August 17, 2017

Why JWTs Suck as Session Tokens

Randall Degges

JSON Web Tokens (JWTs) are *so* hot right now. They're all the rage in web development:

- Trendy? ✓
- Secure? ✓
- Scalable? ✓
- Compact? ✓
- JSON? ✓

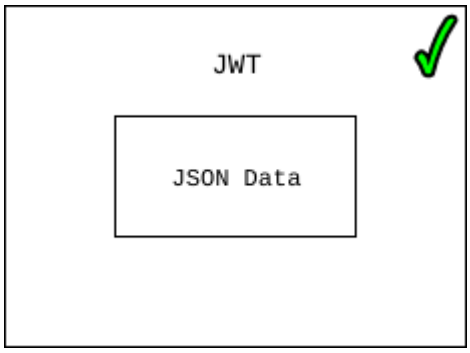
With all these amazing things going for JWTs, they seem like an unstoppable hype train headed straight for Stack Overflow fame and fortune!

But... today I'm here to talk with you about the downsides of using JWTs. Specifically, why it's a bad idea to use JWTs as session tokens for most people.

What are JWTs?

If you aren't already familiar with JWTs, don't panic! They aren't that complicated!

The way I like to think of JWTs is that they're just some JSON data that you can verify came from someone you know.



Pretend I'm blind and hard of hearing. Let's also pretend that last week you bought me lunch, and now I need your Venmo address to pay you back. If I ask you for your Venmo address in person, and someone else shouts *their* Venmo address, I might accidentally send *them* the money I owe *you*.

That's because I heard *someone* shout a Venmo address, and I trusted that it was *you*, even though in this case, it wasn't.

JWTs were designed to prevent this sort of thing from happening. JWTs give people an easy way to pass data between each other, while at the same time verifying who created the data in the first place.

So, going back to our previous example, if I received 1,000,000 different JWTs that contained a Venmo address, I'd easily be able to tell which one actually came from you.

How do JWTs Work?

JWTs are JSON data, encoded as a string, and cryptographically signed. I know that sounds fancy, but it really isn't.

The core of any JWT is claims. Claims are the JSON data inside the JWT. It's the data you care about, and want to pass along securely to someone else. I'm not going into the details here, but just know that JWTs hold JSON data.

How do JWTs make it so that you know whether or not they can be trusted? Cryptographic signatures.

Let's say I write a letter. When I sign that letter, I'm "signing" it. This means that anyone who reads that letter will know that I wrote it. And, because my signature is unique, there will be no question of its authenticity.

Cryptographic signatures work in much the same way. There are two main ways to "sign" JWTs cryptographically: using symmetric or asymmetric keys. Both types are commonly used, and provide the same guarantees of authenticity.

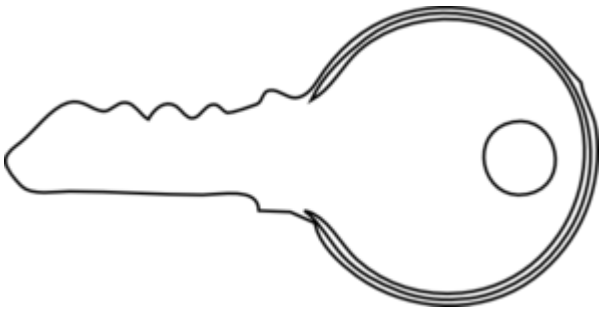
JWTs in a Nutshell

JWTs aren't magic–they're just blobs of JSON that have been cryptographically signed. Regardless of whether they're symmetrically or asymmetrically signed, they provide the same

guarantees: you can trust that a JWT is valid and created by someone you have faith in.

These properties of JWTs make them really useful in certain scenarios where you need to assert that some data can be trusted (such as when using federated login/single sign-on).

JWT Encryption Note



One final note I want to make about JWTs before moving on: their contents (the JSON data inside of them) are usually **not** encrypted. This means that anyone can view the data inside the JWT, even without a key.

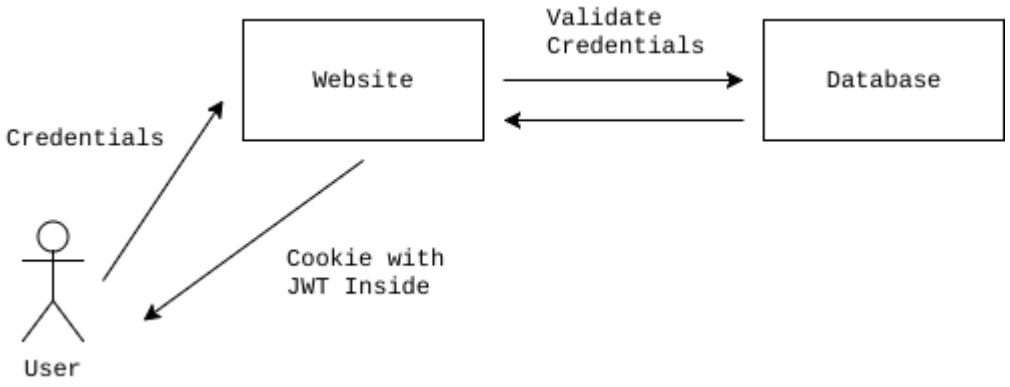
JWTs don't try to encrypt your data so nobody else can see it, they simply help you verify that it was created by someone you trust.

Now, if you *do* want to encrypt your JWTs you can do so by using JWE, but this is not nearly as common as unencrypted JWTs. If you are using JWTs and need encryption, be sure you use the right thing!

How are People Using JWTs Today?

The most common use case for JWTs is authentication. There are tons of web security libraries which use JWTs as session tokens, API tokens, etc.

The idea is that when someone authenticates to a website/API, the server will generate a JWT that contains the user's ID, as well as some other critical information, and then send it to the browser/API/etc. to store as a session token.



When that user visits another page on the website, for instance, their browser will automatically send that JWT to the server, which will validate the JWT to make sure that it's the same token it created originally, then let the user do stuff.

In theory, this sounds nice because:

- When the server receives a JWT, it can validate that it is legitimate and trust that the user is whoever the token says they are
- The server can validate this token locally without making any network requests, talking to a database, etc. This can potentially make session management faster because instead of needing to load the user from a database (or cache) on every request, you just need to run a small bit of local code. This is probably the single biggest reason people like using JWTs: they are stateless.

These two perks sound great because they will speed up webapp performance, reduce load on cache servers and database servers, and generally provide faster experiences.

As a bonus benefit, as the webapp creator you can embed other information about the user into your JWT:

- User permissions
- User personal information
- Etc.

This means that you can reduce your database load even further by simply embedding extra user information in your tokens as well!

Why Do JWTs Suck?

Now that we've seen how JWTs are used for authentication purposes, let's get into my favorite subject of all: why JWTs are not good session tokens.

I often argue with coworkers, colleagues, and friends about this, so it's nice to finally get all my thoughts on the subject down in bytes.

In particular, I plan to explain to you why normal old sessions are superior to JWTs in almost every way.

Context

Before I start making web developers all over the world angry, I want to provide some context into my reasoning.

Most websites that developers build are relatively simple:

- A user registers for the website
- A user signs into the website
- A user clicks around and does stuff
- The website uses the user's information to create, update, and delete information

99.9% of all websites match the criteria above.

For these types of websites, what's important to know is that almost every page a user interacts with contains some sort of dynamic data. Odds are, if you're running a website that requires a user to sign in to use it, you're going to be doing things with that user in your database often:

- Recording the actions a user is taking
- Adding some data for the user to the database
- Checking a user's permissions to see if they can do something
- Etc.

The important thing to remember is that most sites require user information for nearly every operation.

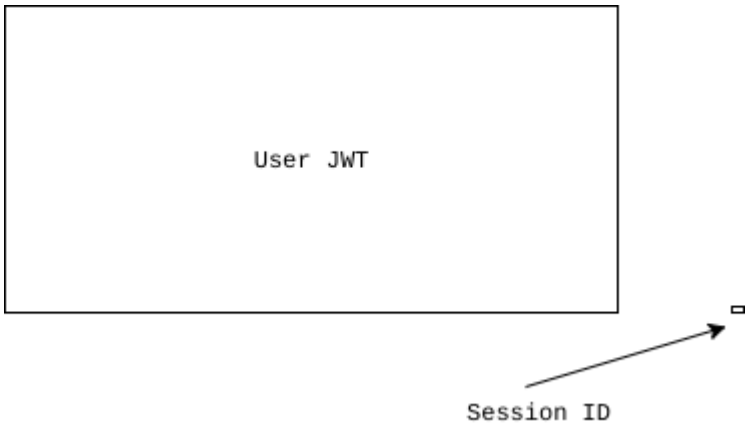
With that out of the way, let's get into the reasons why JWTs suck. First up? Size.

Size

Let's take a look at two scenarios:

- Storing a user ID (abc123) in a cookie
- Storing a user ID (abc123) in a JWT

If we store the ID in a cookie, our total size is 6 bytes. If we store the ID in a JWT (with basic header fields set, as well as a reasonably long secret), the size has now inflated to 304 bytes. For storing a simple user session, that is a ~51x size inflation on every single page request in exchange for cryptographic signing (as well as some header metadata).



For reference, here were the JWT claims I used to get that number:

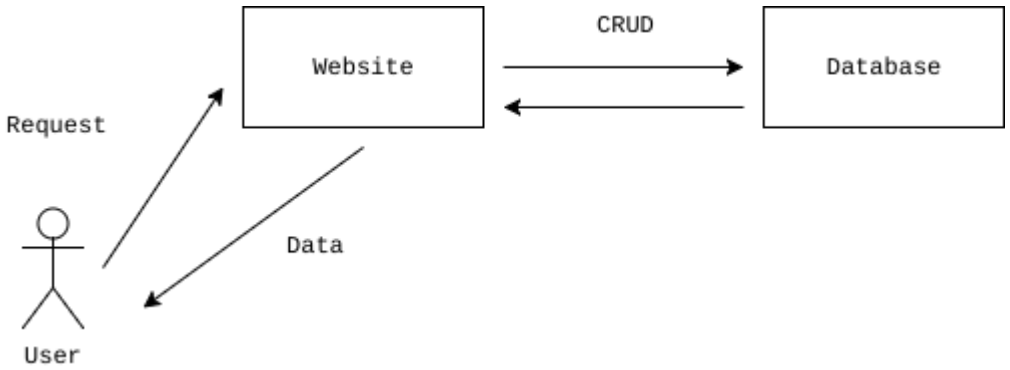
```
{
  "iss": "https://api.mywebsite.com",
  "sub": "abc123",
  "nbf": 1497934977,
  "exp": 1497938577,
  "iat": 1497934977,
  "jti": "1234567",
  "typ": "authtoken"
}
```

Let's say that your website gets roughly 100k page views per month. That means you'd be consuming an additional ~24MB of bandwidth each month. That doesn't sound like a lot, but when you're consistently bloating every single page request, all the little things start to add up.

Also: this example was using the smallest possible amount of information encoded in a JWT. In reality, many people end up storing much more information in JWTs than just a user ID, greatly increasing these byte counts.

You're Going to Hit the Database Anyway

As I mentioned above, most websites that require user login are primarily generating dynamic user content for CRUD operations (create, update, delete).



The issue with using JWTs on these websites is that for almost every single page the user loads, the user object needs to be loaded from a cache / database because one of the following situations are occurring:

- A mission critical user check needs to run (eg: does this user have enough money in their account to complete the transaction?)

- A database write needs to occur to persist information (if this information is related to the user, it's likely that the full user object must also be retrieved from the database)
- The full user object must be pulled out of the cache / database so that the website can properly generate its dynamic page content

Think about the websites you build. Do they often manipulate user data? Do they frequently use various fields on the user account to work? If so, your site falls into this category, and you'll likely be talking to the cache server / database regardless of whether or not you've got a JWT.

This means that on most websites, the stateless benefits of a JWT are not being taken advantage of.

To compound this issue, since JWTs are larger (in bytes) and also require CPU to compute cryptographic signatures, they're actually significantly slower than traditional sessions when used in this manner.

Almost every web framework loads the user on every incoming request. This includes frameworks like Django, Rails, Express.js (if you're using an authentication library), etc. This means that even for sites that are primarily stateless, the web framework you're using is still loading the user object regardless.

Finally: if you're storing your user information in a modern cache like memcached/redis, it's not uncommon over a VPC to achieve cache GET times of 5ms or below, which is *extremely* fast. I've personally used DynamoDB on Amazon in the past as a session store, and consistently achieved 1ms cache retrieval times. Because caching systems are so fast, there's very little performance overhead when retrieving users in this manner.

Redundant Signing

One of the main selling points of JWTs are cryptographic signatures. Because JWTs are cryptographically signed, a receiving party can verify that the JWT is valid, and trusted.

But... what would you say if I told you that in pretty much every single web framework created over the last 20 years, you could also get the benefits of cryptographic signatures when using plain old session cookies?

Well, you can =)

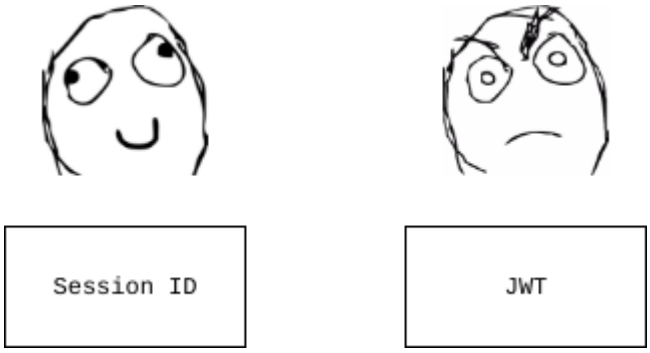
Most web frameworks cryptographically sign (and many encrypt!) your cookies for you automatically. This means that you get the exact same benefits as using JWT signatures without using JWTs themselves.

In fact, in most web authentication cases, the JWT data is stored in a session cookie anyways, meaning that there are now two levels of signing. One on the cookie itself, and one on the JWT.

While having two levels of signing may sound like a good idea, it is not. You get no security benefits, and you've now got to spend twice as long on CPU cycles to validate both signatures. Not really ideal for web environments where milliseconds are important. This is especially true in single threaded environments (cough cough *nodejs*) where number crunching can block your main event loop.

What's a Better Solution?

If JWTs suck, then what's a better solution? Plain old sessions!



If you're building a simple website like the ones described above, then your best bet is to stick with boring, simple, and secure server side sessions. Instead of storing a user ID inside of a JWT, then storing a JWT inside of a cookie: just store the user ID directly inside of the cookie and be done with it.

If your website is popular and has many users, cache your sessions in a backend like memcached or redis, and you can easily scale your service with very little hassle.

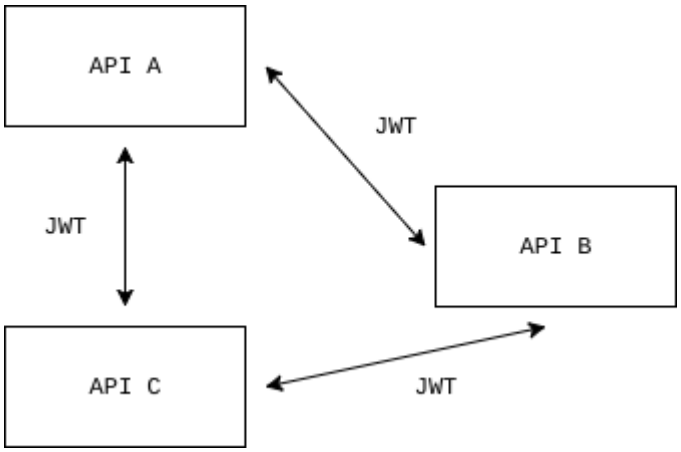
Excellent quality web frameworks like Django know this, which is why they operate this way.

How Should I Use JWTs?

It's important to note that I don't hate JWTs. I just think they're useless for a majority of websites. With that said, however, there are several cases in which JWTs can be useful.

If you're building API services that need to support server-to-server or client-to-server (like a mobile app or single page app (SPA)) communication, using JWTs as your API tokens is a very smart idea. In this scenario:

- You will have an authentication API which clients authenticate against, and get back a JWT
- Clients then use this JWT to send authenticated requests to other API services
- These other API services use the client's JWT to validate the client is trusted and can perform some action without needing to perform a network validation



For these types of API services, JWTs make perfect sense because clients will be making requests frequently, with limited scope, and usually authentication data can be persisted in a stateless way without too much dependence on user data.

If you're building any type of service where you need three or more parties involved in a request, JWTs can also be useful. In this case the requesting party will have a token to prove their identity, and can forward it to the third (or 4th ... nth) service without needing to incur a real-time validation each and every time.

Finally: if you're using user federation (things like single sign-on and OpenID Connect), JWTs become important because you need a way to validate a user's identity via a third party. Thanks to cryptographic signing, JWTs make a valuable addition to federated user protocols.

Wrap Up

When you start building your next website, just rely on your web framework's default authentication libraries and tools, and stop trying to shove JWTs into the mix unnecessarily.

Finally, if you're interested in web security and all sorts of other interesting problems in the authentication space, you should consider signing up for an Okta Developer Account. Our API service stores user accounts for your web apps, mobile apps, and API services, and makes web security fun again. Or, if you're in the mood to read some other interesting security articles, we've got a new security site where we publish lots of stuff like this.

If you have any questions, feel free to hit me up on Twitter @rdegges.

Randall Degges

Randall Degges runs Evangelism at Okta where he works on security research, development, and education. In his spare time, Randall writes articles and gives talks advocating for security best practices. Randall also builds and contributes to various open-source security tools.

Randall's realm of expertise include Python, JavaScript, and Go development, web security, cryptography, and infrastructure security. Randall has been writing software for ~20 years and has built some of the most-used API services on the internet.

Okta Developer Blog Comment Policy

We welcome relevant and respectful comments. Off-topic comments may be removed.

53 Comments **Okta Developer Blog** **Disqus' Privacy Policy** **Login**

Recommend 21 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS

Name

WoanVersace • 3 years ago • edited

This make me think about, why I should care about 24MB even 30MB band width each month. if I already have 100k page views per month, and they are all members.

```
Let's say that your website gets roughly 100k page views per month. That means you'd be consuming an additional ~24MB of bandwidth each month. That doesn't sound like a lot, but when you're consistently bloating every single page request, all the little things start to add up.
```

VISIT OKTA.COM

Social

GITHUB TWITTER FORUM RSS BLOG YOUTUBE

More Info

INTEGRATE WITH OKTA
BLOG
CHANGE LOG
3RD PARTY NOTICES
COMMUNITY TOOLKIT

Contact & Legal

CONTACT OUR TEAM
CONTACT SALES
CONTACT SUPPORT
TERMS & CONDITIONS
PRIVACY POLICY