**JSON Web Token**

*Computer Security*

*Group 4*

Name: Sreerag Marar Name: Jigar Dalwadi

Id: 01660933 Id: 01684802

**Index**

1. *Abstract*
2. *Introduction*
3. *JWT Description*
4. *Implementation*
5. *Pros and Cons*
6. *Updated Implementation*
7. *Conclusion*

**Abstract**

JSON Web Token (JWT, pronounced  is a [JSON](https://en.wikipedia.org/wiki/JSON)-based [open standard](https://en.wikipedia.org/wiki/Open_standard)  for creating [access tokens](https://en.wikipedia.org/wiki/Access_token) that assert some number of claims. For example, a server could generate a token that has the claim "logged in as admin" and provide that to a client. The client could then use that token to prove that it is logged in as admin. The tokens are signed by the server's key, so the client is able to verify that the token is legitimate. The tokens are designed to be compact, [URL](https://en.wikipedia.org/wiki/URL)-safe and usable especially in [web browser](https://en.wikipedia.org/wiki/Web_browser) [single sign-on](https://en.wikipedia.org/wiki/Single_sign-on) (SSO) context. JWT claims can be typically used to pass identity of authenticated users between an identity provider and a [service provider](https://en.wikipedia.org/wiki/Service_provider), or any other type of claims as required by business processes.

The tokens can also be [authenticated](https://en.wikipedia.org/wiki/Authentication) and encrypted. The API model has been used a great amount recently in applications. This has come about because applications can’t just rely on their own data anymore, for a project to fully see its potential, it must be able to have third-party applications, intermingle with other applications, and have its data easily accessible by developers. Think of how Facebook provides an API to grab its data (as long as you are authenticated of course). Facebook also allows third-party applications and other services to access its data. This is all done through an API. Now when we talk about building our own APIs, there’s always going to be the topic of how to secure our own API. We’ve talked a bit on [token based authentication](https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication), and built our own.Today we’ll be looking at a standard (JSON Web Tokens) and how to create them.

**Introduction**

JSON Web Token (JWT) is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA**.

Let's explain some concepts of this definition further.

**Compact**: Because of their smaller size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header. Additionally, the smaller size means transmission is fast.

**Self-contained**: The payload contains all the required information about the user, avoiding the need to query the database more than once. They will carry all the information necessary within itself. This means that a JWT will be able to transmit basic information about itself, a payload (usually user information), and a signature.

**Compatibility**: JWTs work in .NET, Python, Node.js, Java, PHP, Ruby, Go, JavaScript, and Haskell. So you can JWTs work in .NET, Python, Node.js, Java, PHP, Ruby, Go, JavaScript, and Haskell. So you can see that these can be used in many different scenarios

**JWTs can be passed around easily**: Since JWTs are self-contained, they are perfectly used inside an HTTP header when authenticating an API. You can also pass it through the URL.

**When should you use JSON Web Tokens?**

Here are some scenarios where JSON Web Tokens are useful:

**Authentication**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token.

**Single Sign On** is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

**Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties, because as they can be signed, for example using public/private key pairs, you can be sure that the senders are who they say they are.

Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

**What does a JWT look like?**

A JWT is easy to identify. It is three strings separated by .

For example:

Let’s break down the **3 parts** and see what each contains.

[#Breaking Down a JSON Web Token](https://scotch.io/tutorials/the-anatomy-of-a-json-web-token#breaking-down-a-json-web-token)

Since there are 3 parts separated by a ., each section is created differently. We have the 3 parts which are:

* header
* payload
* signature

**#Header**

The header carries 2 parts:

declaring the type, which is JWT

the hashing algorithm to use (HMAC SHA256 in this case)

Here’s an example:

{

"typ": "JWT",

"alg": "HS256"

}

Now once this is base64encode, we have the first part of our JSON web token!

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

**#Payload**

The payload will carry the bulk of our JWT, also called the [JWT Claims](http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html#RegisteredClaimName). This is where we will put the information that we want to transmit and other information about our token.

There are multiple claims that we can provide. This includes registered claim names, public claim names, and private claim names.

**REGISTERED CLAIMS**

Claims that are not mandatory whose names are reserved for us. These include:

iss: The issuer of the token

sub: The subject of the token

aud: The audience of the token

exp: This will probably be the registered claim most often used. This will define the expiration in NumericDate value. The expiration MUST be after the current date/time.

nbf: Defines the time before which the JWT MUST NOT be accepted for processing

iat: The time the JWT was issued. Can be used to determine the age of the JWT

jti: Unique identifier for the JWT. Can be used to prevent the JWT from being replayed. This is helpful for a one time use token.

**PUBLIC CLAIMS**

These are the claims that we create ourselves like user name, information, and other important information.

**PRIVATE CLAIMS**

A producer and consumer may agree to use claim names that are private. These are subject to collision, so use them with caution.

Example **Payload**

Our example payload has two registered claims (iss, and exp) and two public claims (name, admin).

{

"iss": "scotch.io",

"exp": 1300819380,

"name": "Peter ",

"admin": true

}

**This will encode to**:

eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOjEzMDA4MTkzODAsIm5hbWUiOiJDaHJpcyBTZXZpbGxlamEiLCJhZG1pbiI6dHJ1ZX0

That will be the second part of our JSON Web Token.

**#Signature**

The third and final part of our JSON Web Token is going to be the signature. This signature is made up of a hash of the following components:

**the header**

**the payload**

**secret**

This is how we get the third part of the JWT:

The secret is the signature held by the server. This is the way that our server will be able to verify existing tokens and sign new ones.

This gives us the final part of our JWT.

03f329983b86f7d9a9f5fef85305880101d5e302afafa20154d094b229f75773

Now we have our full JSON Web Token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOjEzMDA4MTkzODAsIm5hbWUiOiJDaHJpcyBTZXZpbGxlamEiLCJhZG1pbiI6dHJ1ZX0.03f329983b86f7d9a9f5fef85305880101d5e302afafa20154d094b229f75773

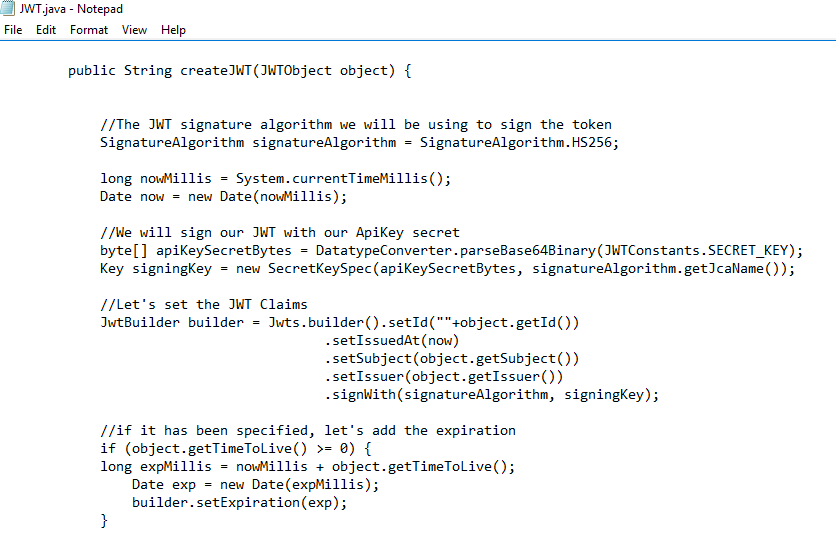
[Auth0](https://auth0.com/) has created a [great site](http://jwt.io/) to go through and test out how JWTs are made. You can see as you change the content on the fly, you are able to see the JWT get updated immediately. Auth0 provides great tools and they also maintain the [jsonwebtoken](https://github.com/auth0/node-jsonwebtoken) Node package to handle creating and verifying JWTs in Node.

**Implementation**

We have implemented JWT 2 ways, one is the traditional one which is using a secret key configured at the server, and the updated one

We have used java JWT api to show both the implementation.

Implementation 1(Existing)



**Explanation:**

We have initially configured a secret key in the server (in the form of a constant defined or in the database based on the program)

Once, we initialize It, we hash base 64 encode the key and then convert it into the signed key using the API

Now, using the signed key and the header and payload, we send the build data to the server.

They key is the reusable key in the front end.

The expiry of the key can be configured at the user end based on requirement.

The key is passed to the user who every time invokes the service in future for authentication using this key.

**Pros**

**No Database Table** : This implies fewer DB queries, which implies faster response time.

**Simpler to use if careful**: If your architecture doesn’t user client Sessions and your security basics are clear, the development time in case of JWT is faster using the existing libraries.

**Used across services :** You can have one authorization server that deals with the Login/Registration and generates the token, all the subsequent requests will need not have to go to the authorization server as the only the Auth-server will have the private key, and rest of the severs will have the public-key to verify the signature.   
This is really useful in case of corporate systems where in the authorization server is in a secure environment. e.g. a user needs to be connected to the intranet to login but once done, the public servers can verify and proceed on.

**Cons**

**Compromised Secret Key :** The best and the worst thing about JWT is that it relies on just one Key. Consider that the Key **is leaked by a careless or a rogue developer/administrator, the whole system is compromised!**   
The attacker(who has access to the Key) can easily access all user data if he has the user-id which can be easily acquired.  
The only way to recover from this point is to generate a new Key(Key-pair) that will be used across systems here on. This would me all the existing client tokens are invalidated and each user would have to login again. Image one day 100% of Facebook users will be logged out.   
Well you might wonder, why is the same not possible if the developer/administrator leaks the Session table?   
It is possible, but it is related to the practicality of the situation. Remember, most of the online breaches are done with social engineering than complicated technical hacks.

·         Practically it is really difficult to leak the whole table. In case of a single key, the admin just has pretend to take a photo of his friend in the office aaaaand the secret is on Reddit the next morning you fire him.

·         As well consider the OpenSSlHeartBreak bug. It is really easy to extract the secret key from just a couple of memory dumps with a simple string match script.

**Cannot manage client from the server:**As well consider the case that a user’s mobile is stolen, and he wants to logout of all existing sessions(e.g. Gmail’s logout other sessions feature). Well it’s not possible in case of JWT. In our case it used to be rogue users. We needed to log them out. Well, in case of of HelpTap it was quite easy as we just had to delete the session tokens. There was no way to do the same in case of Botttr cause we used JWT in that case. 

**Cannot push Messages to clients** (Identifying clients from server)**:** As we have no record about the logged-in clients on the DB end, we cannot push messages to all the clients. Each client has an AWS SQS queue to itself where we push any new messages. In case of JWT this would not have been possible as identifying each client per user is not possible.

**Crypto-algo can be deprecated**: JWT relies completely on the Signing algorithm. Now, though it is not frequent, but in the past many Encryption/Signing algorithms have been deprecated.   
This article shows how you can crack the Wifi password of a WEP Encrypted Wifi which was the most common type of encryption not more than a year ago. The hack was based on the weakness of the crypto algorithm. So, in case of JWT, if such a thing happens, yet again, every user on the platform will have to login again.Yet again one will have to wait till all the JWT libraries update with the latest crypto-algo.

**Data Overhead:** The size of the JWT token will be more than that of a normal Session token. The more data you add in the JWT token, the longer it gets linearly. Remember, each request needs the token in it for request verification. So say, a 1 KB JWT token implies each request will have 1KB over-head upload which is really bad in cases of low speed net connectivity.   
In case of bad developer, someone might put more data in the JSON and that would increase the length. The length of the sessions tokens can be as small it can be and still be secure. e.g. the possible combinations for just a 5 letter alphanumeric session string is almost 1 billion combinations (62⁵)

**Complicated to understand:** JWT uses cryptographic Signature algorithms to verify the data and get the user-id from the token. Understanding the Signing Algo in itself requires basics of cryptography. So, in case if the developer is not completely educated s/he might introduce security loopholes in the system. My co-worker was surprised when I decoded the JWT token without using the secret key. He expected that the whole token was an encrypted one.

**Implementation 2**



In the updated version, we do some minor changes which can reduce the vulnerability of the JWT.

We used the basic Java JWT implementation in the server end, except for key configuration.

Instead of having a centralized key for the entire application, we make a dynamic key based on the issuer.

The advantage of this being is that the key is specific to the user.

Above that, we further encrypt the key using any encryption mechanism. In our case, we have used SHA-256 to do it.

So, the new key to be encoded would be in the form of SHA hash value (server secret constant key + dynamic issuer key + initialized time)

The login time is used to add more complexity to the algorithm.

Using the login time, the expiry can even be done at the front end. Thereby giving a two-way key expiry mechanism.

Once, the SHA hashed key is produced, we implement the remaining implementation as before.

Once the key is created, we pass the key as well the initialize time to the user. So that the user can pass the key and initialize time to the user to reauthenticate

**Conclusion**

The JSON Web Token standard can be used across multiple languages and is quickly and easily interchangeable.

You can use the token in a URL, POST parameter, or an HTTP header. The versatility of the JSON Web Token lets us authenticate an API quickly and easily by passing information through the token.

Hence, we can infer that JSON Web token is a simple, fast and easily compatible API which can be used to provide security to applications. It can be further enhanced and improved for tighter security with improvisation.

**Bibliography**

* [**https://en.wikipedia.org/wiki/JWT**](https://en.wikipedia.org/wiki/JWT)
* [**https://en.wikipedia.org/wiki/JSON\_Web\_Token#Structure**](https://en.wikipedia.org/wiki/JSON_Web_Token)
* [**https://jwt.io/introduction/**](https://jwt.io/introduction/)
* [**https://scotch.io/tutorials/the-anatomy-of-a-json-web-token**](https://scotch.io/tutorials/the-anatomy-of-a-json-web-token)
* [**https://medium.com/@rahulgolwalkar/pros-and-cons-in-using-jwt-json-web-tokens-196ac6d41fb4**](https://medium.com/@rahulgolwalkar/pros-and-cons-in-using-jwt-json-web-tokens-196ac6d41fb4)
* [**https://www.ietf.org/archive/id/draft-ietf-oauth-json-web-token-13.pdf**](https://www.ietf.org/archive/id/draft-ietf-oauth-json-web-token-13.pdf)