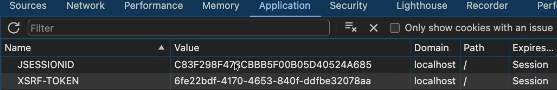
**10. token based authentication - json web token - section 11**

Opaque Tokens vs JSON Web Tokens (JWT)



With the help of JSESSIONID token, we don't have to send the username and password for every request on the backend server.

Also, we don't have to perform the authentication for each and every request.

So that's the purpose of JSESSIONID token, whereas with the help of XSRF-TOKEN,

the Spring Security framework is trying to protect us from the CSRF attacks.

They are just some random String values, but in real applications, you may get requirements to generate the tokens based upon advanced formats like JWT tokens.

I'm also going to teach you how to transmit the token from the back into the UI application without the help of cookies.

For critical applications, you may have requirement to not store the token details

inside the browser cookies as well.

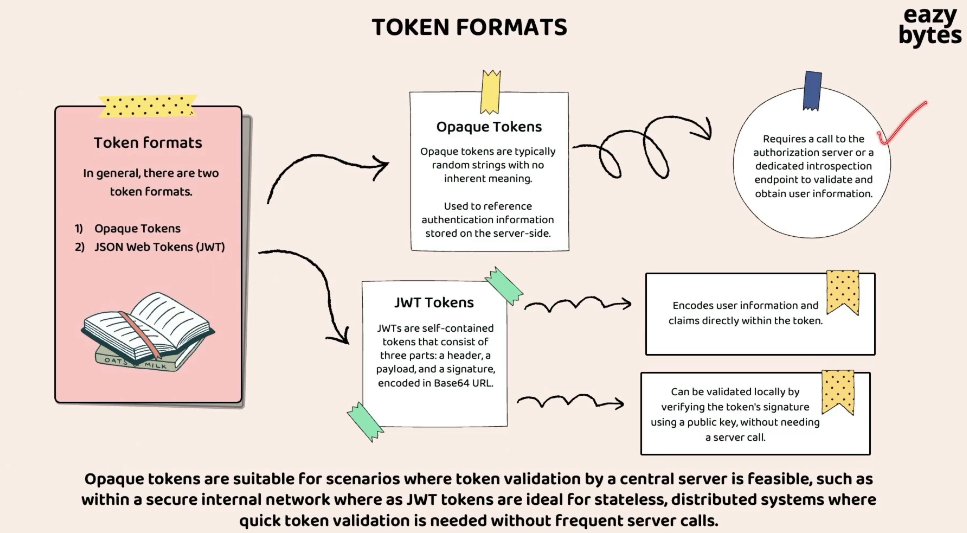
Inside such applications, you need to transmit the token from UI to backend

back into UI continuously for each request, so these kind of techniques we're going to apply inside our web application to make it more secure.

Inside the software industry in general, two types of token formats

are going to be used by the enterprise applications.

The two formats are Opaque Tokens and JSON Web Tokens (JWT or JOT).



**what is the difference between these tokens and when to use a specific token format.?**

The very first token format that we have here is Opaque Tokens.

These tokens are typically random String values with no inherent meaning.

So there won't be any meaning to your token, just a random String value,

which is maintained by your backend server for a user session.

So whenever you want to access a security API, you need to send the same Opaque Token, and behind the scenes, the backend server is going to check if there is any user session associated for a given token. If yes, it is going to allow you to access the secure API.

Whenever an application is using Opaque Tokens format, then there is always going to be a dependency on the backend server to validate the Opaque Tokens for each and every request.

Let me take a scenario?

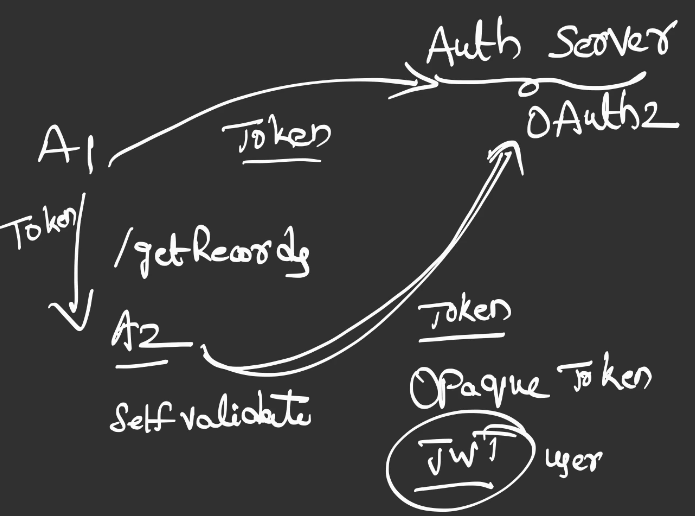
In enterprise applications, always the authentication are security-related logic.

They're going to be maintained inside the server called auth server.

We're going to discuss what is auth server and how to build it in the coming sections.

So these auth servers we're going to build by following the authentication

and authorization framework, which is OAuth2.



The purpose of this auth server is this auth server, it is always responsible

for authentication and authorization-related logic.

So think like there is an app one and there is an app two.

This app one, it is trying to invoke a secured API, maybe think this API as getRecords,

so it is going to give some records in the response, and this is a secured API.

So the A1, before it try to invoke the getRecords API of the A2, the A1 application, what it has to do, it has to go to the auth server and get that token by providing the credentials.

So the same token the A1 has to send to the A2 and the A2 will validate with the auth server if this token is valid or not.

So if the token provided by the A1 is valid, then only the A2 is going to provide the response. Otherwise, it is not going to give any response.

**So if you see here, the A2, it is dependent on the auth server to validate this token.**

**This scenario will occur only when we are using the Opaque Tokens,**

whereas when we try to use JWT format tokens, this A2, it don't have to depend on the auth software.

It can directly self-validate the token without reaching to the auth server,

so that's the advantage of JWT token.

Apart from this advantage, JWT tokens, they're also going to have user-related information as part of the token.

So whenever we are using the JWT format for our tokens, these A1 and the A2 applications will come to know to whom this token belongs, because inside this token itself, there will be some user details.

What are the authorities of the end user?

What is the expiration time of the token?

So all such details, they're going to be present inside the JWT tokens.

**Note that Opaque Tokens are some random Strings which does not have any meaning.**

**Since they don't have any meaning, every time we need to rely on the authorization server, our dedicated introspection endpoint to validate the token.**

**Whereas JWT tokens, they're going to have some user information, expiration time, some token signature, using which we can validate if the token is tampered or not locally itself.**

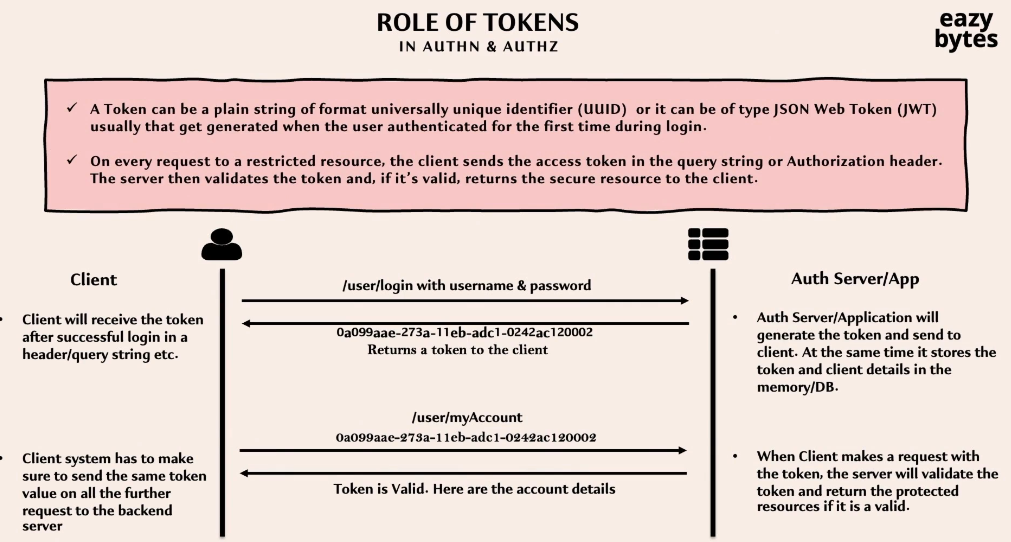
**Here also I have highlighted that Opaque Tokens are suitable for scenarios where token validation by central is feasible, such as within secure internal network,**

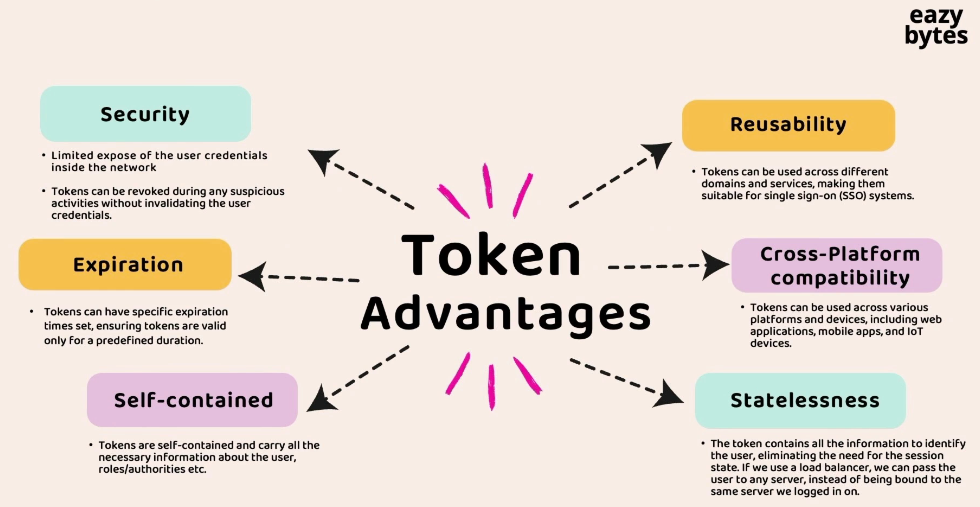
**whereas Jot Tokens are ideal for stateless distributed systems where quick token validation is needed without frequent server calls.**

**So these Jot tokens, they are going to be super helpful in the microservice scenarios**

**because microservices are stateless and distributed in nature.**

Advantages of Token based Authentication





Security:

The end user is going to share the credentials only once that is during the login operation. For all the remaining requests that the end user is going to make, the credentials are not going to be sent inside the network. Instead, the token is going to travel inside the network. So with this, we have one advantage, which is we are not unnecessarily exposing the user credentials to the network multiple times.

In the scenarios when our an enterprise organization saw a suspicious activity where some hacker stole the tokens, what they can do is they can simply invalidate all the tokens issued by the backend server.

when he's trying to make the request, the login page is going to be displayed

where the end user can enter actual credentials.

So these are very first advantage around the Security.

Expiration:

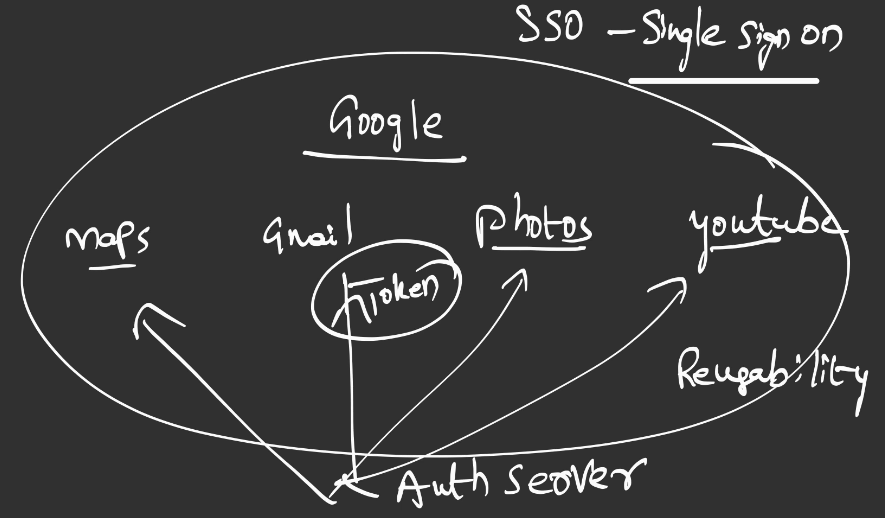
Whenever an backend server is issuing an token, it have a flexibility to issue the token with a specific expiration time. Usually for critical applications, the expiration time will be one hour, or sometimes it'll be even 30 minutes as well.

Self contained:

They have the ability to store the necessary information about the logged in user about their roles and authorities. So from these Self-contained tokens,

the client applications who received the token, they should be able to read the user information. They don't have to rely on the backend server or on the authorization server every time to know the end user details.

Reusability:



Take the scenario of Google.

So Google has lot many products under them, like it has maps, it has Gmail, it has photos, it has YouTube.

So there are good amount of products under the same company, which is Google or Alphabet.

So for all these applications, there is only going to be a single auth server,

which is going to issue the tokens. So when an end user logged in into his Gmail account, inside a browser is going to get a token. So this token is going to receive from the auth server.

Since this token is already present inside his browser

as part of a cookie, what is going to happen is if this end user, if you try to access another product of the Google inside the same browser, then the authentication is not going to happen one more time.

Instead, the same token will be leveraged by these applications like photos, YouTube, and maps, here the end user only logged in into his account once, and the token that the end user received can be used across applications.

So this is where the advantage of Reusability comes into picture.

**So this concept, we also call it as SSO, which means Single Sign On.**

**You are going to sign on into the application only once. Once the token is received,**

**the same token can be used by other applications as well.**

Cross platform compatibility

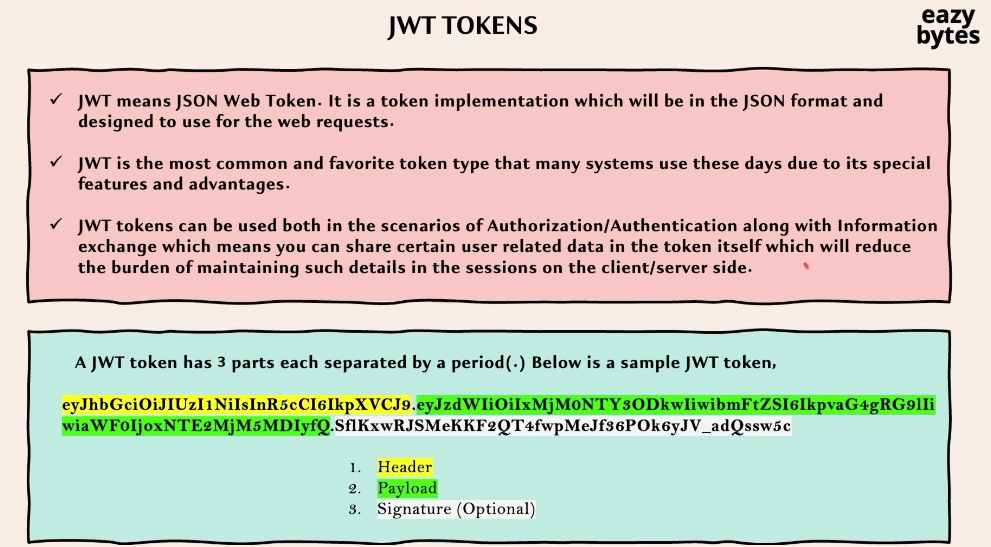
The same token that is issued by your backend server can be used inside web application, mobile applications, and IoT devices as well.

Stateless

Any user information that is going to be required by an application, since it is already going to be available as part of the Jot tokens, the applications, they don't have to remember the user information. Your application is going to be capable

of becoming Statelessness. This Statelessness feature is a mandatory for microservices.

Deep dive about JWT Tokens



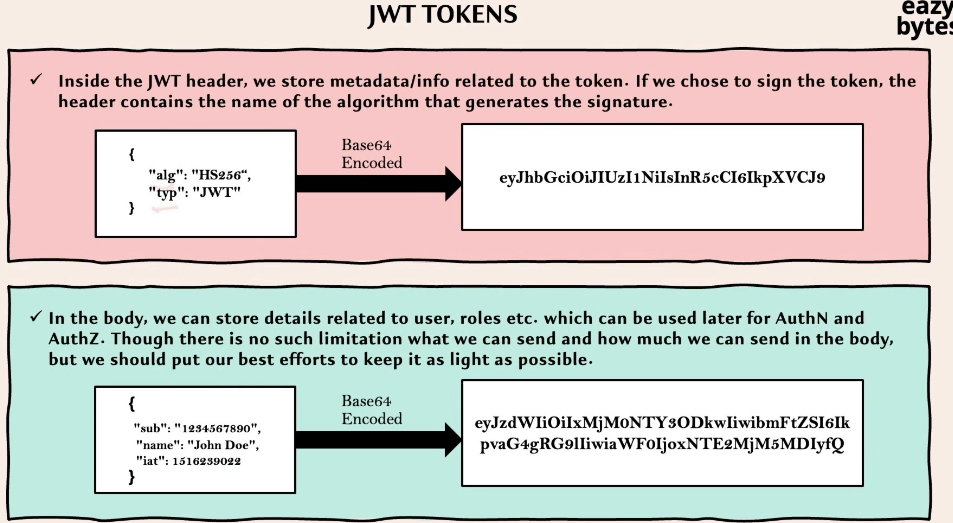
Jwt Header and payload or body

But don't try to store the password of the end user inside the payload because if you try to store the password of the end user inside the payload body, anyone who has access to your token, they should be able to see the password.

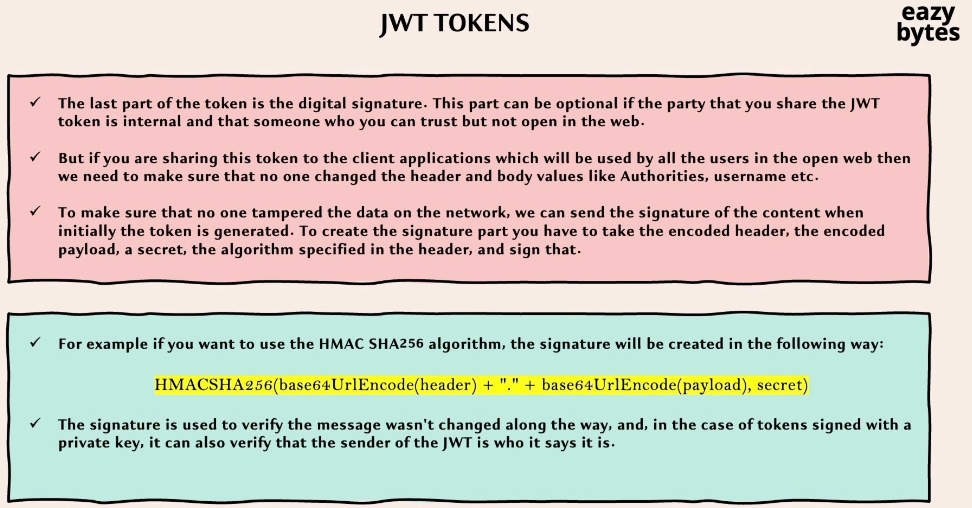
The reason is very simple. All your payload data, they're going to be Base64 encoded,

and the encoded value they're going to be stored inside the Jot token.

Anyone can see the original text by decoding these encoded data.



Digital Signature



let's try to understand under which scenarios this digital signature is optional.

If you are trying to generate a token for an application which is internal to your organization, and if you're sure that the token is not going to be tampered by the client applications or by your trusted users, in such scenarios, there is no need of digital signature because you have that confidence that no one is going

to tamper the tokens inside your own network.

But in the scenarios where you are going to generate the tokens for the client applications or for the end users who are going to use your application from the open web, then in such scenarios, you need to make sure that the Jot token also has digital signature.

**generate the digital signature:**

In order to generate the digital signature from the token, we need to use one of the hashing algorithm. So this HMACSHA256 is the most commonly-used hashing algorithm.

To this hashing algorithm, we need to send the input. So the input is calculated

by Base64 URL encoder of the header. So the header is going to be again encoded

with the help of Base64 URL encode format.

Later on, we need to add a period, and at last we should also Base64 URL encode the payload.

So this string is the very first input to the **SHA256** algorithm.

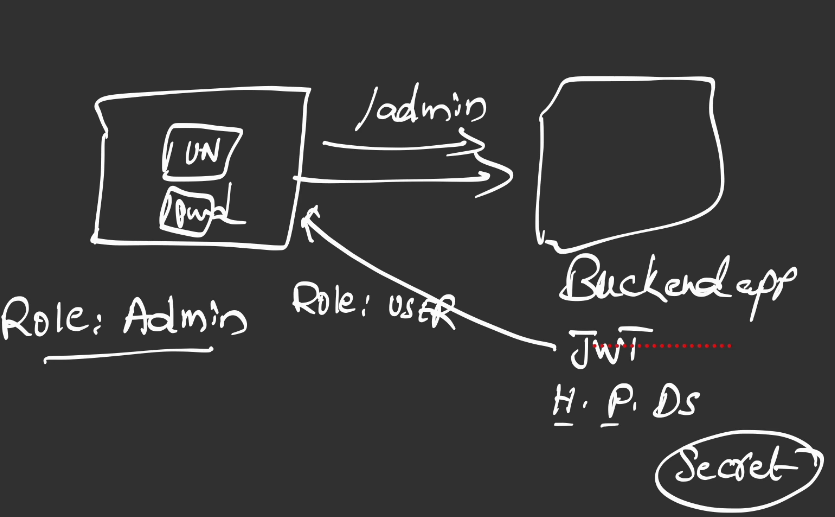
And the next input is the secret value that you want to involve.

So this secret value, you need to keep carefully with you.

You should not lose this secret.

If you lose this secret to someone, then they should be able to generate their own tokens, which are going to be accepted by your backend server.

Usually this secret value is going to be maintained carefully on the backend side.

****

Let's assume inside the payload or inside the body of the Jot token, the role of the end user is mentioned as user.

Let's assume the end user who logged in into the application, he's a hacker, or he's a bad user. So what he's going to do, he will check for the Jot token that he has received inside the response.

He should be able to tamper the token value. So the tampering that he's going to do is he will try to change the role to admin, which means he's trying to elevate his privileges inside the application.

So once the role value is changed, he's going to generate the base64 encoded value of the payload, and the same he's going to update inside the Jot token.

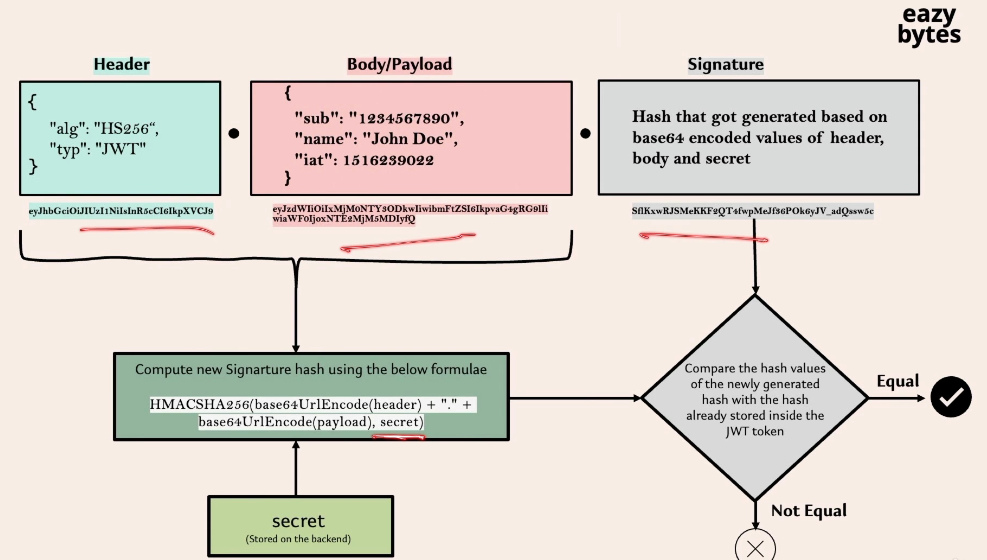
So this updated Jot token can be sent back to the backend server for the further request. Maybe inside the second request, the end user might try to the admin path

with this role admin, and he will be under the assumption that the backend server is going to give you a proper response, but that is not going to happen.

The backend server, which initially generated the token, it is going to validate the token if there is any tampering happen. Since the tampering happened here,

it is going to throw an 403 error.

**So let's try to understand how this backend server is going to identify the tampering scenario?**



So initially during the login operation, the backend server, it is going to generate this Jot token, which has header, body, payload, and signature.

So inside the signature we are going to have the hashing value.

So this hashing value derived by following this formula.

So if the end user or if the client application, if they try to tamper even a single character inside the payload or inside the header or inside the signature, it is going to be easily detected by the backend application.

How it is going to detect this, whenever it is receiving the Jot token in the future requests, **the backend server it is going to perform the same calculation again**.

It is going to take the header, payload, and feed as a first input parameter

to the same hashing algorithm along with the secret value that is known to the backend server only.

**If the input of the hashing algorithm is changed by even one character, it is going**

**to get a different hash value as an output. And if the newly-calculated hash value is not matching with the signature value present inside the token itself, then it is going to enter into the not equal scenario. Otherwise, it is going to enter into the equal scenario.**

You may have a question here.

What if the end user changed the signature itself to match the tampered payload value or tampered header value?

So this can be your question. So the hacker can't do that because the hacker don't have the secret value.

So if someone get this secret value which is being maintained by the backend,

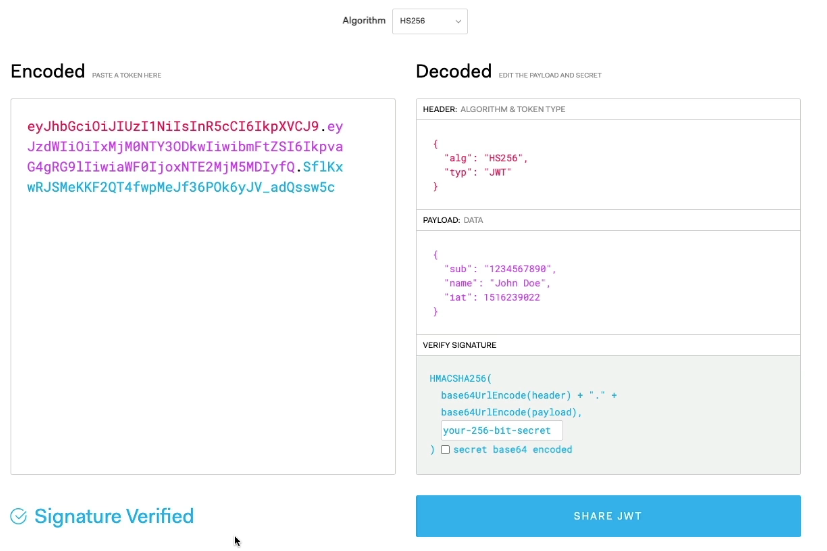
they should be able to generate the successful tokens with whatever data that they want.

So that's why always securing the secret properly inside your backend application is most important. The same I have highlighted here also, the secret is always going to be stored on the backend. So the backend application instead of storing

inside the properties files, they may inject the secret value as an environment variable or they can read from a secret vault. So there are many approaches that industry uses.

As long as the secret is secured, the token tampering scenario can be easily detected by the backend applications.

I hope you understand this tampering scenario, how it is going to detected by the digital signature part inside the Jot token.



Anytime if you want to understand a Jot token value,

you can go to your website, which is JWT.io.

Inside this website, if you provide the Jot token as an input value, it will display the header value in a plain text followed by payload.

At last, it is going to tell you what is the formula that people has used to generate the signature. Since the hash value can't be decoded to your original value, that's why here they're not able to show the original value like secret and the other input parameters that we have used for hashing.

So if you try to understand this default Jot token value present here, so this is the header part which is encoded. This website will decode it and display a normal text value.

So the algorithm that they're trying to use is HS256. In case if someone is using different hashing algorithm, they need to select the corresponding one,

but most of the times HS256 should be good enough.

And this is the payload data representation based upon the encoded value here.

And this is the digital signature.

How this digital signature is calculated?

By using this formula.

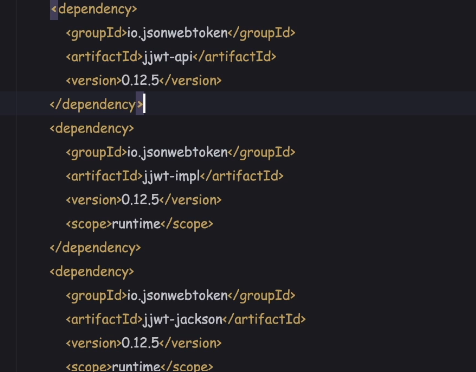
So the same formula is going to be used during the verification process as well.

Think like I'm trying to tamper one of the letter, and you'll be able to see there is an error here. Invalid signature.

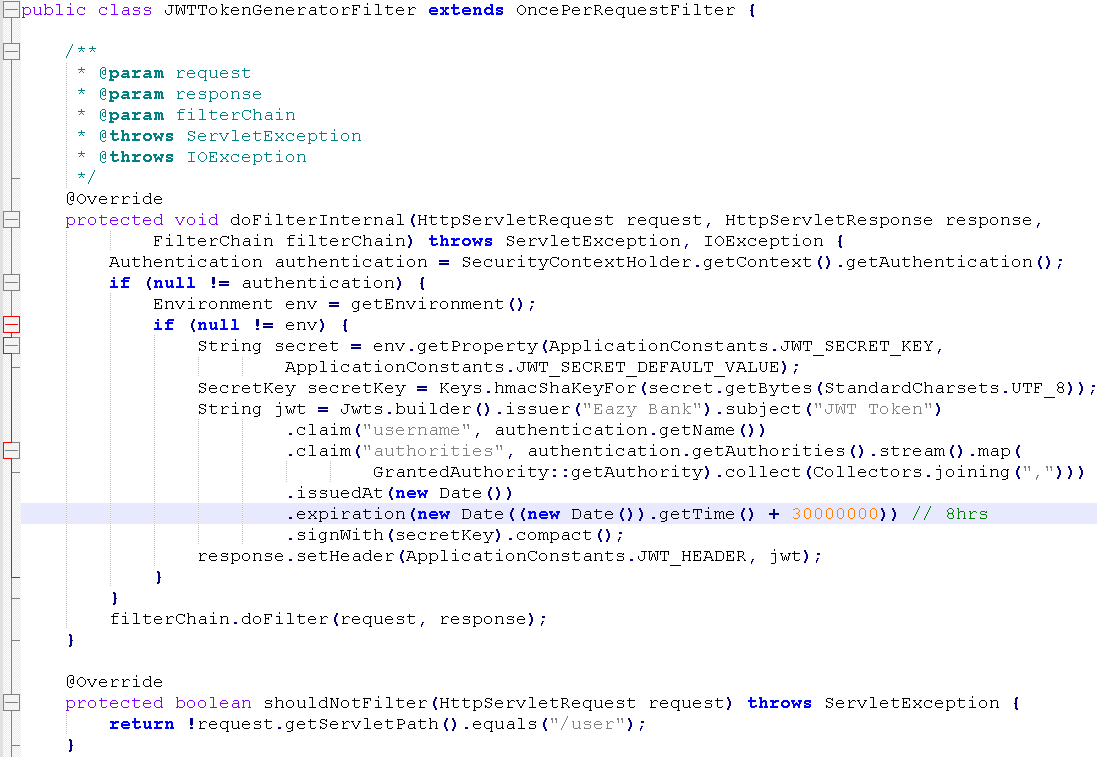
That means the content is not matching with my signature.

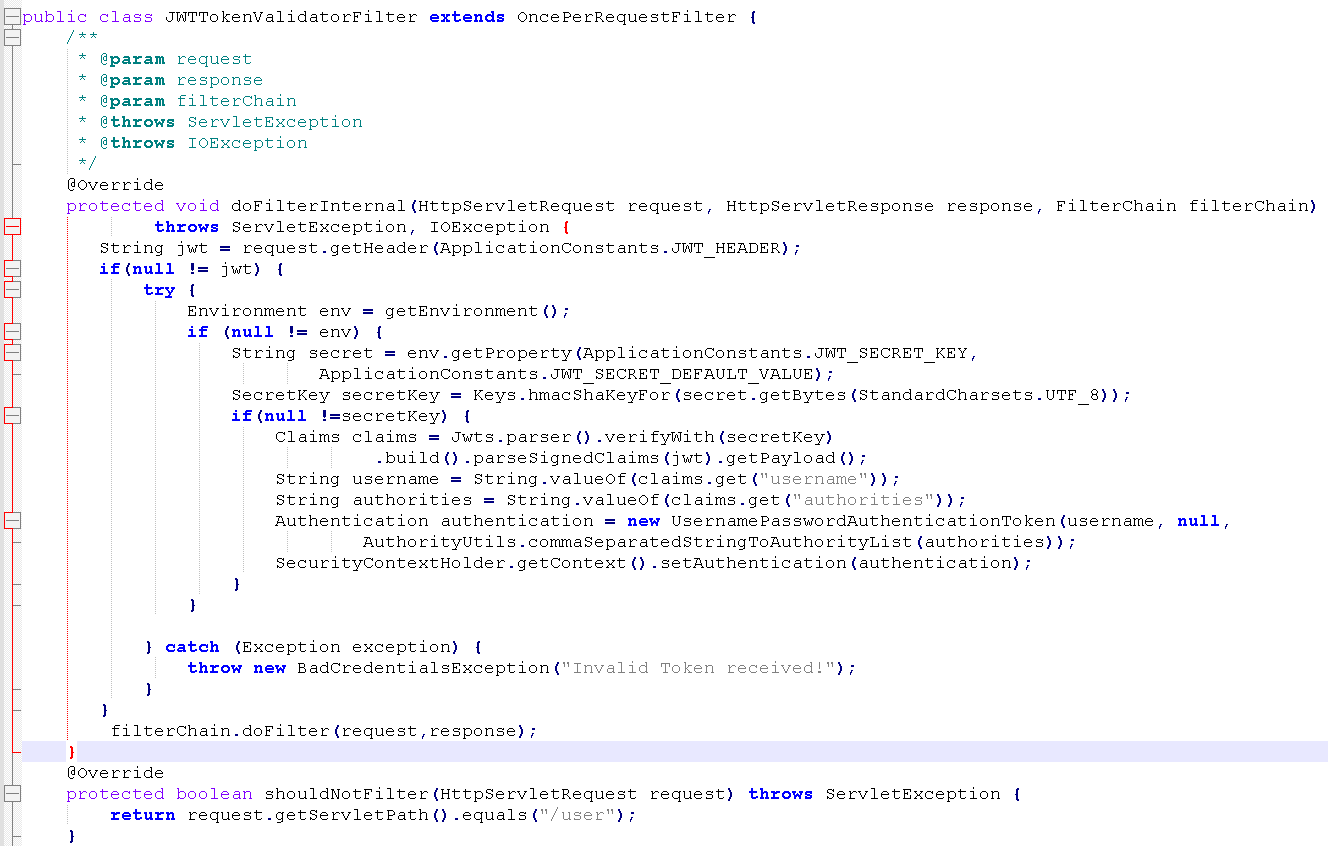
With that the entire Jot token is going to become invalid.

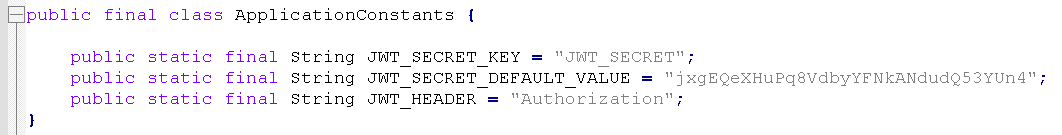
Making project configuration to use JWT token

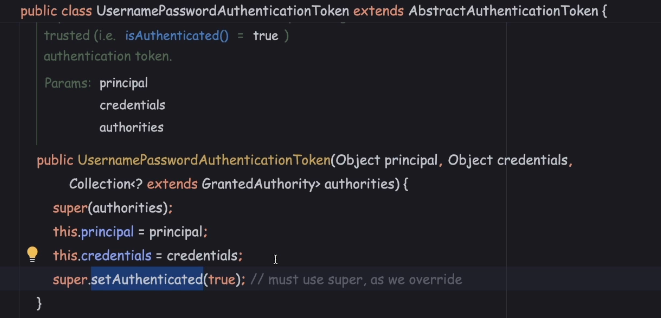
****

****

****

****

****

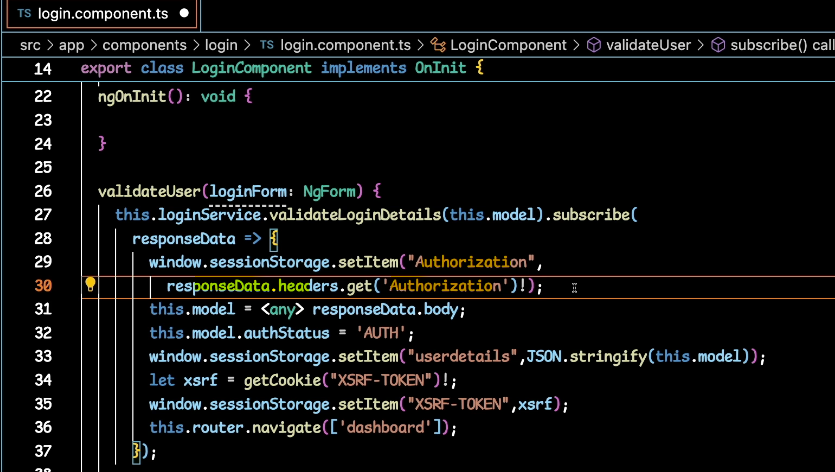
****

So this is an indication to the spring security framework that this fellow already completed authentication, that's why this authentication object has a Boolean value as true. So based upon this Boolean value, my spring security framework

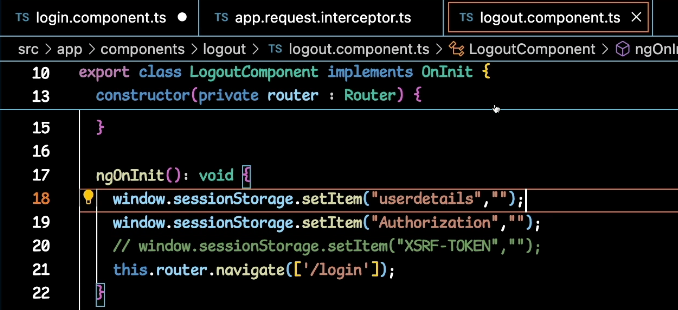
will not try to authenticate the user one more time, because the Boolean value is true. If the Boolean value is false, then only it'll try to authenticate the user one more time. Once this authentication object is created, we need to store it into the SecurityContextHolder by invoking the getContext() method and invoking the setAuthentication() method. To this method, we need to pass the authentication object.

So these are the only changes that we have to do inside this TokenValidatorFilter.

Making changes on the client side for JWT token based authentication

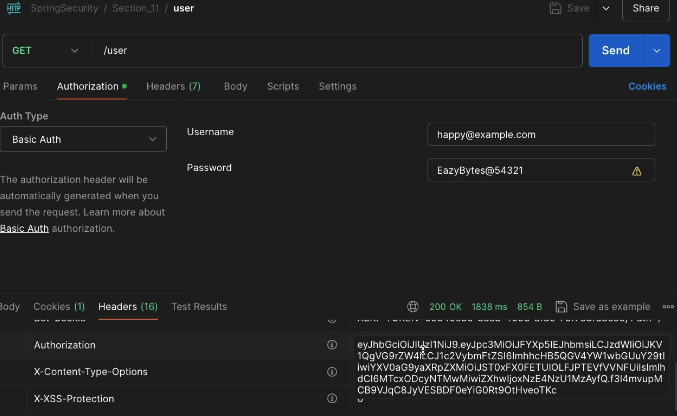
****

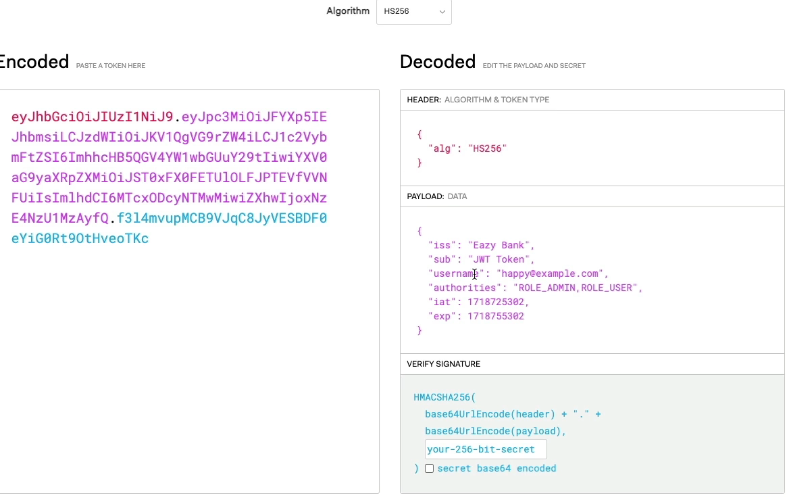
****

****

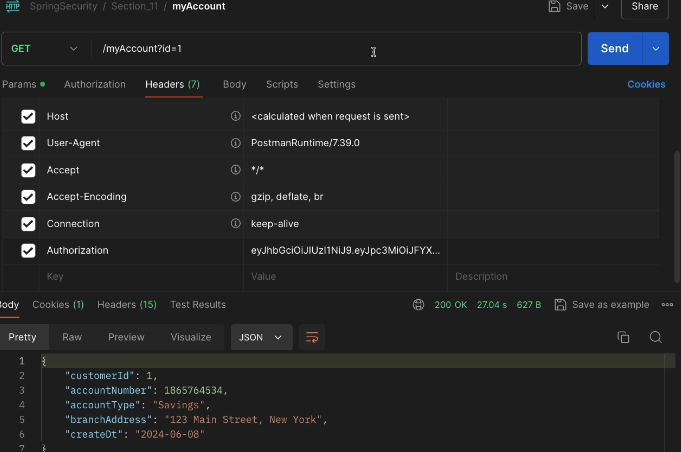
Validating the JWT changes made by running the applications

Login user api

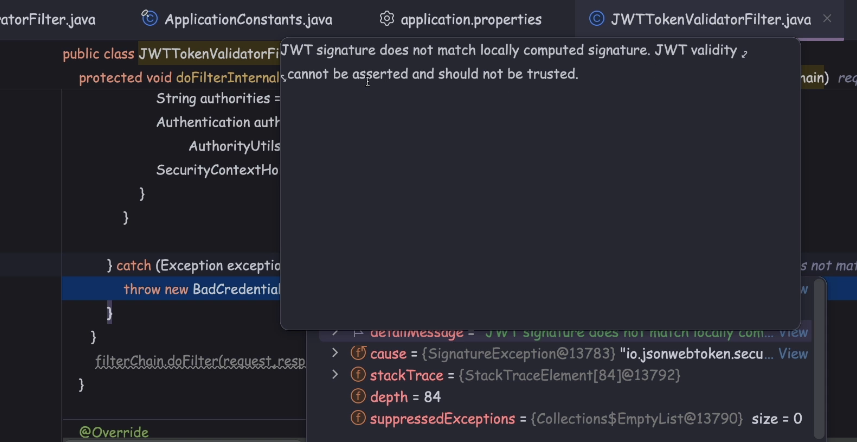
****

****

**Test other secured apis by sending authorization header and no auth token**

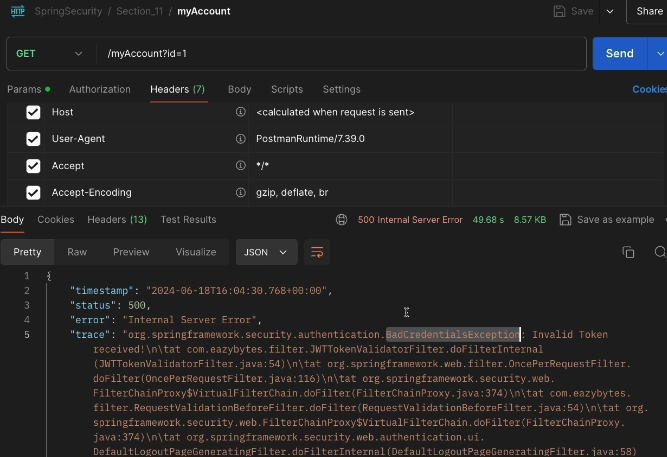
****

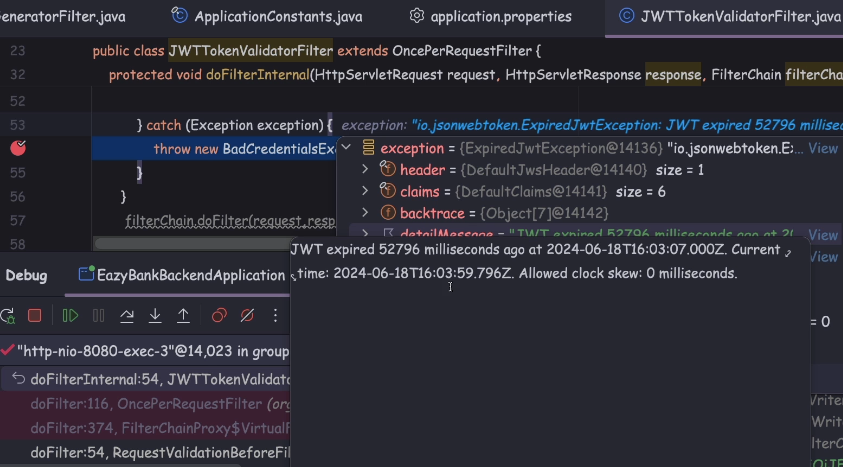
**Tamper authorization token in above api**

****

Validating the JWT token expiration scenario

Inside our JWTTokenGeneratorFilter, we have set the expiration time as around eight hours. If anyone tried to access this token after eight hours, it is going to be useless for them. The end user again has to restart the login journey to get a new token.



****

Publish an AuthenticationManager for custom or manual authentication

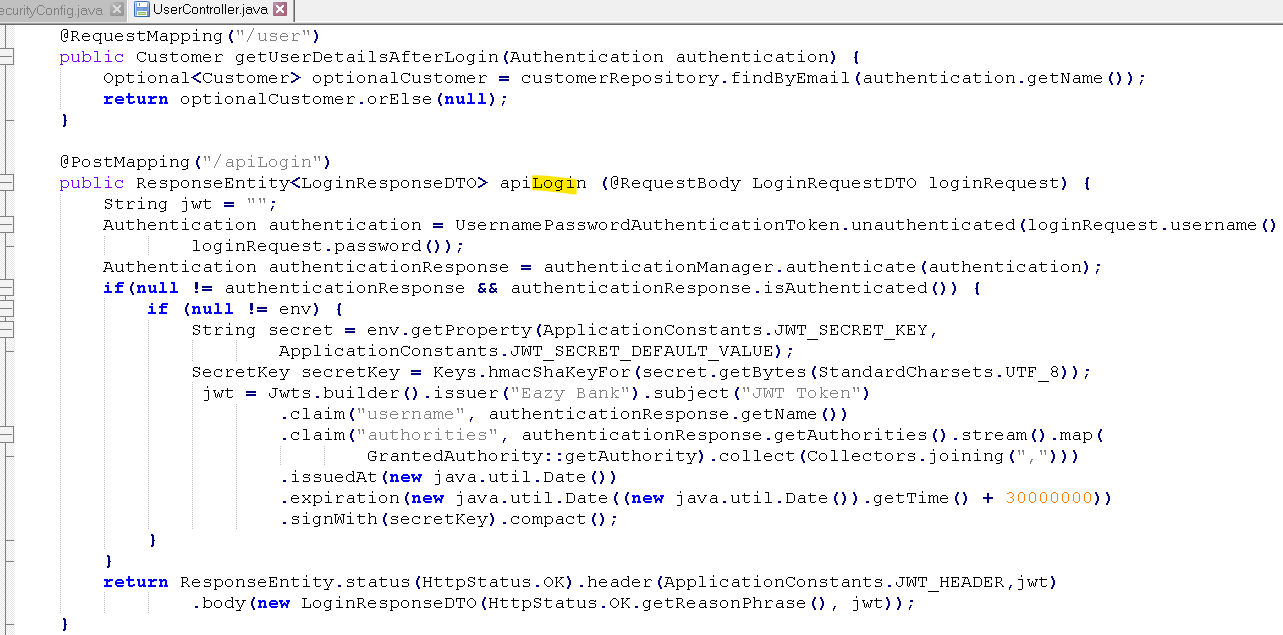
Inside real projects you may get a requirement, which is that you need to accept

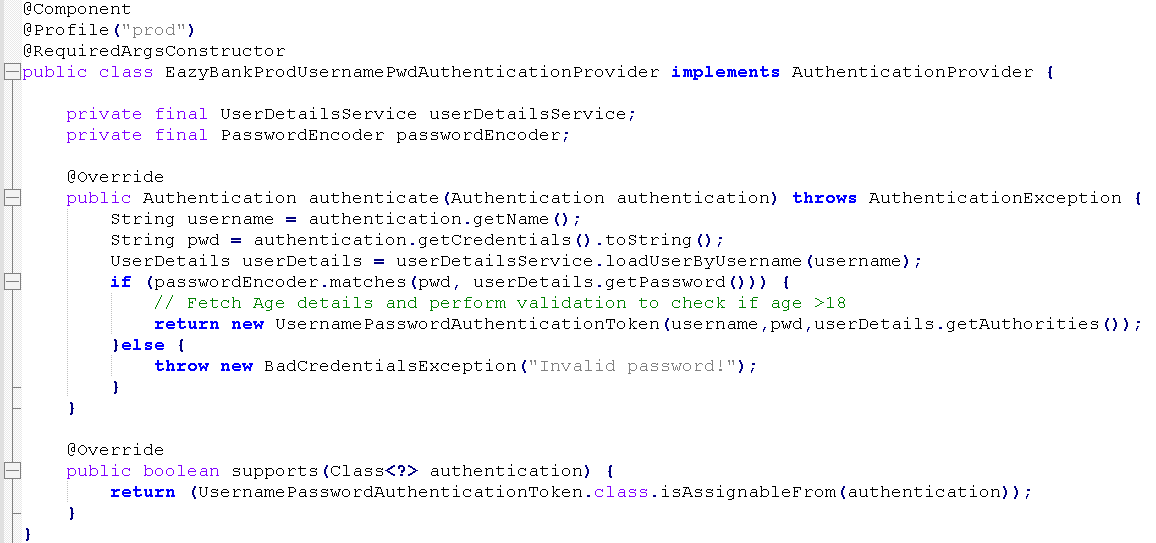
the credentials inside the RequestBody or it can be inside the RequestHeaders.

But the request that you're going to receive is not going to follow any of the standards of form login and HttpBasic.

In these kind of scenarios, whatever spring security provider it is not going to help you. You need to build your own rest API operation that accepts the user credentials wherever you want. And inside this rest API that you're going to build, you need to invoke the authentication manually.

Once the authentication is completed, the JWT token or any other token value you want to send inside the ResponseBody. So these kind of requirements you may get in real applications.





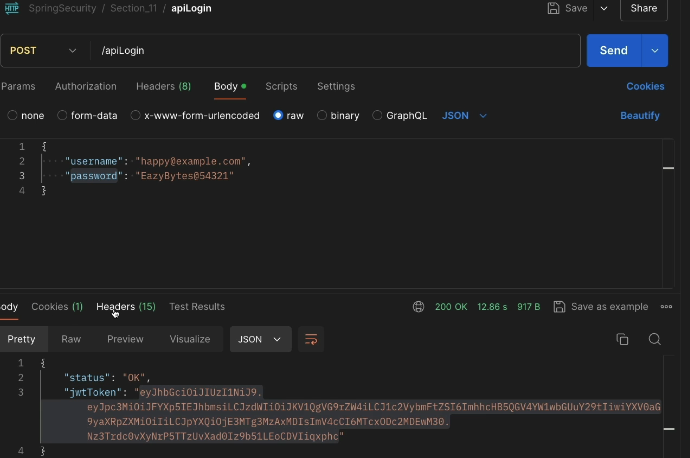
So since this is an first API that is going to be invoked by my clients, I need to make this permitAll(). Under the ProjectSecurityConfig class, here we have permitAll() configuration. So let me mention the new API path, which is /apiLogin.

And since this API is going to support HTTP post, I should also mention this API path

under the CSRF Ignore list. Otherwise, the request is going to be blocked due to the CSRF protection. Since this path is a public path, we don't have to protect it from the CSRF attack.



Demo



There is an response header with the name authorization with the same jwtToken value.



Top of Form

Which of the following statement is TRUE regarding token based authentication?

**Token helps us not to share the credentials for every request which is a security risk to make credentials send over the network frequently.**

Top of Form

Which of the following statements is FALSE regarding JWT tokens?

* 

**JWT means JSON Web Token.**

* 

**JWT tokens can be used both in the scenarios of Authorization/Authentication along with Information exchange which means you can share certain user related data in the token itself which will reduce the burden of maintaining such details in the sessions on the server side.**

* 

**JWT is the most common and favorite token type that many systems use these days due to its special features and advantages.**

* 

**JWT is a token implementation which will be in the XML format and designed to use for the web requests.**

Bottom of Form

Top of Form

Which of the following is the correct order of 3 parts present inside JWT tokens?

**Header.Payload.Signature**

Top of Form

Which of the following part of JWT token is optional?

**Signature**

Top of Form

Which of the following best describes about header part inside JWT tokens?

**Inside the header of JWT token, we store metadata/info related to the token. If I chose to sign the token, the header contains the name of the algorithm that generates the signature.**

Top of Form

What is a key difference between Opaque tokens and JWT (JSON Web Tokens) in the context of Spring Security?

**Opaque tokens are not self-contained and require the authorization server to validate and obtain user information**