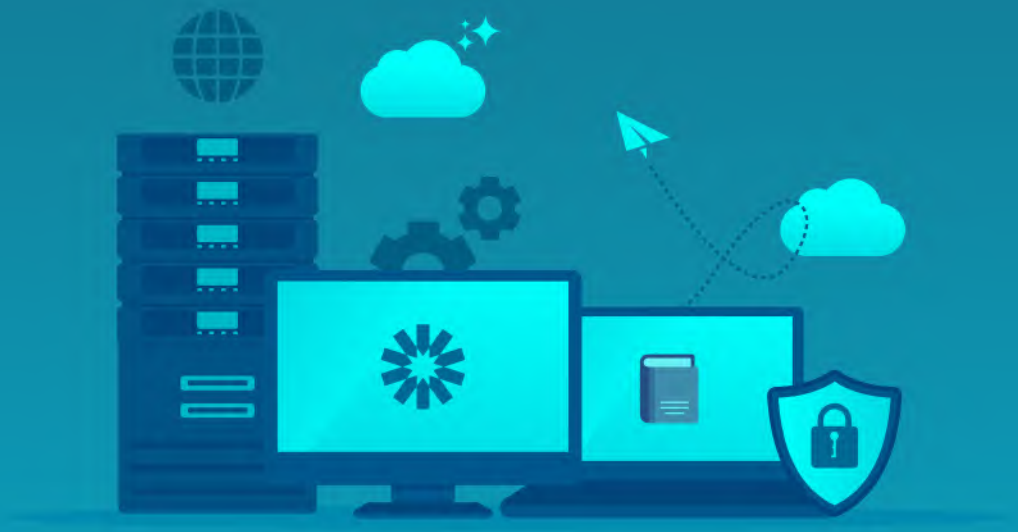
FusionAuth

Breaking down JSON Web Tokens: From pros and cons to building and revoking



Breaking down JSON Web Tokens

From pros and cons to building and revoking

The FusionAuth Team

This book is for sale at http://leanpub.com/json-web-tokens

This version was published on 2022-02-25

Contents

What are JWTs

First things first. JSON Web Tokens, or JWTs, are pronounced ‘jot’, not J-W-T. You’re welcome!

JWTs encapsulate arbitrary JSON in a standardized way, and are useful to communicate between different parts of a software system.

They are an IETF standard. The main RFC is 7519, but there are others as well. RFC 7515, RFC 7516, RFC 7517, RFC 7518 and RFC 7520 all concern this technology in one way or another.

There are two kinds of JWTs: signed and encrypted.

Signed JWTs allow you to cryptographically verify the integrity of the JWT. That means you can be assured the contents are unchanged from when the signer created it. However, signed JWTs do not protect the data carried from being seen; anyone who possesses a JWT can see its content. You don’t want to put anything in a JWT that should be a secret or that might leak information.

Encrypted JWTs, on the other hand, have a payload that cannot be read by those who do not possess the decryption key. If you have a payload that must be secret and both the creator and recipient of the JWT support it, encrypted JWTs are a good solution.

In general, signed JWTs are far more common. Unless otherwise noted, if this book uses the term JWT, it refers to a signed JWT.

JWTs are often used as stateless, portable tokens of identity. This usage will be the focus of this book, but what does that actually mean?

- They are stateless because the integrity of the information can be determined without contacting any remote service or

server. The aforementioned signature allows a consumer of a JWT to verify the integrity without any network access.
- They are portable because, even though they contain characters such as { that are typically not acceptable in certain contexts, JWTs use base64 URL encoding. This encoding ensures that the contents are safe for HTTP headers, cookies, and form parameters.
- Because of the flexibility of the JSON format, JWTs can encapsulate identity information, such as roles and user identifiers.

The combination of these attributes mean that JWTs are great for transporting identity information to different services. One service may authenticate the user and create a JWT for the client, and then other services, which offer different functionality and data depending on who the user is, can consume that JWT. This works especially well for APIs and microservices, which have minimal information about the user in their datastore. This is why many auth servers, also known as identity providers, issue JWTs.

You can sign a JWT with either a symmetric or asymmetric algorithm. Using a symmetric algorithm will be faster, but has significant security and operational ramifications. This is not unique to JWTs, because a symmetric algorithm like HS256 requires a shared secret. Therefore, when using a symmetric algorithm, any consumer of a JWT can also create JWTs indistinguishable from those created by an identity provider. Therefore, asymmetric solutions are recommended, even though they are slower. If performance is critical, make sure you benchmark your system to understand how signing algorithm choice affects both creation and consumption of JWTs.

Not all bearer tokens are JWTs and not all JWTs are bearer tokens, but the use case is common enough that it is worth mentioning. A bearer token is like a car key. If I have a car key, that gives me access to the car. The key doesn't care if I'm the owner, a friend or a thief.

In the same way, bearer tokens offer access to protected resources no matter who is presenting them. That means that you need to protect JWTs used in this way by using TLS for transport and by storing them safely in a location not accessible to other applications or rogue code.

If your use case is such that the risks of bearer tokens are unacceptable, there are a few options that cryptographically bind a token to a client. JWTs used in this way are not bearer tokens. While I won't be covering these solutions, both MTLS[1] and DPoP[2] may meet your needs.

As mentioned above, JWTs are produced by many identity providers. They are also widely supported elsewhere, having many articles, open source libraries and implementations available. I have yet to run into a major language that didn't have at least one library for creating and parsing JWTs. And, because JWTs depend on cryptographic operations and are often used as temporary credentials, using a well known, well vetted open source library to interact with JWTs is a good idea.

In the rest of this book, we'll cover different aspects of JSON Web Tokens and systems that use them.
What are JWTs

First things first. JSON Web Tokens, or JWTs, are pronounced 'jot', not J-W-T. You're welcome!

JWTs encapsulate arbitrary JSON in a standardized way, and are useful to communicate between different parts of a software system.

They are an IETF standard. The main RFC is 7519, but there are others as well. RFC 7515, RFC 7516, RFC 7517, RFC 7518 and RFC 7520 all concern this technology in one way or another.

There are two kinds of JWTs: signed and encrypted.

[1] https://datatracker.ietf.org/doc/html/rfc8705
[2] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop-04

Signed JWTs allow you to cryptographically verify the integrity of the JWT. That means you can be assured the contents are unchanged from when the signer created it. However, signed JWTs do not protect the data carried from being seen; anyone who possesses a JWT can see its content. You don't want to put anything in a JWT that should be a secret or that might leak information.

Encrypted JWTs, on the other hand, have a payload that cannot be read by those who do not possess the decryption key. If you have a payload that must be secret and both the creator and recipient of the JWT support it, encrypted JWTs are a good solution.

In general, signed JWTs are far more common. Unless otherwise noted, if this book uses the term JWT, it refers to a signed JWT.

JWTs are often used as stateless, portable tokens of identity. This usage will be the focus of this book, but what does that actually mean?

- They are stateless because the integrity of the information can be determined without contacting any remote service or server. The aforementioned signature allows a consumer of a JWT to verify the integrity without any network access.
- They are portable because, even though they contain characters such as `{` that are typically not acceptable in certain contexts, JWTs use base64 URL encoding. This encoding ensures that the contents are safe for HTTP headers, cookies, and form parameters.
- Because of the flexibility of the JSON format, JWTs can encapsulate identity information, such as roles and user identifiers.

The combination of these attributes mean that JWTs are great for transporting identity information to different services. One service may authenticate the user and create a JWT for the client, and then other services, which offer different functionality and data depending on who the user is, can consume that JWT. This works

especially well for APIs and microservices, which have minimal information about the user in their datastore. This is why many auth servers, also known as identity providers, issue JWTs.

You can sign a JWT with either a symmetric or asymmetric algorithm. Using a symmetric algorithm will be faster, but has significant security and operational ramifications. This is not unique to JWTs, because a symmetric algorithm like HS256 requires a shared secret. Therefore, when using a symmetric algorithm, any consumer of a JWT can also create JWTs indistinguishable from those created by an identity provider. Therefore, asymmetric solutions are recommended, even though they are slower. If performance is critical, make sure you benchmark your system to understand how signing algorithm choice affects both creation and consumption of JWTs.

Not all bearer tokens are JWTs and not all JWTs are bearer tokens, but the use case is common enough that it is worth mentioning. A bearer token is like a car key. If I have a car key, that gives me access to the car. The key doesn't care if I'm the owner, a friend or a thief.

In the same way, bearer tokens offer access to protected resources no matter who is presenting them. That means that you need to protect JWTs used in this way by using TLS for transport and by storing them safely in a location not accessible to other applications or rogue code.

If your use case is such that the risks of bearer tokens are unacceptable, there are a few options that cryptographically bind a token to a client. JWTs used in this way are not bearer tokens. While I won't be covering these solutions, both MTLS[3] and DPoP[4] may meet your needs.

As mentioned above, JWTs are produced by many identity providers. They are also widely supported elsewhere, having many articles, open source libraries and implementations available. I

[3] https://datatracker.ietf.org/doc/html/rfc8705
[4] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop-04

have yet to run into a major language that didn't have at least one library for creating and parsing JWTs. And, because JWTs depend on cryptographic operations and are often used as temporary credentials, using a well known, well vetted open source library to interact with JWTs is a good idea.

In the rest of this book, we'll cover different aspects of JSON Web Tokens and systems that use them.

Building a Secure Signed JWT

JSON Web Tokens (JWTs) get a lot of hate online for being insecure. Tom Ptacek, founder of Latacora[5], a security consultancy, had this to say about JWTs in 2017[6]:

> So, as someone who does some work in crypto engineering, arguments about JWT being problematic only if implementations are "bungled" or developers are "incompetent" are sort of an obvious "tell" that the people behind those arguments aren't really crypto people. In crypto, this debate is over.
>
> I know a lot of crypto people who do not like JWT. I don't know one who does.

Despite some negative sentiment, JWTs are a powerful and secure method of managing identity and authorization – they simply need to be used properly. They have other benefits too, they're flexible, standardized[7], stateless, portable, easy to understand, and extendable. They also have libraries to help you generate and consume them in almost every programming language.

This chapter will help make sure your JWTs are unassailable. It'll cover how you can securely integrate tokens into your systems by illustrating the most secure options.

However, every situation is different. You know your data and risk factors, so please learn these best practices and then apply

[5] https://latacora.com/
[6] https://news.ycombinator.com/item?id=14292223
[7] https://tools.ietf.org/html/rfc7519

them using judgement. A bank shouldn't follow the same security practices as a 'todo' SaaS application; take your needs into account when implementing these recommendations.

One additional note regarding the security of JWTs is that they are similar in many respects to other signed data, such as SAML assertions. While JWTs are often stored in a wider range of locations than SAML tokens are, always carefully protect any signed tokens.

Definitions

- **creator**: the system which creates the JWTs. In the world of OAuth this is often called an Authorization Server or AS.
- **consumer**: a system which consumes a JWT. In the world of OAuth this is often called the Resource Server or RS. These consume a JWT to determine if they should allow access to a Protected Resource such as an API.
- **client**: a system which retrieves a token from the creator, holds it, and presents it to other systems like a consumer.
- **claim**: a piece of information asserted about the subject of the JWT. Some are standardized, others are application specific.

Out of scope

This article will only be discussing signed JWTs. Signing of JSON data structures is standardized[8].

There are also standards for encrypting JSON data[9] but signed tokens are more common, so we'll focus on them. Therefore, in this article the term JWT refers to signed tokens, not encrypted ones.

[8] https://tools.ietf.org/html/rfc7515
[9] https://tools.ietf.org/html/rfc7516

Security considerations

When you are working with JWTs in any capacity, be aware of the footguns that are available to you (to, you know, let you shoot yourself in the foot).

The first is that a signed JWT is like a postcard. Anyone who has access to it can read it. Though this JWT string may look illegible, it's trivial to decode:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJmdXNpb25h\
dXRoLmlvIiwiZXhwIjoxNTkwNzA4Mzg1LCJhdWQiOiIyMzhkNDc5My03M\
GRlLTQxODMtOTcwNy00OGVkOGVjZDE5ZDkiLCJzdWIiOiIxOTAxNmI3My\
0zZmZhLTRiMjYtODBkOC1hYTkyODc3Mzg2NzciLCJuYW1lIjoiRGFuIE1\
vb3JlIiwicm9sZXMiOlsiUkVUUklFVkVfVE9ET1MiXX0.8QfosnY2Zled\
xWajJJqFPdEvrtQtP_Y3g5Kqk8bvHjo
```

You can decode it using any number of online tools[10], because it's just three base64 encoded strings joined by periods.

Keep any data that you wouldn't want in the hands of someone else outside of your JWT. When you sign and send a token, or when you decode and receive it, you're guaranteed the contents didn't change. But you're not guaranteed the contents are unseen.

A corollary of that is that any information you do send should avoid unintentional data leakage. This includes identifiers. If a JWT includes a value like `123` for an id, that means anyone viewing it has a pretty good idea that there is an entity with the id of `122`. Use a GUID or random string for identifiers to avoid this issue. Likewise, because tokens are not encrypted, use TLS for transmitting them.

Don't send JWTs using an HTTP method that may be cached or logged. So don't append the token to a `GET` request as a parameter. If you must send it in a GET request, use an HTTP header. You can

[10] https://fusionauth.io/learn/expert-advice/dev-tools/jwt-debugger

also use other HTTP methods such as `POST`, which sends the JWT as a part of a request body. Sending the token value as part of a `GET` URL might result in the JWT being stored in a proxy's memory or filesystem, a browser cache, or even in web server access logs.

If you are using OAuth, be careful with the Implicit grant, because if not implemented correctly, it can send the JWT (the access token) as a request parameter or fragment. While ignored by proxies, that can be cached by browsers and accessed by any JavaScript running on a page. For example, the Docusign esign REST API[11] delivers the access token as a URL fragment. Oops.

Creating tokens

When you are creating a JWT, use a library. Don't implement this RFC yourself. There are lots of great libraries out there[12]. Use one.

Set the `typ` claim of your JWT header to a known value. This prevents one kind of token from being confused with one of a different type.

Signature algorithms

When using JWTs, choose the correct signing algorithm. You have two families of options, a symmetric algorithm like HMAC or an asymmetric choice like RSA or elliptic curves (ECC). The `"none"` algorithm, which doesn't sign the JWT and allows anyone to generate a token with any payload they want, should not be used and any JWTs that use this signing algorithm (which actually means they aren't signed) should be rejected immediately.

[11] https://developers.docusign.com/esign-rest-api/guides/authentication/oauth2-implicit#step-1-obtain-the-access-token

[12] https://openid.net/developers/jwt/

There are three factors in algorithm selection. Performance, security and operational concerns all will play a role in your decision.

Performance

The first thing to consider is performance. A symmetric signing algorithm like HMAC is faster. Here are the benchmark results using the `ruby-jwt` library, which signed and verified a token 50,000 times:

```
hmac sign
  4.620000   0.008000   4.628000 (  4.653274)
hmac verify
  6.100000   0.032000   6.132000 (  6.152018)
rsa sign
 42.052000   0.048000  42.100000 ( 42.216739)
rsa verify
  6.644000   0.012000   6.656000 (  6.665588)
ecc sign
 11.444000   0.004000  11.448000 ( 11.464170)
ecc verify
 12.728000   0.008000  12.736000 ( 12.751313)
```

Don't look at the absolute numbers, they're going to change based on the programming language, what's happening on the system during a benchmark run, and CPU horsepower. Instead, focus on the ratios.

RSA encoding took approximately 9 times as long as HMAC encoding. Using ECC took almost two and a half times as long to encode and twice as long to decode. The code is publicly available[13] if you'd like to take a look. The takeaway is that symmetric signatures are faster than asymmetric options.

[13] https://github.com/FusionAuth/fusionauth-example-ruby-jwt/blob/master/benchmark_algos.rb

If you have a choice between RSA and elliptic curve cryptography for a public/private key signing algorithm, choose elliptic curve cryptography, as it's easier to configure correctly, more modern, has fewer attacks, and is faster. You might have to use RSA if other parties don't support ECC, however.

Operational concerns

Another factor in choosing the correct signing algorithm is secret distribution, which is an operational as well as a security concern. HMAC requires a shared secret to decode and encode the token. This means you need some method to provide the secret to both the creator and consumer of the JWT.

If you control both parties and they live in a common environment, this is not typically a problem; you can add the secret to whatever secrets management solution you use and have both entities pull the secret from there.

However, if you want outside entities to be able to verify your tokens, choose an asymmetric option. This might happen if the consumer is operated by a different department or business. The token creator can use the JWK[14] specification to publish public keys, and then the consumer of the JWT can validate it using that key.

Security ramifications

The shared secret required for options like HMAC has security implications. The token consumer can create a JWT indistinguishable from a token built by the creator, because both have access to the algorithm and the shared secret. This means you'll need to secure both the creator and the consumer of your tokens equally.

By using public/private key cryptography to sign the tokens, the issue of a shared secret is bypassed. Because of this, using an asymmetric option allows a creator to provide JWTs to token

[14] https://tools.ietf.org/html/rfc7517

consumers that are not trusted. No system lacking the private key can generate a valid token.

Claims

Make sure you set your claims appropriately. The JWT specification is clear:

> The set of claims that a JWT must contain to be considered valid is context dependent and is outside the scope of this specification.

Therefore no claims are required by the RFC. But to maximize security, the following registered claims should be part of your token creation:

- `iss` identifies the issuer of the JWT. It doesn't matter exactly what this string is as long as it is unique, doesn't leak information about the internals of the issuer, and is known by the consumer.
- `aud` identifies the audience of the token. This can be a scalar or an array value, but in either case it should also be known by the consumer.
- `nbf` and `exp` claims determine the timeframe that the token is valid. The `nbf` claim can be useful if you are issuing a token for future use, as it declares the time before which the token is not valid ("not before"). The `exp` claim, a time beyond which the JWT is no longer valid, should always be set. These are both in seconds since the unix epoch.

Revocation

Because it is difficult to invalidate JWTs once issued–one of their benefits is that they are stateless, which means holders don't need

to reach out to any server to verify they are valid–you should keep their lifetime on the order of second, minutes or hours, rather than days or months.

Short lifetimes mean that should a JWT be stolen, the token will soon expire and no longer be accepted by the consumer.

But there are times, such as a data breach or a user logging out of your application, when you'll want to revoke tokens, either across a system or on a more granular level.

You have a few choices here. These are in order of how much effort implementation would require from the token consumer:

- Let tokens expire. No effort required here.
- Have the creator rotate the secret or private key. This invalidates all extant tokens signed with that key.
- Use a 'time window' solution in combination with webhooks. Revoking JWTs in general is covered in more depth here.

Keys

It's important to use a long, random secret when you are using a symmetric algorithm. Don't choose a key that is in use in any other system.

Longer keys or secrets are more secure, but take longer to generate signatures. To find the appropriate tradeoff, make sure you benchmark the performance. The JWK RFC[15] does specify minimum lengths for various algorithms.

The minimum secret length for HMAC:

> A key of the same size as the hash output (for instance, 256 bits for "HS256") or larger MUST be used with this algorithm.

[15] https://tools.ietf.org/html/rfc7518

The minimum key length for RSA:

> A key of size 2048 bits or larger MUST be used with these algorithms.

The minimum key length for ECC is not specified in the RFC. Please consult the RFC for more specifics about other algorithms.

You should rotate your token signing keys regularly. Ideally you'd set this up in an automated fashion.

Rotation renders all tokens signed with the old key invalid (unless the consumer caches the keys), so plan accordingly. It's best to allow for a grace period equal to the lifetime of a JWT.

Holding tokens

Clients request and hold tokens, which then then present to consumers to gain access to protected data or resources.

A client can be a browser, a mobile phone or anything else. A client receives a token from a token creator (sometimes through a proxy or a backend service that is usually part of your application).

Clients are then responsible for two things:

- Passing the token on to any token consumers for authentication and authorization purposes, such as when a web application makes an HTTP request to a backend or API.
- Storing the token securely.

Clients should deliver the JWT to consumers over a secure connection, typically TLS version 1.2 or later.

The client must store the token securely as well. How to do that depends on what the client actually is.

For a browser, you should avoid storing the JWT in localstorage or a cookie accessible to JavaScript. You should instead keep it in a cookie with the following flags:

- `Secure` to ensure the cookie is only sent over TLS.
- `HttpOnly` so that no rogue JavaScript can access the cookie.
- `SameSite` with a value of `Lax` or `Strict`. Either of these will ensure that the cookie is only sent to the domain it is associated with.

An alternative to a cookie with these flags would be using a web worker[16] to store the token outside of the main JavaScript context. You could also store the token in memory, which works great as long as the page isn't reloaded.

For a mobile device, store the token in a secure location. For example, on an Android device, you'd want to store a JWT in internal storage with a restrictive access mode[17] or in shared preferences[18]. For an iOS device, storing the JWT in the keychain[19] is the best option.

For other types of clients, use platform specific best practices for securing data at rest.

Consuming a JWT

Tokens must be examined as carefully as they are crafted. When you are consuming a JWT, verify the JWT to ensure it was signed correctly, and validate and sanitize the claims.

Just as with token creation, don't roll your own implementation; use existing libraries.

[16] https://gitlab.com/jimdigriz/oauth2-worker
[17] https://developer.android.com/training/articles/security-tips#StoringData
[18] https://developer.android.com/reference/android/content/SharedPreferences
[19] https://developer.apple.com/documentation/security/keychain_services

First, verify that the JWT signature matches the content. Any library should be able to do this, but ensure that the algorithm that the token was signed with, based on the header, is the same used to decode it.

In addition, verify the `kid` value in the header; make sure the key id matches the key you are using the validate the signature. It's worth mentioning again here that any JWTs using the `none` algorithm should be rejected immediately.

Once the signature checks out, validate the claims are as expected. This includes any implementation specific registered claims set on creation, as well as the issuer (`iss`) and the audience (`aud`) claims. A consumer should know the issuer it expects, based on out of band information such as documentation or deploy time configuration. Checking the `aud` claim ensures the JWT is meant for you.

Other claims matter too. Make sure the `typ` claim, in the header, is as expected. Check that the current time is within the JWT's lifetime; that is that 'now' is before the `exp` value and after the `nbf` value, if present. If you're concerned about clock skew, allow some leeway.

If any of these claims fail to match expected values, the consumer should provide only a minimal error message to the client. Just as authentication servers should not reveal whether a failed login was due to an non-existent username or invalid password, you should return the same error message and status code, `403` for example, for any invalid token. This minimizes the information an attacker can learn by generating JWTs and sending them to a consumer.

If you are going to use claims for further information processing, make sure you sanitize those values. For instance, if you are going to query a database based on a claim, use a parameterized query.

In conclusion

JWTs are a flexible technology, and can be used in many ways. We discussed a number of steps you can take, as either a creator, client or consumer of tokens, to ensure your JWTs are as secure as possible.

Pros and Cons of JWTs

This chapter provides an analysis of JWTs (JSON Web Tokens, pronounced "jot") from how they are used to pros and cons of using JWTs in your application.

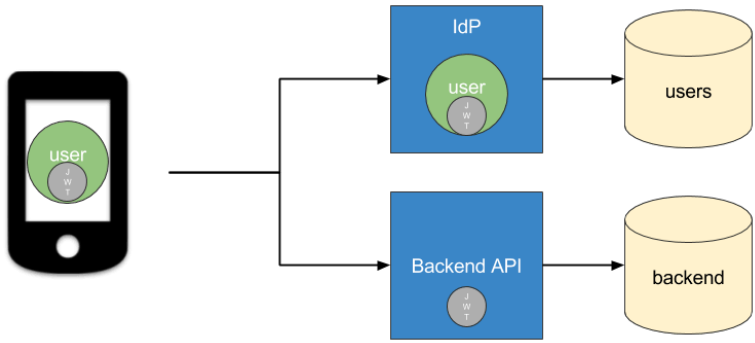
JWTs are becoming more and more ubiquitous. Customer identity and access management (CIAM) providers everywhere are pushing JWTs as the silver bullet for everything.

JWTs are pretty cool, but let's talk about some of the downsides of JWTs and other solutions you might consider.

One way to describe JWTs is that they are portable units of identity. That means they contain identity information as JSON and can be passed around to services and applications. Any service or application can verify a JWT itself.

The service/application receiving a JWT doesn't need to ask the identity provider that generated the JWT if it is valid. Once a JWT is verified, the service or application can use the data inside it to take action on behalf of the user.

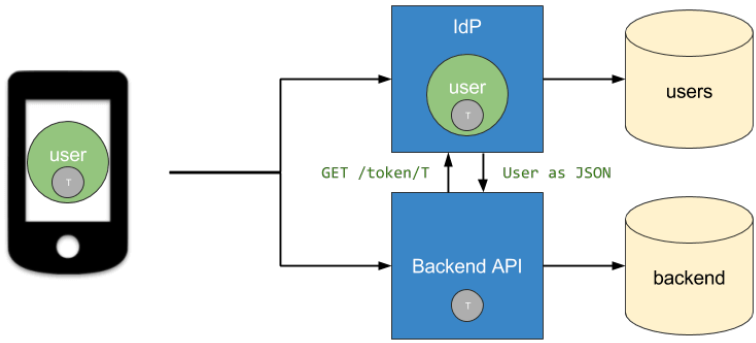
Here's a diagram that illustrates how the identity provider creates a JWT and how a service can use the JWT without calling back to the identity provider: (yes that is a Palm Pilot in the diagram)



JWT Example

When you contrast this with an opaque token, you'll see why so many developers are using JWTs. Opaque tokens are just a large string of characters that don't contain any data. A token must be verified by asking the identity provider if it is still valid and returning the user data the service needs. This is known as introspection.

Here's a diagram that illustrates how the identity provider is called to verify the opaque token and fetch the user data:



Opaque Token Example

This method of verifying and exchanging tokens can be very "chatty" and it also requires a method of persisting and loading

the tokens inside the identity provider. JWTs on the other hand don't require any persistence or logic in the identity provider since they are portable and standalone. Once a JWT has been issued, the identity provider's job is done.

There are a couple of things you should consider when deciding to use JWTs. Let's look at a few of the main ones.

JWTs expire at specific intervals

When a JWT is created it is given a specific expiration instant. The life of a JWT is definitive and it is recommended that it is somewhat small (think minutes not hours). If you have experience with traditional sessions, JWTs are quite different.

Traditional sessions are always a specific duration from the last interaction with the user. This means that if the user clicks a button, their session is extended. If you think about most applications you use, this is pretty common. You are logged out of the application after a specific amount of inactivity.

JWTs on the other hand, are not extended by any user interaction. Instead they either expire or are programmatically replaced by creating a new JWT for the user.

To solve this problem, most applications use refresh tokens. Refresh tokens are opaque tokens that are used to generate new JWTs. Refresh tokens also need to expire at some point, but they can be more flexible in this mechanism because they are persisted in the identity provider. This property makes them similar to the opaque tokens described above.

JWTs are signed

Since JWTs are cryptographically signed, they require a cryptographic algorithm to verify. Cryptographic algorithms are purposefully designed to be slow. The slower the algorithm, the higher the complexity, and the less likely that the algorithm can be cracked using brute-force methods.

As of 2019, on a quad-core MacBook Pro, about 200 JWTs can be created and signed per second using RSA public-private key signing. This number drops dramatically on virtualized hardware like Amazon EC2s. HMAC signing is much faster but lacks the same flexibility and security characteristics. Specifically, if the identity provider uses HMAC to sign a JWT, then all services that want to verify the JWT must have the HMAC secret. This means that all the services can now create and sign JWTs as well. This makes the JWTs less portable (specifically to public services) and less secure.

To give you an idea of the performance characteristics of JWTs and the cryptographic algorithms used, we ran some tests on a quad-core MacBook. These are from 2019, so the absolute numbers will change over time, but the general trends will not. Here are some of the metrics and timings we recorded for JWTs:

Metric | | Timing
— | | —
JSON Serialization + Base64 Encoding | | 400,000/s
JSON Serialization + Base64 Encoding + HMAC Signing | | 150,000/s
JSON Serialization + Base64 Encoding + RSA Signing | | 200/s
Base64 Decoding + JSON Parsing | | 400,000/s
Base64 Decoding + JSON Parsing + HMAC Verification | | 130,000/s
Base64 Decoding + JSON Parsing + RSA Verification | | 6,000/s

JWTs aren't easily revocable

This means that a JWT could be valid even though the user's account has been suspended or deleted. There are a couple of ways around this, discussed further in this chapter.

JWTs have exploits

This is more a matter of bad coding than flaws that are inherent to JWTs. The "none" algorithm and the "HMAC" hack are both well know exploits of JWTs. I won't go into details about these exploits here, but there are many discussions of them online.

Both of these exploits have simple fixes. Specifically, you should never allow JWTs that were created using the "none" algorithm.

Also, you should not blindly load signing keys using the "kid" header in the JWT. Instead, you should validate that the key is indeed the correct key for the algorithm specified in the header.

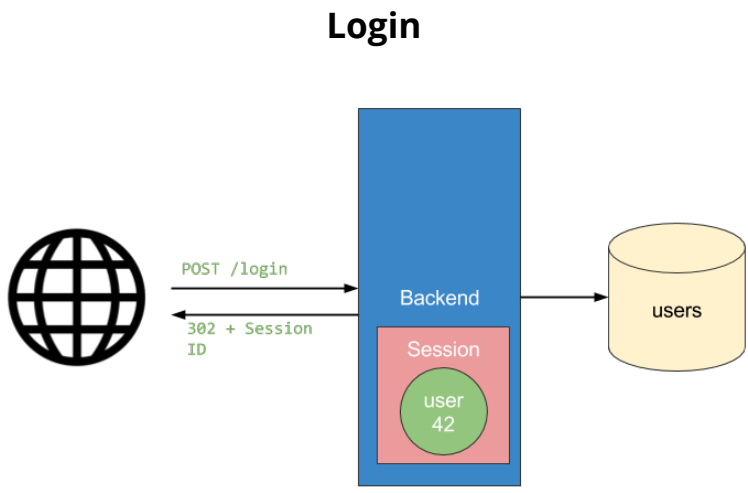
Sessions as an alternative

Instead of using JWTs or opaque tokens, you always have the option of using sessions. Sessions have been around for over two decades and are proven technology. Sessions generally work through the use of cookies and state that is stored on the server. The cookie contains a string of characters that is the session id. The server contains a large hash that keys off the session id and stores arbitrary data safely server-side.

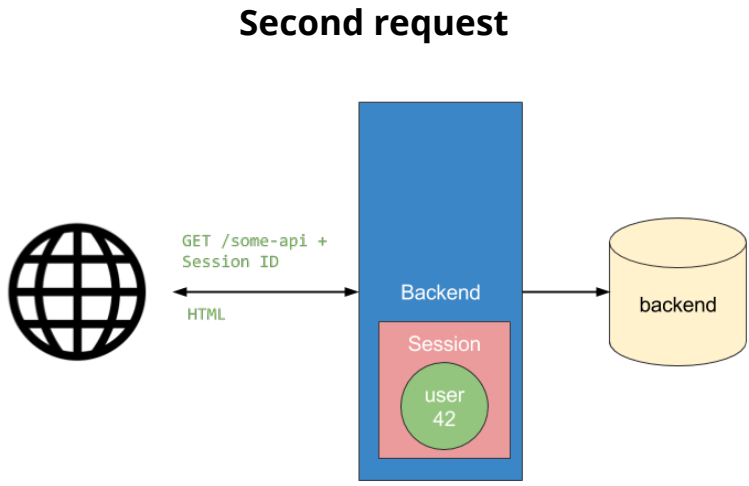
When a user logs in, the user object is stored in the session and the server sends back a session cookie that contains the session id. Each subsequent request to the server includes the session cookie.

The server uses the session cookie to load the user object out of the session Hash. The user object is then used to identify the user making the request.

Here are two diagrams that illustrate this concept:



Login example with sessions.



API call example with sessions.

If you have a smaller application that uses a single backend, sessions work well. Once you start scaling or using microservices, sessions can be more challenging.

Larger architectures require load-balancing and session pinning, where each client is pinned to the specific server where their session is stored. Session replication or a distributed cache might be needed to ensure fault tolerance or allow for zero-downtime upgrades. Even with this added complexity, sessions might still be a good option.

I hope this overview of JWTs and Sessions has been helpful in shedding some light on these technologies that are used to identity and manage users.

Either of these solutions will work in nearly any application. The choice generally comes down to your needs and the languages and frameworks you are using.

Revoking JWTs & JWT Expiration

Whenever I talk with developers about JSON Web Tokens (JWTs), one question keeps coming up: "How do I revoke a JWT?"

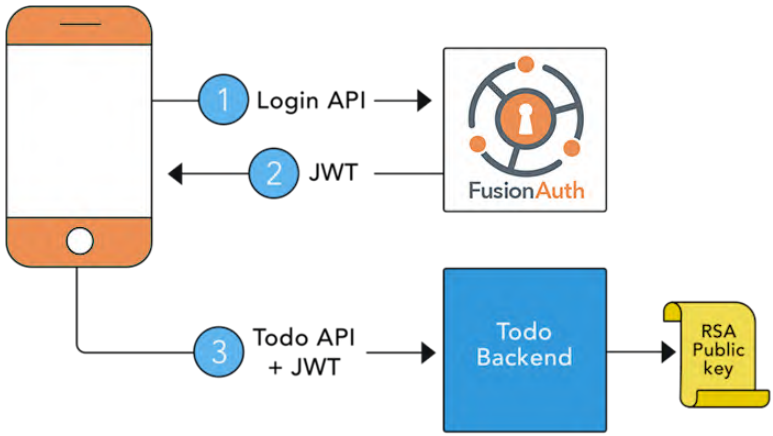
If you poke around online, you'll find that the most common answers are:

- Set the duration of the JWT to a short period (a few minutes or seconds)
- Implement complicated denylisting techniques
- Store every JWT so you can validate them

There is not a simple solution because JWTs are designed to be portable, decoupled representation of identities. Once you authenticate against an identity provider (IdP) and get back a JWT, you don't need to ever ask the IdP if the JWT is valid.

This is particularly powerful when you use ECC or RSA public/private key signing. The IdP signs the JWT using the private key and then any service that has the public key can verify the integrity of the JWT.

Here's a diagram that illustrates this architecture:



Revoking JWTs

The Todo Backend in the diagram can use the JWT and the public key to verify the JWT and then pull the user's id (in this case the subject) out of the JWT. The Todo Backend can then use the user's id to perform operations on that user's data.

However, because the Todo Backend isn't verifying the JWT with the IdP, it has no idea if an administrator has logged into the IdP and locked or deleted that user's account.

What are some solutions to this issue?

Reduce the duration of the JWT

The most common solution is to reduce the duration of the JWT and revoke the refresh token so that the user can't generate a new JWT.

With this approach, the JWT's expiration duration is set to something short (5-10 minutes) and the refresh token is set to something long (2 weeks or 2 months). At any time, an administrator can revoke the refresh token at the IdP, which means that the next time

the refresh token is presented, the user must re-authenticate to get a new JWT.

The refresh token, however, won't need to be presented until the user's JWT is invalid.

Here's where things get tricky. That user basically has that 5 to 10 minutes to use the JWT before it expires.

Once it expires, they'll use their current refresh token to try and get a new JWT. Since the refresh token has been revoked, this operation will fail and they'll be forced to login again.

It's this 5 to 10 minute window that freaks everyone out.

So, how do we fix it?

Rotate keys

If the Todo Backend pulls in the keys from the IdP every time it validates the signature of the JWT, then if the key is removed, the JWT will not validate.

Assume a JWT has a key identifier (`kid`) of `abcd`. During normal operation, the Todo Backend requests the list of public keys, and that includes one with the `kid` of `abcd`. That key can then be used to validate the signature of the JWT.

If the IdP removes the key `abcd` from the list, the JWT will no longer have a valid signature.

This approach has two downsides:

- The Todo Backend must continually poll the IdP for a list of valid keys. This obviates some of the distributed benefits of JWTs.
- All JWTs signed by the removed key are rendered invalid. This will affect many users.

Build a deny list

Another way leverages a distributed event system that notifies services when refresh tokens have been revoked. The IdP broadcasts an event when a refresh token is revoked and other backends/services listen for the event. When an event is received the backends/services update a local cache that maintains a set of users whose refresh tokens have been revoked.

This cache is checked whenever a JWT is verified to determine if the JWT should be revoked or not. This is all based on the duration of JWTs and expiration instant of individual JWTs.

Example: Revoking JWTs in FusionAuth

To illustrate this, I'm going to use FusionAuth[20]'s event and webhook system as well as the *jwt.refresh-token.revoke* event. If you are building your own IdP or using another system, you might need to build out your own eventing system based on this article.

The FusionAuth **jwt.refresh-token.revoke** event looks like this:

```
{
  "event": {
    "type": "jwt.refresh-token.revoke",
    "applicationTimeToLiveInSeconds": {
      "cc0567da-68a1-45f3-b15b-5a6228bb7146": 600
    },
    "userId": "00000000-0000-0000-0000-000000000001"
  }
}
```

Next, let's write a simple webhook in our application that will receive this event and update the JWTManager. (NOTE: our exam-

[20] https://fusionauth.io/

ple has a variable called `applicationId` that is a global variable that stores the id of the application itself - in this case it would be **cc0567da-68a1-45f3-b15b-5a6228bb7146**). Our code below is written in Node.js and uses the FusionAuth Node client library[21].

```
/* Handle FusionAuth event. */
router.post('/fusionauth-webhook', function(req, res, nex\
t) {
  JWTManager.revoke(req.body.event.userId, req.body.event\
.applicationTimeToLiveInSeconds[applicationId]);
  res.sendStatus(200);
});
```

Here is how the JWTManager maintains the list of user ids whose JWTs should be revoked.

This implementatin doesn't depend on unique ids in the JWT, but rather on the expiration time of the JWT and the time to live value configured for the application which created the JWT. You could also use the unique id of the JWT, often found in the `jti` claim, to implement this without the datetime math.

Our implementation also starts a thread to clean up after itself so we don't run out of memory.

```
const JWTManager = {
  revokedJWTs: {},

  /**
   * Checks if a JWT is valid. This assumes that the JWT \
contains a property named <code>exp</code> that is a
   * NumericDate value defined in the JWT specification a\
nd a property named <code>sub</code> that is the user id \
the
   * JWT belongs to.
```

[21] https://github.com/FusionAuth/fusionauth-node-client

```
11      *
12      * @param {object} jwt The JWT object.
13      * @returns {boolean} True if the JWT is valid, false i\
14  f it isn't.
15      */
16    isValid: function(jwt) {
17      const expiration = JWTManager.revokedJWTs[jwt.sub];
18      return expiration === undefined || expiration === nul\
19  l || expiration < jwt.exp * 1000;
20    },
21
22    /**
23      * Revokes all JWTs for the user with the given id usin\
24  g the duration (in seconds).
25      *
26      * @param {string} userId The user id (usually a UUID a\
27  s a string).
28      * @param {Number} durationSeconds The duration of all \
29  JWTs in seconds.
30      */
31    revoke: function(userId, durationSeconds) {
32      JWTManager.revokedJWTs[userId] = Date.now() + (durati\
33  onSeconds * 1000);
34    },
35
36    /**
37      * Cleans up the cache to remove old user's that have e\
38  xpired.
39      * @private
40      */
41    _cleanUp: function() {
42      const now = Date.now();
43      Object.keys(JWTManager.revokedJWTs).forEach((item, in\
44  dex, _array) => {
45        const expiration = JWTManager.revokedJWTs[item];
```

```
      if (expiration < now) {
        delete JWTManager.revokedJWTs[item];
      }
    });
  }
};

/**
 * Set an interval to clean-up the cache.
 */
setInterval(JWTManager._cleanUp, 7000);
```

Our backend must ensure that it checks JWTs with the JWTManager on each API call. This becomes another part of the claims validation process.

```
router.get('/todo', function(req, res, next) {
  const jwt = _parseJWT(req);
  if (!JWTManager.isValid(jwt)) {
    res.sendStatus(401);
    return;
  }

  // ...
});
```

You'll need to configure the webhook to push the refresh token revocation events to every API. In our case, it's just the Todo API for now.

We can now revoke a user's refresh token and FusionAuth will broadcast the event to our webhook. The webhook then updates the JWTManager which will cause JWTs for that user to be revoked.

This solution works well even in large systems with numerous backends. It requires the use of refresh tokens, an API that allows

refresh tokens to be revoked, and webhooks to broadcast that revocation. The only caveat is to be sure that your JWTManager code cleans up after itself to avoid running out memory.

Conclusion

To answer the question that started this chapter, you can "sorta" revoke JWTs.

Because they are self-contained, you need to build additional functionality to ensure that the 5 to 10 minute window can be effectively shut. This is one of the tradeoffs of using JWTs for authorization purposes.

Anatomy of a JWT

In this chapter, you'll learn more about what goes into a JSON Web Token and how they are constructed. Here's an example JWT, freshly minted:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6ImY1ODg5MGQxO\
SJ9.eyJhdWQiOiI4NWEwMzg2Ny1kY2NmLTQ4ODItYWRkZS0xYTc5YWVlY\
zUwZGYiLCJleHAiOjE2NDQ4ODQxODUsImlhdCI6MTY0NDg4MDU4NSwiaX\
NzIjoiYWNtZS5jb20iLCJzdWIiOiIwMDAwMDAwMC0wMDAwLTAwMDAtMDA\
wMC0wMDAwMDAwMDEiLCJqdGkiOiIzZGQ2NDM0ZC03OWE5LTRkMTUt\
OThiNS03YjUxZGJiMmNkMzEiLCJhdXRoZW50aWNhdGlvblR5cGUiOiJQQ\
VNTV09SRCIsImVtYWlsIjoiYWRtaW5AZnVzaW9uYXV0aC5pbyIsImVtYW\
lsX3Zlcmlma WVkIjp0cnVlLCJhcHBsaWNhdGlvbklkIjoiODVhMDM4Njc\
tZGNjZi00ODgyLWFkZGUtMWE3OWFlZWM1MGRmIiwicm9sZXMiOlsiY2Vv\
Il19.dee-Ke6RzR0G9avaLNRZf1GUCDfe8Zbk9L2c7yaqKME
```

This may look like a lot of gibberish, but as you learn more about JWTs, it begins to make more sense.

There are a few types of JWTs, but I'll focus on signed JWTs as they are the most common. A signed JWT may also be called a JWS. It has three parts, separated by periods.

There's a header, which in the case of the JWT above, starts with `eyJhbGc`. Then there is a body or payload, which above starts with `eyJhdWQ`. Finally, there is a signature, which starts with `dee-K` in the example JWT.

Let's break this example JWT apart and dig a bit deeper.

The header

`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6ImY1ODg5MGQxOSJ9` is the header of this JWT. The header contains metadata about a token, including the key identifier, what algorithm was used to sign in and other information.

If you run the above header through a base64 decoder:

```
echo 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6ImY1ODg\
5MGQxOSJ9'|base64 -d
```

You will see this JSON:

```
{"alg":"HS256","typ":"JWT","kid":"f58890d19"}%
```

`HS256` indicates that the JWT was signed with a symmetric algorithm, specifically HMAC using SHA-256.

The list of algorithms and implementation support level is below.

"alg" Param Value	Digital Signature or MAC Algorithm	Implementation Requirements
HS256	HMAC using SHA-256	Required
HS384	HMAC using SHA-384	Optional
HS512	HMAC using SHA-512	Optional
RS256	RSASSA-PKCS1-v1_5 using SHA-256	Recommended
RS384	RSASSA-PKCS1-v1_5 using SHA-384	Optional

"alg" Param Value	Digital Signature or MAC Algorithm	Implementation Requirements
RS512	RSASSA-PKCS1-v1_5 using SHA-512	Optional
ES256	ECDSA using P-256 and SHA-256	Recommended+
ES384	ECDSA using P-384 and SHA-384	Optional
ES512	ECDSA using P-521 and SHA-512	Optional
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256	Optional
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384	Optional
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512	Optional
none	No digital signature or MAC performed	Optional

This table is drawn from RFC 7518. As only HS256 is required to be compliant with the spec, consult the software or library used to create JWTs for details on supported algorithms.

Other metadata is also stored in this part of the token. The `typ` header indicates the type of the JWT. In this case, the value is `JWT`, but other values are valid. For instance, if the JWT conforms to RFC 9068, it may have the value `at+JWT` indicating it is an access token.

The `kid` value indicates what key was used to sign the JWT. For a symmetric key the `kid` could be used to look up a value in a secrets vault. For an asymmetric signing algorithm, this value lets the consumer of a JWT look up the correct public key corresponding to the private key which signed this JWT. Processing this value correctly is critical to signature verification and the integrity of the JWT payload.

Typically, you'll offload most of the processing of header values to a library. There are plenty of good open source JWT processing libraries. You should understand the values, but probably won't have to implement the actual processing.

The body

The payload, or body, is where things get interesting. This section contains the data that this JWT was created to transport. If the JWT, for instance, represents a user authorized to access certain data or functionality, the payload contains user data such as roles or other authorization info.

Here's the payload from the example JWT:

```
1 eyJhdWQiOiI4NWEwMzg2Ny1kY2NmLTQ4ODItYWRkZS0xYTc5YWVlYzUwZ\
2 GYiLCJleHAiOjE2NDQ4ODQxODUsImlhdCI6MTY0NDg4MDU4NSwiaXNzIj\
3 oiYWNtZS5jb20iLCJzdWIiOiIwMDAwMDAwMC0wMDAwLTAwMDAtMDAwMC0\
4 wMDAwMDAwMDAwMDEiLCJqdGkiOiIzZGQ2NDM0ZC03OWE5LTRkMTUtOThi\
5 NS03YjUxZGJiMmNkMzEiLCJhdXRoZW50aWNhdGlvblR5cGUiOiJQQVNTV\
6 09SRCIsImVtYWlsIjoiYWRtaW5AZnVzaW9uYXV0aC5pbyIsImVtYWlsX3\
7 ZlcmlmaWVkIjp0cnVlLCJhcHBsaWNhdGlvbklkIjoiODVhMDM4NjctZGN\
8 jZi00ODgyLWFkZGUtMWE3OWFlZWM1MGRmIiwicm9sZXMiOlsiY2VvIl19
```

If you run the sample payload through a base64 decoder:

```
echo 'eyJhdWQiOiI4NWEwMzg2Ny1kY2NmLTQ4ODItYWRkZS0xYTc5YWV\
lYzUwZGYiLCJleHAiOjE2NDQ4ODQxODUsImlhdCI6MTY0NDg4MDU4NSwi\
aXNzIjoiYWNtZS5jb20iLCJzdWIiOiIwMDAwMDAwMC0wMDAwLTAwMDAtM\
DAwMC0wMDAwMDAwMDEiLCJqdGkiOiIzZGQ2NDM0ZC03OWE5LTRkMT\
UtOThiNS03YjUxZGJiMmNkMzEiLCJhdXRoZW50aWNhdGlvblR5cGUiOiJ\
QQVNTV09SRCIsImVtYWlsIjoiYWRtaW5AZnVzaW9uYXV0aC5pbyIsImVt\
YWlsX3ZlcmlmaWVkIjp0cnVlLCJhcHBsaWNhdGlvbklkIjoiODVhMDM4N\
jctZGNjZi00ODgyLWFkZGUtMWE3OWFlZWM1MGRmIiwicm9sZXMiOlsiY2\
VvIl19' |base64 -d
```

You'll see this JSON:

```
{
  "aud": "85a03867-dccf-4882-adde-1a79aeec50df",
  "exp": 1644884185,
  "iat": 1644880585,
  "iss": "acme.com",
  "sub": "00000000-0000-0000-0000-000000000001",
  "jti": "3dd6434d-79a9-4d15-98b5-7b51dbb2cd31",
  "authenticationType": "PASSWORD",
  "email": "admin@fusionauth.io",
  "email_verified": true,
  "applicationId": "85a03867-dccf-4882-adde-1a79aeec50df",
  "roles": [
    "ceo"
  ]
}
```

Note that the algorithm to create signed JWTs can remove base64 padding, so there may be missing = signs at the end of the JWT. You may need to add that back in order to decode a JWT. This depends on the length of the content. You can learn more about it here[22].

As mentioned above, the payload is what your application cares

[22] https://datatracker.ietf.org/doc/html/rfc7515#appendix-C

about, so let's take a look at this JSON more closely. Each of the keys of the object are called "claims".

Some claims are well known with meanings dictated by standards bodies such as the IETF. You can view examples of such claims here[23]. These include the `iss` and `aud` claims from the example token. Both of these have defined meanings when present in the payload of a JWT.

There are other non-standard claims, such as `authenticationType`. These claims may represent business domain or custom data. For example, `authenticationType` is a proprietary claim used by FusionAuth to indicate the method of authentication, such as password, refresh token or via a passwordless link.

You may add any claims you want to a JWT, including data useful to downstream consumers of the JWT. As you can see from the `roles` claim, claims don't have to be simple JSON primitives. They can be any data structure which can be represented in JSON.

Claims to verify

When code is presented with a JWT, it should verify certain claims. At a minimum, these claims should be checked out:

- `iss` identifies the issuer of the JWT. It doesn't matter exactly what this string is (UUID, domain name, URL or something else) as long as the issuer and consumer of the JWT agree on valid values, and that the consumer validates the claim matches a known good value.
- `aud` identifies the audience of the token, that is, who should be consuming it. `aud` may be a scalar or an array value. Again, the issuer and the consumer of the JWT should agree on the specific values considered acceptable.

[23] https://www.iana.org/assignments/jwt/jwt.xhtml

- `nbf` and `exp`. These claims determine the timeframe for which the token is valid. The `nbf` claim can be useful if you are issuing a token for future use. The `exp` claim, a time beyond which the JWT is no longer valid, should always be set. Unlike other claims, these have a defined value format: seconds since the unix epoch.

In addition to these, verify business domain specific claims. For instance, someone consuming the above JWT could deny access when `authenticationType` is an unknown value.

Avoid putting unused claims into a JWT. While there is no limit to the size of a JWT, in general the larger they are, the more CPU is required to sign and verify them and the more time it takes to transport them. Benchmark expected JWTs to have an understanding of the performance characteristics.

Claims and security

The claims of a signed JWT are visible to anyone who possesses the token.

As you saw above, all you need to view the claims in plaintext is a base64 decoder, which is available at every command line and everywhere in the internet.

Therefore, you shouldn't put anything that should remain secret into a JWT. This includes:

- private information such as government Ids
- secrets like passwords
- anything that would leak information like an integer id

Another security concern is related to the verification of the `aud` claim. Since consuming code already possesses the token, isn't verifying the `aud` claim extra work? The `aud` claim indicates who

should receive this JWT, but the code already has it. Nope, always verify this claim.

Why?

Imagine a scenario where you have two different APIs. One is to create and manage todos and the other is a billing API, used to transfer money. Both APIs expect some users with a role of `admin`. However, that role means vastly different things in terms of what actions can be taken.

If both the todo and billing APIs don't verify that any given JWT was created for them, an attacker could take a JWT from the todo API with the `admin` role and present it to the billing API.

This would be at best a bug and at worst an escalation of privilege with negative ramifications for bank accounts.

Signature

The signature of a JWT is critical, because it guarantees the integrity of the payload and the header. Verifying the signature must be the first step that any consumer of a JWT performs. If the signature doesn't match, no further processing should take place.

While you can read the relevant portion of the specification[24] to learn how the signature is generated, the high level overview is:

- the header is turned into a base64 URL encoded string
- the payload is turned into a base64 URL encoded string
- they are concatenated with a `.`
- the resulting string is run through the cryptographic algorithm selected, along with the corresponding key
- the signature is base64 URL encoded

[24] https://datatracker.ietf.org/doc/html/rfc7515#page-15

- the encoded signature is appended to the string with a `.` as a separator

When the JWT is received, the same operations can be performed. If the generated signature is correct, the contents of the JWT are unchanged from when it was created.

Limits

In the specifications, there are no hard limits on length of JWTs. In practical terms, think about:

- Where are you going to store the JWT
- What is the performance penalty of large JWTs

Storage

JWTs can be sent in HTTP headers, stored in cookies, and placed in form parameters. In these scenarios, the storage dictates the maximum JWT length.

For example, the typical storage limit for cookies in a browser is typically 4096 bytes, including the name. The limit on HTTP headers varies widely based on software components, but 8192 bytes seems to be a common value.

Consult the relevant specifications or other resources for limits in your particular use case, but rest assured that JWTs have no intrinsic size limits.

Performance penalty

Since JWTs can contain many different kinds of user information, developers may be tempted to put too much in them. This can

degrade performance, both in the signing and verification steps as well as in transport.

For an example of the former, here are the results of a benchmark from signing and verifying two different JWTs. Each operation was done 50,000 times.

This first JWT had a body approximately 180 characters in length; the total encoded token length was between 300 and 600, depending on the signing algorithm used.

```
hmac sign
  1.632396   0.011794   1.644190 (  1.656177)
hmac verify
  2.452983   0.015723   2.468706 (  2.487930)
rsa sign
 28.409793   0.117695  28.527488 ( 28.697615)
rsa verify
  3.086154   0.011869   3.098023 (  3.109780)
ecc sign
  4.248960   0.017153   4.266113 (  4.285231)
ecc verify
  7.057758   0.027116   7.084874 (  7.113594)
```

The next JWT payload was of approximately 1800 characters, so ten times the size of the previous token. This had a total token length of 2400 to 2700 characters.

```
hmac sign
  3.356960   0.018175   3.375135 (  3.389963)
hmac verify
  4.283810   0.018320   4.302130 (  4.321095)
rsa sign
 32.703723   0.172346  32.876069 ( 33.072665)
rsa verify
  5.300321   0.027455   5.327776 (  5.358079)
ecc sign
  6.557596   0.032239   6.589835 (  6.624320)
ecc verify
  9.184033   0.035617   9.219650 (  9.259225)
```

You can see that the total time increased for the longer JWT, but typically not linearly. The increase in time taken ranges from about 20% for RSA signing to approximately 100% for HMAC signing.

Be mindful of additional time taken to transport longer JWT; this can be tested and optimized in the same way you would with any other API or HTML content.

Conclusion

Signed JWTs have a header, body, and signature.

Understanding all three of these components are critical to the correct use of JWTs.

Conclusion

This book has covered a fair bit of ground on the topic of JSON Web Tokens (JWTs). I hope you enjoyed learning more about this powerful token format and have a better understanding of this technical topic. At the end of the day, JWTs are another tool, and the goal is not to use them everywhere, but to feel more comfortable about when JWTs make sense and when they do not.

You also learned about revocation solutions and the implementation tradeoffs available for this vital function.

As a widely supported, understandable, flexible way to transport information that can be cryptographically verified, JWTs deserve a place in your developer toolbelt. I hope this book helped you decide that they've earned it.