5 Easy Steps to Understanding JSON Web Tokens (JWT)

Mikey Stecky-Efantis
May 16, 2016 · 7 min read

In this article, the fundamentals of what JSON Web Tokens (JWT) are, and why they are used will be explained. JWT are an important piece in ensuring trust and security in your application. JWT allow claims, such as user data, to be represented in a secure manner.

To explain how JWT work, let's begin with an abstract definition.

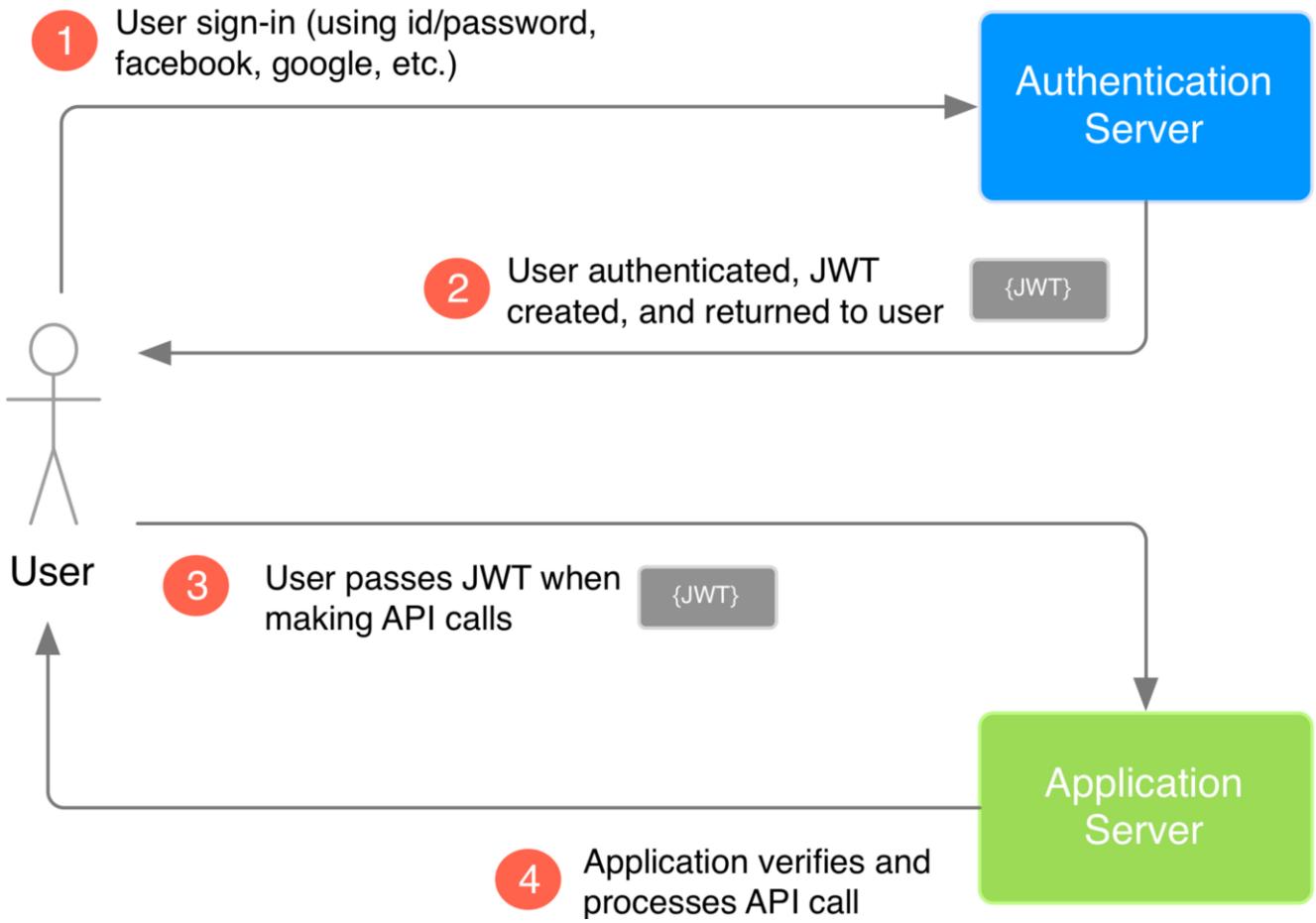
> A JSON Web Token (JWT) is a JSON object that is defined in RFC 7519 as a safe way to represent a set of information between two parties. The token is composed of a header, a payload, and a signature.

Simply put, a JWT is just a string with the following format:

```
header.payload.signature
```

It should be noted that a double quoted string is actually considered a valid JSON object.

To show how and why JWT are actually used, we will use a simple 3 entity example (see the below diagram). The entities in this example are the user, the application server, and the authentication server. The authentication server will provide the JWT to the user. With the JWT, the user can then safely communicate with the application.



How an application uses JWT to verify the authenticity of a user.

In this example, the user first signs into the authentication server using the authentication server's login system (e.g. username and password, Facebook login, Google login, etc). The authentication server then creates the JWT and sends it to the user. When the user makes API calls to the application, the user passes the JWT along with the API call. In this setup, the application server would be configured to verify that the incoming JWT are created by the authentication server (the verification process

will be explained in more detail later). So, when the user makes API calls with the attached JWT, the application can use the JWT to verify that the API call is coming from an authenticated user.

Now, the JWT itself, and how it's constructed and verified, will be examined in more depth.

Step 1. Create the HEADER

The header component of the JWT contains information about how the JWT signature should be computed. The header is a JSON object in the following format:

```
{
    "typ": "JWT",
    "alg": "HS256"
}
```

header.json hosted with ❤ by GitHub view raw

In this JSON, the value of the "typ" key specifies that the object is a JWT, and the value of the "alg" key specifies which hashing algorithm is being used to create the JWT signature component. In our example, we're using the HMAC-SHA256 algorithm, a hashing algorithm that uses a secret key, to compute the signature (discussed in more detail in step 3).

Step 2. Create the PAYLOAD

The payload component of the JWT is the data that's stored inside the JWT (this data is also referred to as the "claims" of the JWT). In our example, the authentication server creates a JWT with the user information stored inside of it, specifically the user ID.

```
{
    "userId": "b08f86af-35da-48f2-8fab-cef3904660bd"
}
```

payload.json hosted with ❤ by GitHub view raw

The data inside the payload is referred to as the "claims" of the token.

In our example, we are only putting one claim into the payload. You can put as many claims as you like. There are several different standard claims for the JWT payload, such as "iss" the issuer, "sub" the subject, and "exp" the expiration time. These fields

can be useful when creating JWT, but they are optional. See the wikipedia page on JWT for a more detailed list of JWT standard fields.

Keep in mind that the size of the data will affect the overall size of the JWT, this generally isn't an issue but having excessively large JWT may negatively affect performance and cause latency.

Step 3. Create the SIGNATURE

The signature is computed using the following pseudo code:

```
// signature algorithm

data = base64urlEncode( header ) + "." + base64urlEncode( payload )

hashedData = hash( data, secret )

signature = base64urlEncode( hashedData )
```

What this algorithm does is base64url encodes the header and the payload created in steps 1 and 2. The algorithm then joins the resulting encoded strings together with a period (.) in between them. In our pseudo code, this joined string is assigned to *data*. The *data* string is hashed with the secret key using the hashing algorithm specified in the JWT header. The resulting hashed data is assigned to *hashedData*. This hashed data is then base64url encoded to produce the JWT signature.

In our example, both the header, and the payload are base64url encoded as:

```
// header
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

// payload
eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ
```

Then, applying the specified signature algorithm with the secret key on the period-joined encoded header and encoded payload, we get the hashed data needed for the signature. In our case, this means applying the HS256 algorithm, with the secret key set as the string "secret", on the *data* string to get the *hashedData* string. After, through base64url encoding the *hashedData* string we get the following JWT signature:

```
// signature

-xN_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM
```

Step 4. Put All Three JWT Components Together

Now that we have created all three components, we can create the JWT. Remembering the *header.payload.signature* structure of the JWT, we simply need to combine the components, with periods (.) separating them. We use the base64url encoded versions of the header and of the payload, and the signature we arrived at in step 3.

```
// JWT Token
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhmODZhZi0zNWR
hLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN_h82PHVTCMA9vdoHrcZxH-
x5mb11y1537t3rGzcM
```

You can try creating your own JWT through your browser at jwt.io.

Going back to our example, the authentication server can now send this JWT to the user.

How does JWT protect our data?

It is important to understand that the purpose of using JWT is **NOT** to hide or obscure data in any way. The reason why JWT are used is to prove that the sent data was actually created by an authentic source.

As demonstrated in the previous steps, the data inside a JWT is **encoded** and **signed**, not **encrypted**. The purpose of encoding data is to transform the data's structure. Signing data allows the data receiver to verify the authenticity of the source of the data. So encoding and signing data does NOT secure the data. On the other hand, the main purpose of encryption is to secure the data and to prevent unauthorized access. For a more detailed explanation of the differences between encoding and encryption, and also for more information on how hashing works, see this article.

> Since JWT are signed and encoded only, and since JWT are not encrypted, JWT do not guarantee any

security for sensitive data.

Step 5. Verifying the JWT

In our simple 3 entity example, we are using a JWT that is signed by the HS256 algorithm where only the authentication server and the application server know the secret key. The application server receives the secret key from the authentication server when the application sets up its authentication process. Since the application knows the secret key, when the user makes a JWT-attached API call to the application, the application can perform the same signature algorithm as in Step 3 on the JWT. The application can then verify that the signature obtained from it's own hashing operation matches the signature on the JWT itself (i.e. it matches the JWT signature created by the authentication server). If the signatures match, then that means the JWT is valid which indicates that the API call is coming from an authentic source. Otherwise, if the signatures don't match, then it means that the received JWT is invalid, which may be an indicator of a potential attack on the application. So by verifying the JWT, the application adds a layer of trust between itself and the user.

In Conclusion

We went over what JWT are, how they are created and validated, and how they can be used to ensure trust between an application and its users. This is a starting point for understanding the fundamentals of JWT and why they are useful. JWT are just one piece of the puzzle in ensuring trust and security in your application.

. . .

It should be noted that the JWT authentication setup described in this article is using a symmetric key algorithm (HS256). You can also set up your JWT authentication in a similar way except using an asymmetric algorithm (such as RS256) where the authentication server has a secret key, and the application server has a public key. Check out this Stack Overflow question for a detailed breakdown of the differences between using symmetric and asymmetric algorithms.

It should also be noted that JWT should be sent over HTTPS connections (not HTTP). Having HTTPS helps prevents unauthorized users from stealing the sent JWT by making it so that the communication between the servers and the user cannot be intercepted .

Also, having an expiration in your JWT payload, a short one in particular, is important so that if old JWT ever get compromised, they will be considered invalid and can no longer be used.

. . .

If you enjoyed this article and are writing handlers for AWS Lambda and are implementing JWT, please check out our project: Vandium

Vandium: The Node.js Framework for AWS Lamba

One of the most exciting new cloud technologies over the last few years has been the emergence of Amazon Web Services'...

medium.com

Thanks to Richard Hyatt.

JavaScript Security Cloud Computing