JWT Details

JWT Details
Session-based Authentication & Token-based Authentication

For using any website, mobile app or desktop app , you need to create an account, then use it to login for accessing features of the app : Authentication.

So, how to authenticate a user?
Simple method used : Session-based Authentication.
(refer : session-based-authentication.png)

As per the diag , when a user logs into a website, the Server will create a new Session for that user and store it (in Memory or Database). Server also returns a SessionId for the Client to save it in Browser Cookie.

The Session on Server has an expiration time. After that time, this Session has expired and the user must re-login to create another Session.

If the user has logged in and the Session has not expired yet, the Cookie (including SessionId) always goes with all HTTP Request to Server. Server will compare this SessionId with stored Session to authenticate and return corresponding Response.

If it's been working fine , then why do we need Token-based Authentication?
The answer is we don’t have the only consumer as a browser (end user : client a person), we may have different consumers of our services here.

So if you have a website which works well with thin client(browser client) & want to implement system for Mobile (Native Apps) and use the same Web app. You can't authenticate users who use Native App using Session-based Authentication because these native apps don’t support cookies. Or if you are implementing a REST server , every REST request HAS TO be stateless , meaning you can't think of maintaining a Http Session in the back end. On the other hand , you can't expect your front end app , to send you the credentials (username / password) along with every request.

That’s why Token-based Authentication was born.

With this method, the user login state is encoded into a JSON Web Token (JWT) by the Server and sent to the Client after successful authentication.

Instead of creating a Session, the Server generates a JWT from user login data and sends it to the Client. The Client saves the JWT and then onwards sends this JWT along with every request , typically in a header to the server. The Server will validate the JWT and return the Response.
(Giving access to the secured resources)

For storing JWT on Client side, it depends on the platform you use:

Browser: Local Storage
IOS: Keychain

Android: SharedPreferences
---

3 important parts of a JWT:

Header
Payload
Signature
---
1. Header
The Header answers the question: How will we calculate JWT?
It’s a JSON object
eg :

```
{
  "typ": "JWT",
  "alg": "HS512"
}
```

– typ is ‘type’, indicates that Token type here is JWT.
– alg stands for ‘algorithm’ which is a hash algorithm for generating Token signature. Here HS256 is HMAC-SHA256/512 – the algorithm which uses Secret Key.
---
2. Payload
The Payload helps us to answer: What do we want to store in JWT?
This is a payload sample:

```
{
  "userId": "abcd123456",
  "username": "rama",
  "email": "rama@gmail.com",
  // standard fields
  "iss": "Issuer at developers.com",
  "iat": 1570238918,
  "exp": 1570238992
}
```

In the JSON object above, we store 3 user fields: userId, username, email. You can save any field you want.

We also have some Standart Fields. They are optional.

iss (Issuer): who issues the JWT
iat (Issued at): time the JWT was issued at
exp (Expiration Time): JWT expiration time
---
3. Signature
This part is where we use the Hash Algorithm : HMAC-SHA256
Look at the code for getting the Signature below:
In Java code :
java.util.Base64 class offers encoding n decoding.
Base64 is a binary-to-text encoding scheme. It represents binary data in a

printable ASCII string format by translating it into a radix-64 representation.

The basic encoder keeps things simple and encodes the input as is, without any line separation.
The output is mapped to a set of characters in A-Za-z0-9+/ character set, and the decoder rejects any character outside of this set.
eg : In Java
String originalInput = "test input";
String encodedString =
Base64.getEncoder().encodeToString(originalInput.getBytes());

In Javascript
const data = Base64UrlEncode(header) + '.' + Base64UrlEncode(payload);
const hashedData = Hash(data, secret);
const signature = Base64UrlEncode(hashedData);

– First, it encodes Header and Payload, join them with a dot .
- Next, makes a hash of the data using Hash algorithm (defined at Header) with a secret string.
– Finally, encodes the hashing result to get Signature.
-
After having Header, Payload, Signature, combines them into JWT standard structure: header.payload.signature.

Does JWT secures our data ?
JWT does NOT secure your data

JWT does not hide, obscure, secure data at all. You can see that the process of generating JWT (Header, Payload, Signature) only encode & hash data, not encrypt data.

The purpose of JWT is to prove that the data is generated by an authentic source.

So, what if there is a Man-in-the-middle attack that can get JWT, then decode user information?
Yes, that is possible, so always make sure that your application has the HTTPS encryption.

How Server validates JWT from Client ?
For creating a JWT , we use a Secret string to create Signature. This Secret string is unique for every Application and must be stored securely in the server side.

When receiving JWT from Client, the Server get the Signature, verify that the Signature is correctly hashed by the same algorithm and Secret string as above. If it matches the Server’s signature, the JWT is valid.