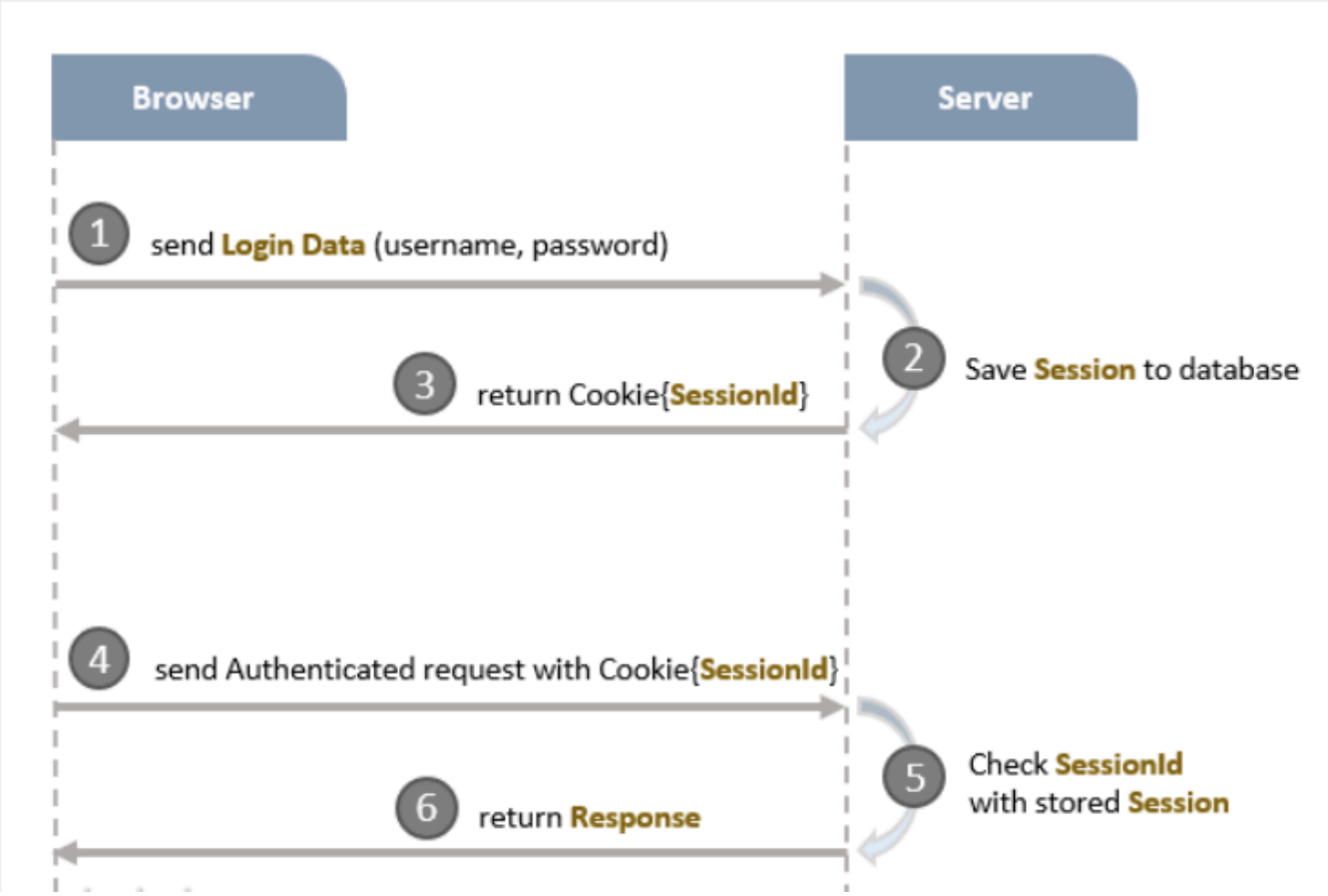
- ALL ABOUT AUTHENTICATION -

->SANSKAR DWIVEDI

Session-Based Authentication:



When a user logs into a website, the Server will generate a `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.

Why Token-based Authentication?

The answer is we don't have only website, there are many platforms over there.

Assume that we have a website which works well with Session. One day, we want to implement system for Mobile (Native Apps) and use the same Database with the current Web app. What should we do? We cannot authenticate users who use Native App using Session-based Authentication because these kinds don't have Cookie.

Should we build another backend project that supports Native Apps?

Or should we write an Authentication module for Native App users?

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 send to the Client. Now-a-days, many RESTful APIs use it.**

#What is JWT?

"JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code(MAC) and/or encrypted."

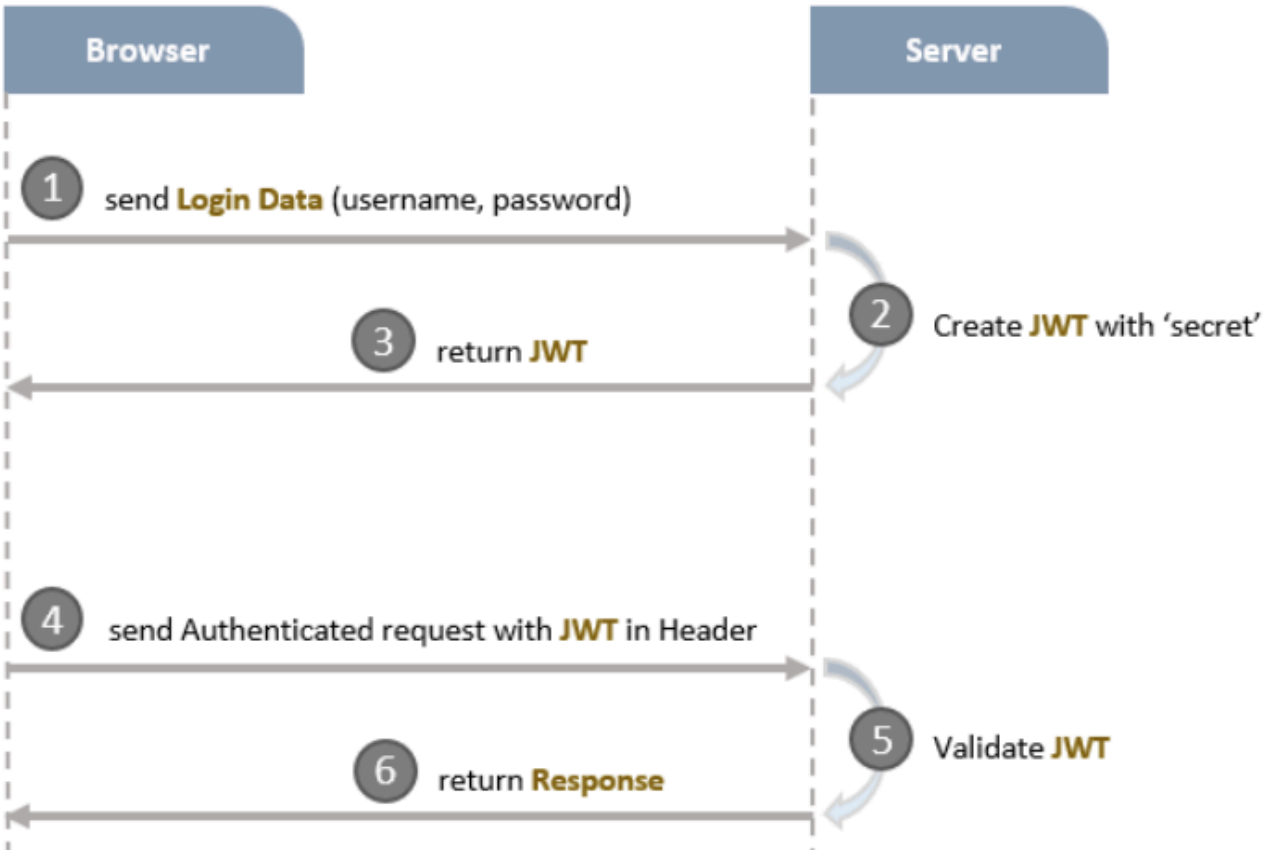
- Open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

"An open standard is a standard that is publicly available and has various rights to use associated with it and may also have various properties of how it was designed. There is no single definition, and interpretations vary with usage."

- - The information can be verified and trusted because it is digitally signed.

- - The client will need to authenticate with the server using the credentials only once. During this time the server validates the credentials and returns the client a JSON Web Token(JWT). For all future requests the client can authenticate itself to the server using this JSON Web Token(JWT) and so does not need to send the credentials like username and password.

Workflow Of JWT



Instead of creating a `Session`, the **Server** here will generate a `JWT` token from user login data and send it to the **Client**. The Client saves the `JWT` and from now on, each & every Request from the Client should be attached that `JWT` (commonly at ***header***). The Server will validate the JWT and return the Response.

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

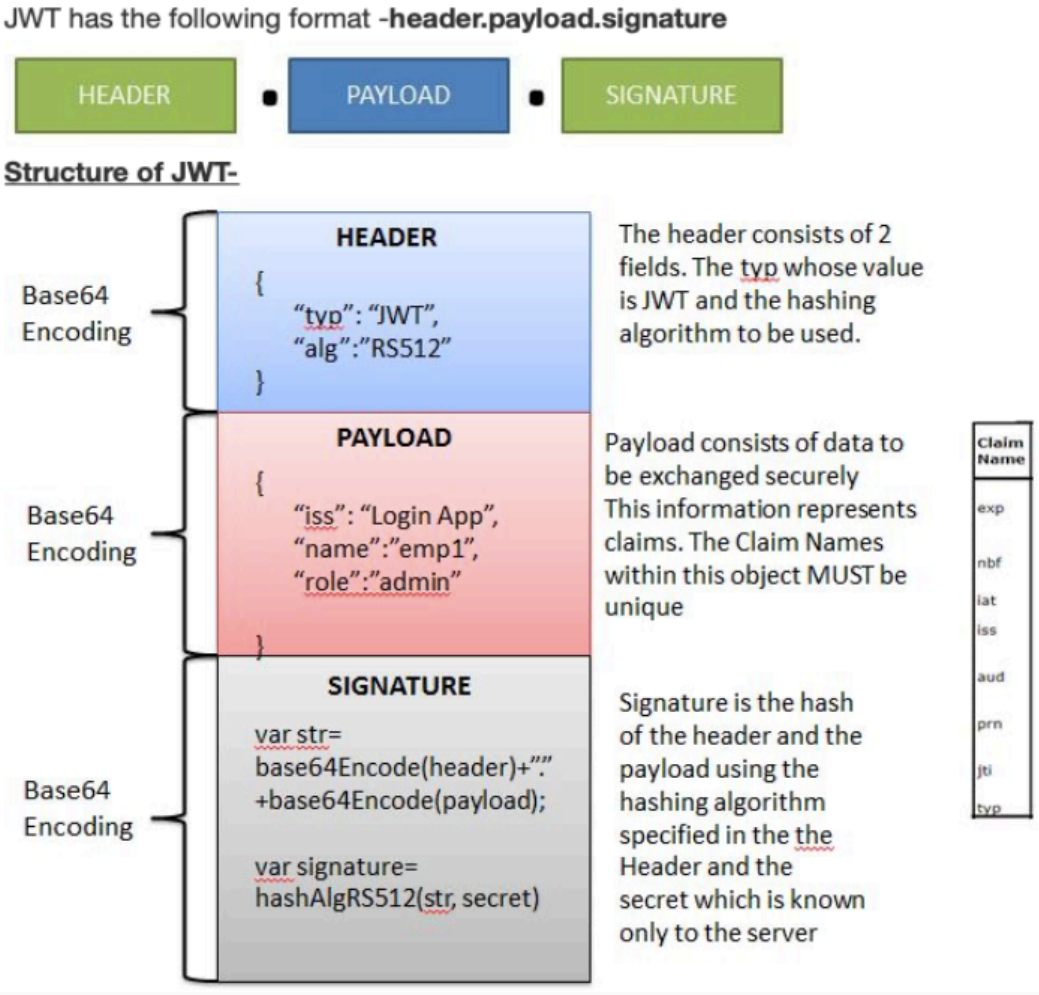
- Browser: Local Storage
- IOS: Keychain
- Android: SharedPreferences

Step-BY-Step Explanation:

1. During the first request, the client sends a POST request with username and password.
2. The server validates the username & password & upon successful authentication the server generates the JWT & sends this JWT to the client. This JWT can contain a ***payload*** of data.

3. On all subsequent requests the client sends this JWT token in the header. Using this token, the server authenticates the user. So we don't need the client to send the user name and password to the server during each request for authentication, but only once after which the server issues a JWT to the client.

"A JWT payload can contain things like user ID so that when the client again sends the JWT, you can be sure that it is issued by you, and you can see to whom it was issued."



Creation Of JWT

There are three important parts of a JWT:

- Header
- Payload
- Signature

- HEADER :

The Header answers the question: How will we calculate JWT?
Now look at an example of `header`, it's a JSON object like this:

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

– `typ` is 'type', indicates that Token type here is JWT.
– `alg` stands for 'algorithm' which is a **hash algorithm** for generating Token `signature`. In the code above, **`HS256`** is HMAC-SHA256 – the algorithm which uses *Secret Key*.

- PAYLOAD :

The Payload helps us to answer: What do we want to store in JWT?
This is a payload sample:

```
{
   "username": "sanskarDwivedi",
  "email": "sanskar.dwivedi@capgemini.com",
  // standard fields
  "iss": "sanskar, author of this document",
  "iat": 1680439948,
  "exp": 1680439992
}
```

In the JSON object above, we store 2 user fields: `username` & `email`. You can save as many fields you want.

We also have some **Standard 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.

- SIGNATURE :

This part is where we use the Hash Algorithm. Look at the code for getting the Signature below:

```
const data = Base64UrlEncode(header) + '.' +
Base64UrlEncode(payload);

const hashedData = Hash(data, secret);

const signature = Base64UrlEncode(hashedData);
```

Let's explain it :-
– First, we encode Header and Payload, join them with a dot `.`

```
data = '[encodedHeader].[encodedPayload]'
```

– Next, we make a hash of the `data` using Hash algorithm (defined at Header) with a `secret` string.
– Finally, we encode the hashing result to get **Signature**.

NOW, Combine all of these 3 :

After having Header, Payload, Signature, we're gonna combine them into JWT standard structure: `header.payload.signature`.

Following code will illustrate how we do it.

```
const encodedHeader = base64urlEncode(header);
/* Result */
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9"

const encodedPayload = base64urlEncode(payload);
/* Result */
"eyJ1c2VySWQiOiJhYmNkMTIzNDVnaGlqayIsInVzZXJuYW1lIjoiYmV6
a29kZXIiLCJlbWFpbCI6ImNvbnRhY3RAYmV6a29kZXIuY29tIn0"
```

```
const data = encodedHeader + "." + encodedPayload;
const hashedData = Hash(data, secret);
const signature = base64urlEncode(hashedData);
/* Result */
"crrCKWNGay10ZYbzNG3e0hfLKbL7ktolT7GqjUMwi3k"
```

```
// header.payload.signature
const JWT = encodedHeader + "." + encodedPayload + "." +
signature;
/* Result */
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJhYmN
kMTIzNDVnaGlqayIsInVzZXJuYW1lIjoiYmV6a29kZXIiLCJlbWFpbCI6
ImNvbnRhY3RAYmV6a29kZXIuY29tIn0.5IN4qmZTS3LEaXCisfJQhrSyh
SPXEgM1ux-qXsGKacQ"
```

NOTE*********************************

Information in the payload of the JWT is visible to everyone. So we should not pass any sensitive information like passwords in the payload. We can encrypt the payload data if we want to make it more secure. However we can be sure that no one can tamper and change the payload information. If this is done the server will recognise it.

****

Does JWT secures our data ?

NO, JWT does not hide, obscure, secure data at all. You can see that the process of generating JWT (Header, Payload, Signature) only encodes & hash data, it does 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 ?

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 ->verifies that the Signature is correctly hashed by the same algorithm and Secret string -> If it matches the Server's signature, the JWT is valid.

!!!!!!Important!!!!!
Experienced programmers can still add or edit **Payload** information when sending it to the server. What should we do in this case?

We store the Token before sending it to the Client. It can ensure that the JWT transmitted later by the Client is valid.

In addition, saving the user's Token on the Server will also benefit the ***Force Logout*** feature from the system.

CONCLUSION :-

There will never be a best method for authentication. It depends on the use case and how you want to implement.

However, for app that you want to scale to a large number of users across many platforms, JWT Authentication is preferred because the Token will be stored on the Client side.