What is JWT?

JWT stands for JSON Web Token. JSON Web Token (JWT) is an open standard (RFC 7519) 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. 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.

What Is a Token-Based Authentication System?

The token-based authentication systems allow users to enter their username and password in order to obtain a token which allows them to fetch a specific resource - without entering their username and password at each request. Once their token has been obtained, the user can use the token to access specific resources for a set time period.

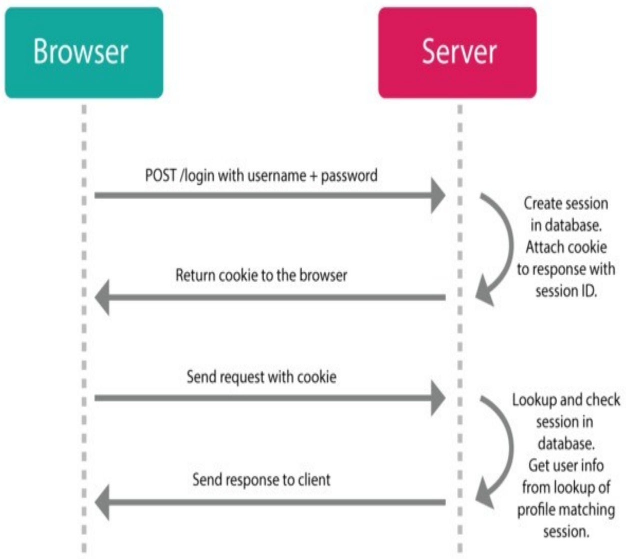
JWT (pronounced 'jot') is a token based authentication system. It is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is digitally signed using JSON Web Signature. The JWT is a self-contained token which has authentication information, expire time information, and other user defined claims digitally signed.

How We Used to Do Authentication

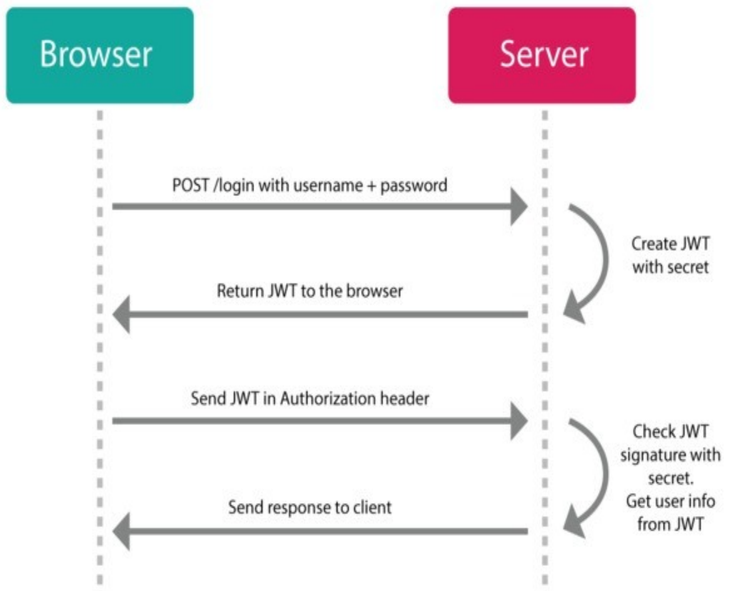
HTTP is a stateless protocol. That means it doesn't store any state from request to response. If you login for one request, you'll be forgotten and will need to login again to make another request. As you can imagine, this can get very annoying, very fast.

The old-school solution was to create what's called a "session." A session is implemented in two parts:

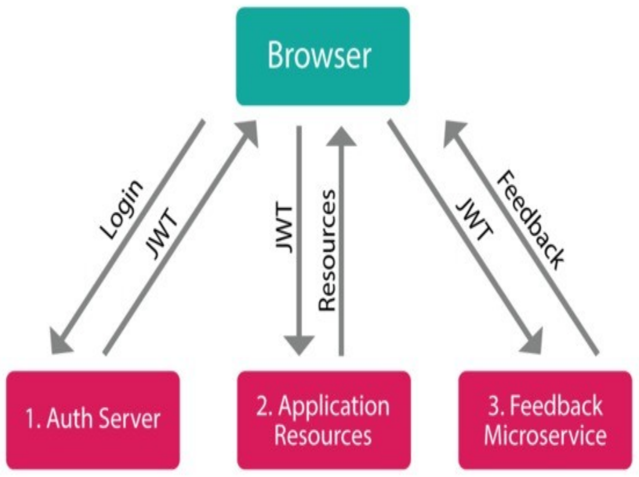
1. An object stored on the server that remembers if a user is still logged in, a reference to their profile, etc.
2. A cookie on the client-side that stores some kind of ID that can be referenced on the server against the session object's ID.Image title


Browser
Server
POST /login with username + password
Create session in database. Attach cookie to response with session ID.
Return cookie to the browser
Send request with cookie
Lookup and check session in database. Get user info from lookup of profile matching session.
Send response to client

This solution still works, but nowadays we have different requirements, i.e. hybrid application or single page application contacting multiple backends (split up into separate micro-service authetication servers, databases, image processing servers, etc). In these types of scenarios, the session cookie we get from one server won't correspond to another server.


Browser
Server
POST /login with username + password
Create JWT with secret
Return JWT to the browser
Send JWT in Authorization header
Check JWT signature with secret. Get user info from JWT
Send response to client

JWTs don't use sessions and have no problem with micro-service architectures. Instead of making a session and setting a cookie, the server will send you a JSON Web Token instead. Now you can use that token to do whatever you want to do with the server (that you have authorization to do).



Think of it like a hotel key: you register at the front-desk, and they give you one of those plastic electronic keys with which you can access your room, the pool, and the garage, but you can't open other people's rooms or go into the manager's office. And, like a hotel key, when your stay has ended, you're simply left with a useless piece of plastic (i.e., the token doesn't do anything anymore after it's expired).

Advantages of JWTs
No Session to Manage (stateless): The JWT is a self contained token which has authetication information, expire time information, and other user defined claims digitally signed.

Portable: A single token can be used with multiple backends.

No Cookies Required, So It's Very Mobile Friendly

Good Performance: It reduces the network round trip time.

Decoupled/Decentralized: The token can be generated anywhere. Authentication can happen on the resource server, or easily separated into its own server.

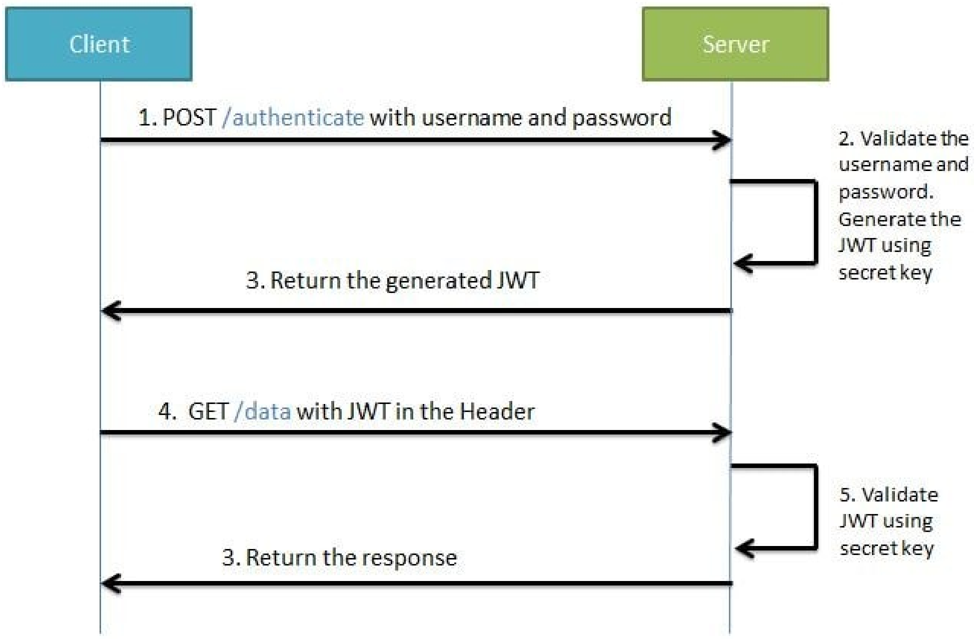
JWT Workflow

During the first request the client sends a POST request with username and password.

Upon successful authentication the server generates the JWT sends this JWT to the client.
This JWT can contain a payload of data.
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.

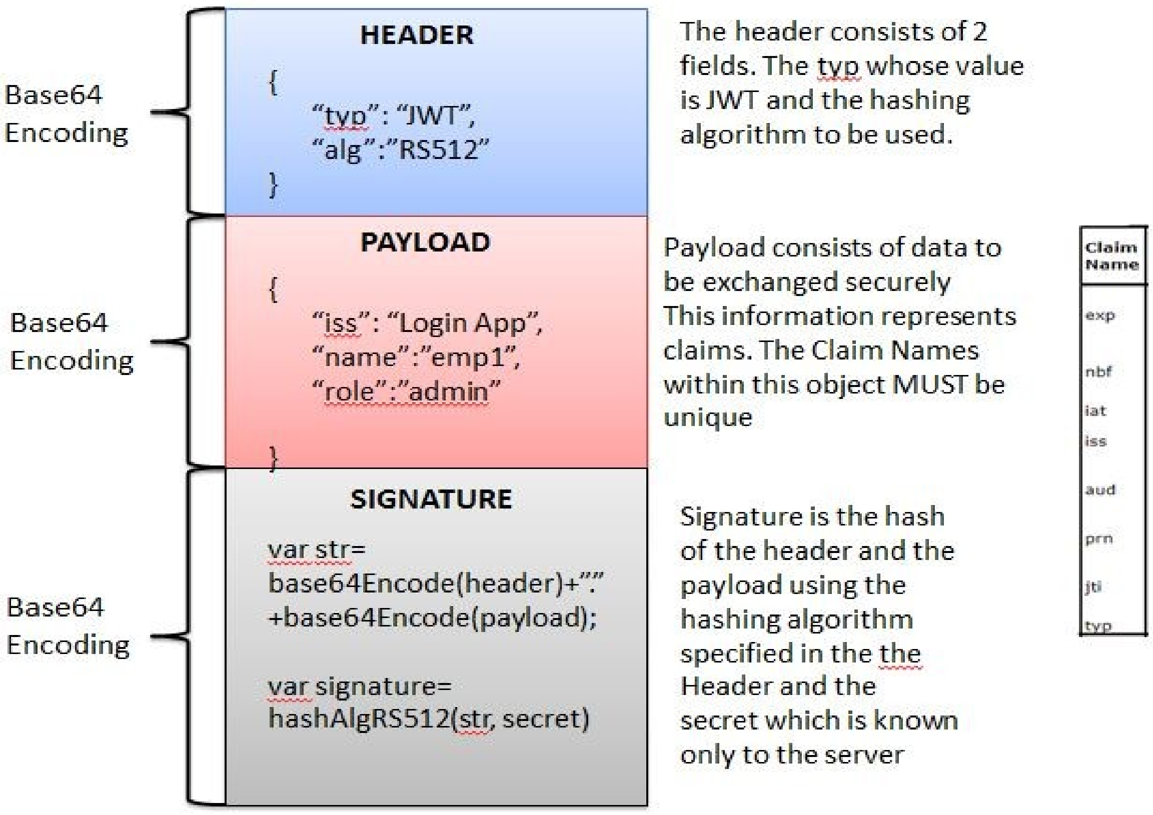
A JWT payload can contain things like user ID so that when the client again sends the JWT, you can be sure that it is issued by you, and you can see to whom it was issued.


Client
Server
1. POST /authenticate with username and password
2. Validate the username and password. Generate the JWT using secret key
3. Return the generated JWT
4. GET /data with JWT in the Header
5. Validate JWT using secret key
3. Return the response

Structure of JWT

JWT has the following format -

header.payload.signature



```
{
    "typ": "JWT",
    "alg":"RS512"
}
```

The header consists of 2 fields. The typ whose value is JWT and the hashing algorithm to be used.

```
{
    "iss": "Login App",
    "name":"emp1",
    "role":"admin"
}
```

Payload consists of data to be exchanged securely This information represents claims. The Claim Names within this object MUST be unique

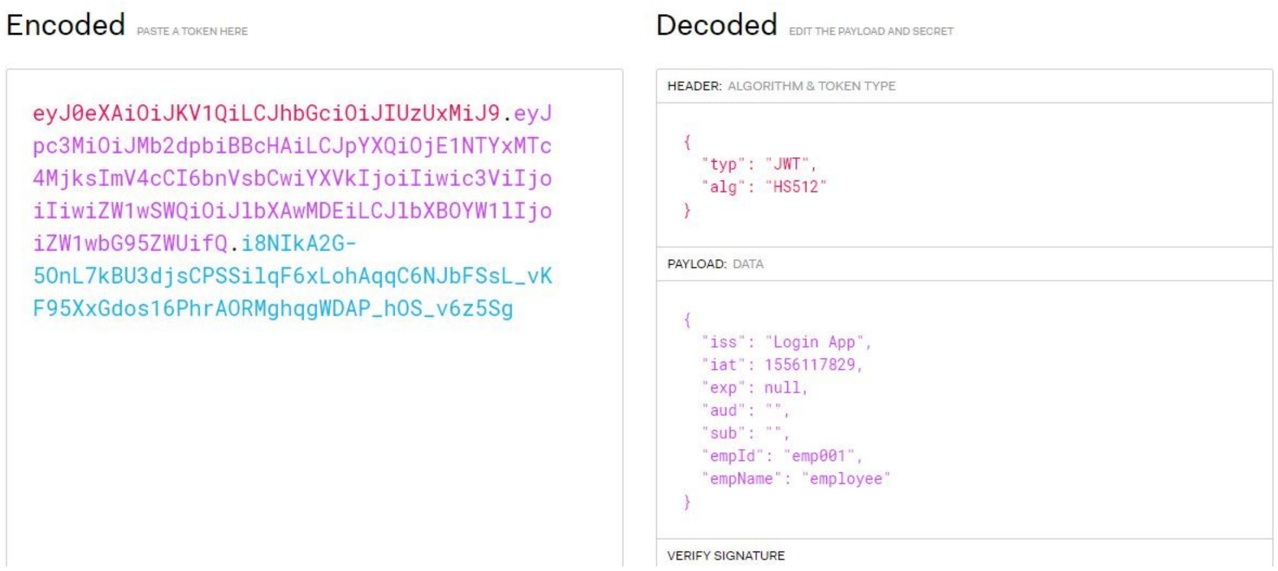
```
var str=
base64Encode(header)+"."
+base64Encode(payload);

var signature=
hashAlgRS512(str, secret)
```

Signature is the hash of the header and the payload using the hashing algorithm specified in the the Header and the secret which is known only to the server

An important point to remember about JWT is that the 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 recognize it.


Encoded PASTE A TOKEN HERE
Decoded EDIT THE PAYLOAD AND SECRET
HEADER: ALGORITHM & TOKEN TYPE
PAYLOAD: DATA
VERIFY SIGNATURE

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9.eyJ
pc3MiOiJMb2dpbiBBcHAiLCJpYXQiOjE1NTYxMTc
4MjksImV4cCI6bnVsbCwiYXVkIjoiIiwic3ViIjo
iIiwiZW1wSWQiOiJlbXAwMDEiLCJlbXBOYW1lIjo
iZW1wbG95ZWUifQ.i8NIkA2G-
5OnL7kBU3djsCPSSilqF6xLohAqqC6NJbFSsL_vK
F95XxGdos16PhrAORMghqgWDAP_hOS_v6z5Sg
```

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

```
{
  "iss": "Login App",
  "iat": 1556117829,
  "exp": null,
  "aud": "",
  "sub": "",
  "empId": "emp001",
  "empName": "employee"
}
```

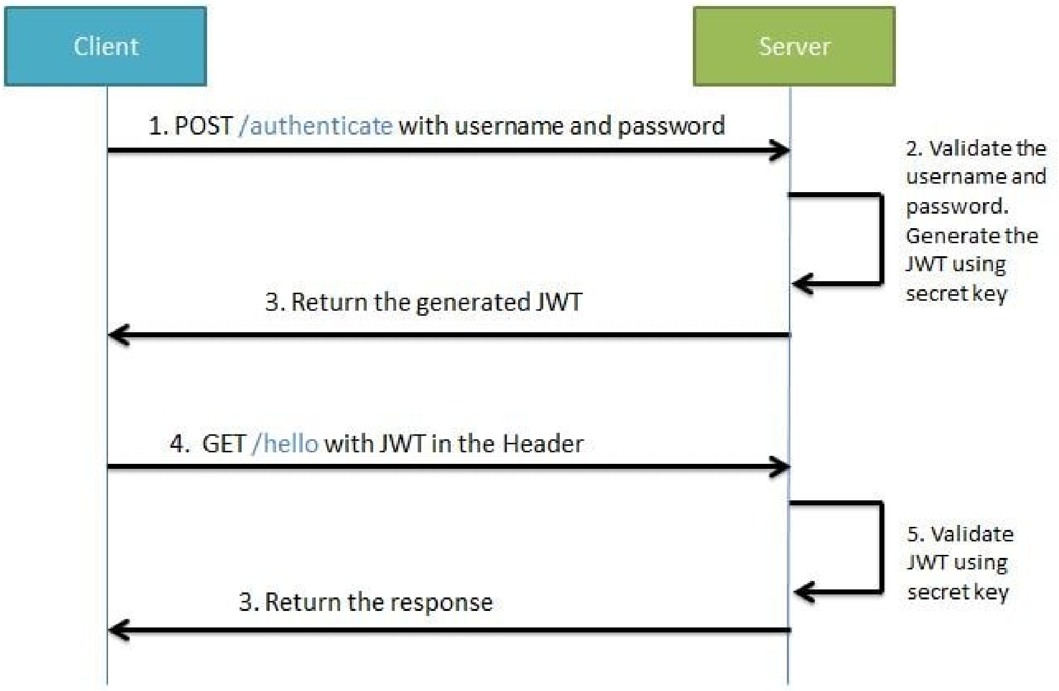
<u>Lets Begin?</u>

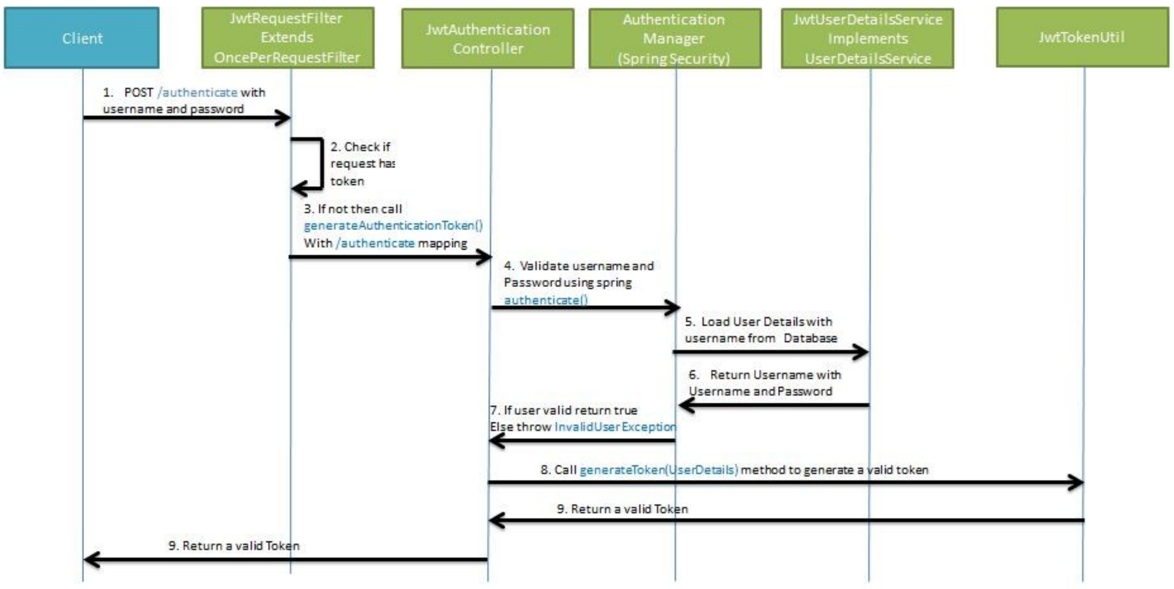
For better understanding we will be developing the project in stages

Develop a Spring Boot Application to expose a Simple REST GET API with mapping /hello.
Configure Spring Security for JWT.

Expose REST POST API with mapping /authenticate using which User will get a valid JSON Web Token. And then allow the user access to the api /hello only if it has a valid token

Spring Boot JWT Workflow


Client
Server
1. POST /authenticate with username and password
2. Validate the username and password. Generate the JWT using secret key
3. Return the generated JWT
4. GET /hello with JWT in the Header
5. Validate JWT using secret key
3. Return the response


Client
JwtRequestFilter Extends OncePerRequestFilter
JwtAuthentication Controller
Authentication Manager (Spring Security)
JwtUserDetailsService Implements UserDetailsService
JwtTokenUtil
1. POST /authenticate with username and password
2. Check if request has token
3. If not then call generateAuthenticationToken() With /authenticate mapping
4. Validate username and Password using spring authenticate()
5. Load User Details with username from Database
6. Return Username with Username and Password
7. If user valid return true Else throw InvalidUser Exception
8. Call generateToken(UserDetails) method to generate a valid token
9. Return a valid Token
9. Return a valid Token