been bolstered beyond the actual definition of the object itself.

JWT is an open standard defined by [RFC 7519](https://tools.ietf.org/html/rfc7519). The JWT is considered by its authors to be a “[compact and self-contained way for securely transmitting information between parties as a JSON object](https://jwt.io/introduction/).” The JWT itself is composed of a **Header**, a **Payload**, and a **signature** that proves the integrity of the message to the receiving server.

Content encoded inside a JWT is **digitally signed**, either using a secret utilizing the [HMAC algorithm](https://tools.ietf.org/html/rfc2104.html), or by leveraging the [Public Key Infrastructure](https://tools.ietf.org/html/rfc5280) (**PKI**) model with a private/public RSA configuration. While this does lend a certain amount of integrity protection, it does not specifically guarantee security — we will discuss this at greater length in just a moment, but it should be understood that **a JWT is an encoding format, and only an encoding format**.

JWTs are loved because they are small, and lend themselves to efficient transport as part of a URL, as part of the POST parameter, or even within the HTTP header. More lightweight transport options exist, and further extensions of the concept exist; **CWT** is a great example, utilizing **CBOR**, or Concise Binary Object Representation, to even further reduce the size of the package and improve efficiency.

The main benefit (and perhaps the main drawback from a security standpoint) of the JWT standard is that the encoded package is self-contained. The JWT package contains everything the system would need to know about the user, and as such, can be delivered as a singular object.

Also Read: [Using Open Standards Is Critical to API Longevity](https://nordicapis.com/building-with-open-standards-will-result-in-it-longevity/)

## **The Dangers of a Lone Solution**

JWTs are powerful — there’s simply no denying that. Unfortunately, many developers seem to think that the JWT is more than an encoding methodology, but a complete and secure implementation. This is often because JWTs are typically paired with a proper protocol and encryption standard in the wild — but this is a conscious choice, not the result of an automatic security due to the structure of the JWT itself.

**A JWT is only secure when it’s used in tandem with encryption and transport security methodologies**. JWT is a great encoding methodology, but it’s not a holistic security measure. Without additional protocols backing it up, a JWT is nothing more than an admittedly lightweight and slightly more secure [API key](https://nordicapis.com/why-api-keys-are-not-enough/).

For an example of this insecurity, let’s look at a common use case. A web API serves as the backend to a web application, and when the user generates a JWT, it is stored as an HTML5 data storage element. This is done so as to aid in the utilization of the API over multiple gateways and functions.

In this common situation, the issue is that the JWT is essentially exposed for common use. The JWT is digitally signed, which assures a certain amount of guaranteed integrity. The server itself is also set to reject any JWT with a manipulated Header, Payload, or Signature component, and as such, can reject a modified JWT token. That being said, the token doesn’t need to be modified in order to breach security. In theory, an attacker could take that token and use it in a sort of replay attack, getting resources that they do not have authorization to have.

While this type of attack can be somewhat mitigated through the use of expiration dates, this does nothing for **man-in-the-middle attacks**. In the MITM attack scheme, the expiration does not matter, as the attack is initiated live as a middleman.

These issues all arise from the simple fact that JWTs are a mechanism for transferring data — not for securing it.

Read Our [Deep Dive Into OAuth and OpenID Connect](https://nordicapis.com/api-security-oauth-openid-connect-depth/)

## **Securing JWTs**

JWTs are self-contained solutions containing everything the server needs to know about **who** the user is, **what** they need, and what they’re **authorized** to do. Accordingly, they’re great for stateless authentication, and work well with such methods geared for [stateless](https://nordicapis.com/defining-stateful-vs-stateless-web-services/) environments.

While there are a number of third party solutions and implementations of stateless authentication, the fact is that what you’d essentially be creating is a **bearer token** or, alternately, an **access token**.

That’s ok, and in fact, what we want to do, but this raises a simple question — if we are indeed creating such a bearer token, why not use the built-in functionality of the **OAuth** schema designed specifically to work with JWTs? There’s already a great deal of built-in security functionality in the OAuth specification that’s specifically engineered to support the JWT, so using **external solutions** — often the second question after why can’t I just sent JWTs without OAuth — is somewhat nonsensical.

If we utilize the OAuth 2.0 Bearer Token Usage standard under [RFC 6750](https://tools.ietf.org/html/rfc6750), which incorporates authorization headers, we can essentially create JWTs that would be recognized and specially treated by a wide variety of devices, from HTTP proxies to servers. We would thereby reduce data leakage, unintended storage of requests (as displayed above), and enable transport over something as simple as HTTPS.

Also Learn: [How to Handle Batch Processing With OAuth 2.0](https://nordicapis.com/how-to-handle-batch-processing-with-oauth%E2%80%932%E2%80%930/)

## **Proper JWT Utilization**

While it’s important to secure your JWTs, it’s also important to state what the proper utilization of a JWT within the OAuth schema would look like. While a JWT can serve many functions, let’s take a look at a common use case in the form of the **access token**. Both OAuth 2.0 and [OpenID Connect](https://nordicapis.com/3-unique-authorization-applications-of-openid-connect/) are vague on the type of access\_token, allowing for a wide range of functions and formats. That being said, the utilization of a JWT as that token is quite ubiquitous, for the benefits in efficiency and size already noted.

An access token is, in simple terms, is a token that is used by the API to make requests on behalf of the user who requested the token. It is part of the fundamental authorization mechanism within OAuth, and as such, **confidentiality** and **integrity** are extremely important. In order to generate an access token, an **authorization code** is required. All of the elements of this code are also extremely important to keep confidential and secure.

Accordingly, a JWT fits this role almost perfectly. Because of the aforementioned standards that allow for transmission over HTTPS, the JWT can contain all of the information needed to generate the access token. Once the token is generated, it can likewise be kept in JWT form as what is called a **self-encoded access token**.

The key benefit of handling the encoding of the access token in this way in the OAuth 2.0 schema is that **applications don’t have to understand your access token schema** — all of the information is encoded within the token itself, meaning that the schema can change fundamentally without requiring the clients to be aware, or even affected, by such changes.

Additionally, the JWT is great for this application because of the wide range of libraries that offer functionality such as expiration. A good example of this would be the [Firebase PHP-JWT library](https://github.com/firebase/php-jwt), which offers such expiration functionality.

The following code is taken from the [official OAuth documentation](https://www.oauth.com/oauth2-servers/access-tokens/self-encoded-access-tokens/), and demonstrates what such an implementation would look like during encoding:

$user\_id,

# Issuer (the token endpoint)

'iss' => 'https://' . $\_SERVER['PHP\_SELF'],

# Audience (intended for use by the client that generated the token)

'aud' => $client\_id,

# Issued At

'iat' => time(),

# Expires At

'exp' => time()+7200, // Valid for 2 hours

# The list of OAuth scopes this token includes

'scope' => $scope

);

$token\_string = JWT::encode($token\_data, $jwt\_key);

Such a mechanism would result in an **encoded string** that looks like this:

**eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9**

.eyJzdWIiOjEwMDAsImlzcyI6Imh0dHBzOi8

**vYXV0aG9yaXphdGlvbi1zZXJ2ZXIuY29tIiw**

**iYXVkIjoiaHR0cHM6Ly9leGFtcGxlLWFwcC5**

**jb20iLCJpYXQiOjE0NzAwMDI3MDMsImV4cCI**

6**MTQ3MDAwOTkwMywic2NvcGUiOiJyZWFkIHd**

**yaXRlIn0**.zhVmPMfS3\_Ty4qUl5ZMh4TipXsU

**CSH0mHzb4P\_Ijhxs**

## **Caveats**

Of course, as with any security implementation, there are caveats to consider. In the case of the JWT as a self-encoded authorization solution, replay attacks should be considered. While adopting proper encryption methodologies should negate many of those issues, the fact is that the issue is still fundamental to the concept as a whole, and should probably be addressed as a possibility rather than an impossible threat.

Accordingly, **caching** the authorization code for the lifetime of the code is the suggested solution from OAuth itself. By doing this, code can be verified against the known cached code for validity and integrity, and once the expiration date is reached, automatically rejected for date reasons.

It should also be noted that, due to the nature of the JWT, once an authorization code is issued, the JWT is self-contained — as such, it cannot technically be invalidated, at least in its most basic configuration. The JWT is designed to not hit the database for every verification, and when using a global secret, the JWT is valid until expiration.

There are a few ways around this, such as adding a **counter** in the JWT that increments upon certain events (such as role change, user data change, etc). This of course results in database [polling](https://nordicapis.com/stop-polling-and-consider-using-rest-hooks/) for each request, but the amount of data being checked is miniscule enough to make any processing increase somewhat negligible.

Additionally, at least in theory, you could use sections for specific functions, domains, and scopes, and change that secret when a breach is discovered. While this would affect more users than admins would like, it does have the effect of instituting revocation.

That being said, proper utilization of the JWT should make this largely a non-issue, as the user still has to provide a certain amount of secret information over an encrypted channel, and as such, should already be “vetted” or controlled.

[How to Control User Identity Within Microservices](https://nordicapis.com/how-to-control-user-identity-within-microservices/)

## **Don’t Leave JWT All Alone**

The simple fact is that JWTs are a great solution, especially when used in tandem with something like OAuth. Those benefits quickly disappear when used alone, and in many cases can result in **worse overall security**.

That being said, adopting the proper solutions can mitigate many of these threats, resulting in a more secure, efficient system. The first step to securing your JWT is to understand what it’s not — a JWT is an encoding method, not an encryption or transport security method, and as such, is only part of the puzzle.

What is your best solution for securing JWTs? Let us know in the comment section below.

JSON Web Token (JWT)

JWT defines a way in which certain common information pertaining to the process of authentication/authorization may be **represented**. As the name implies, the data format is **JSON**. JWTs carry certain **common fields** such as **subject**, **issuer**, **expiration time**, etc. JWTs become really useful when combined with other specs such as [JSON Web Signature (JWS)](https://tools.ietf.org/html/rfc7515) and [JSON Web Encryption (JWE)](https://tools.ietf.org/html/rfc7516). Together these specs provide not only all the information usually needed for an authorization token, but also a means to **validate the content** of the token so that it cannot be tampered with (JWS) and a way to **encrypt information** so that it remains **opaque** to the client (JWE). The simplicity of the data format (and its other virtues) have helped JWTs become one of the most common types of tokens.