**OpenID** is an open standard and decentralized authentication protocol promoted by the non-profit OpenID Foundation such as Google, Microsoft and Facebook

It allows users to be authenticated by cooperating sites (known as relying parties, or RP) using a third-party identity provider (IDP) service, eliminating the need for webmasters to provide their own ad hoc login systems, and allowing users to log into multiple unrelated websites without having to have a separate identity and password for each.[1] Users create accounts by selecting an OpenID identity provider,[1] and then use those accounts to sign onto any website that accepts OpenID authentication. Several large organizations either issue or accept OpenIDs on their websites

Simply, **OpenID** is used for authentication <https://www.baeldung.com/spring-security-openid-connect>

**OAuth** is an open standard for access delegation.

**OpenID Connect (OIDC)** Combines the features of OpenID and OAuth i.e. does both Authentication and Authorization. OpenID Connect is built on the OAuth 2.0 protocol and uses an additional JSON Web Token (JWT), called an **ID token**, to standardize areas that OAuth 2.0 leaves up to choice, such as scopes and endpoint discovery. It is specifically focused on user authentication and is widely used to enable user logins on consumer websites and mobile apps.

If you’ve used your Google to sign into applications like YouTube, or Facebook to log into an online shopping cart, then you’re familiar with this authentication option. OpenID Connect is an open standard that organizations use to authenticate users. IdPs use this so that users can sign in to the IdP, and then access other websites and apps without having to log in or share their sign-in information.

**SAML** is independent of OAuth, relying on an exchange of messages to authenticate in XML SAML format, as opposed to JWT. It is more commonly used to help enterprise users sign in to multiple applications using a single login.

You’ve more likely experienced SAML authentication in action in the work environment. For example, it enables you to log into your corporate intranet or IdP and then access numerous additional services, such as Salesforce, Box, or Workday, without having to re-enter your credentials. SAML is an XML-based standard for exchanging authentication and authorization data between IdPs and service providers to verify the user’s identity and permissions, then grant or deny their access to services.

References:

<https://www.okta.com/identity-101/whats-the-difference-between-oauth-openid-connect-and-saml/>

**OAuth 2.0 terminology**

**Resource Owner**: Entity that can grant access to a protected resource. Typically, this is the end-user.

**Client**: Application requesting access to a protected resource on behalf of the Resource Owner.

**Resource Server**: Server hosting the protected resources. This is the API you want to access.

**Authorization Server**: Server that authenticates the Resource Owner and issues Access Tokens after getting proper authorization. In this case, Auth0.

**User Agent**: Agent used by the Resource Owner to interact with the Client (for example, a browser or a native application).

**Refresh token**: The Refresh Token grant type is used by clients to exchange a refresh token for an access token when the access token has expired. This allows clients to continue to have a valid access token without further interaction with the user.

References to OAuth specs

<https://www.oauth.com/oauth2-servers/map-oauth-2-0-specs/>

<https://datatracker.ietf.org/doc/html/rfc6749>

<https://oauth.net/specs/>

**JWT 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. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA**.**

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

1. Header: The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA. For example:

{

"alg": "HS256",

"typ": "JWT"

}

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

1. Payload: The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. An example payload could be:

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

The payload is then Base64Url encoded to form the second part of the JSON Web Token.

Do note that for signed tokens this information, though protected against tampering, is readable by anyone. Do not put secret information in the payload or header elements of a JWT unless it is encrypted.

1. Signature: to create the signature part you must take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that in this signature.

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is

More useful information will be found in the <https://jwt.io/introduction>

Now let us put some lights on that by discussing the below options or scenarios for the OAuth

The oauth2.0 specification <https://oauth.net/specs/> defines four authorization grant flows for issuing access token:

1. Authorization code grant flow: this is the safest, but also the most complex grant flow. This grant flow requires that user interact with the authorization server using a web browser for **authentication** and **giving consent** to the client application.

First, the client app will ask the authorization server to get the access as below

<http://localhost:8080/oauth/authorize?response_type=code&client_id=web&redirect_uri=http://my.redirect.uri&scope=read&state=3527>

where:

response\_type : means that this is authorization code grant flow

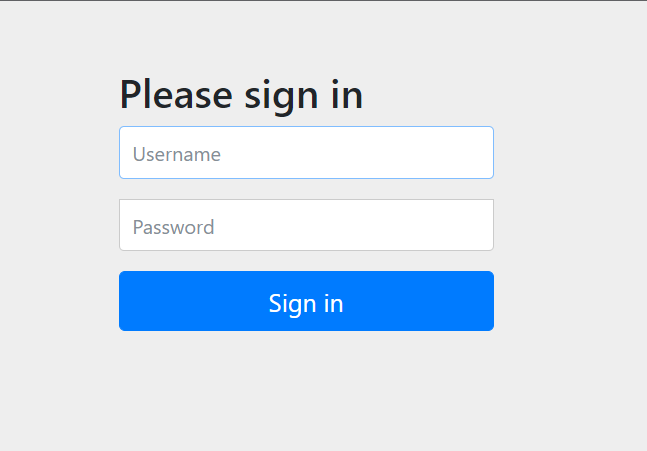
client\_id: the client Id configured in oauth\_client\_details.CLIENT\_ID

redirect\_uri: This is a client application URL. This tells authorization server where to send the grant code. Since the grant code is passed back to the client application using the web browser, that is , to an ensure environment where malicious Java script code potentially can pick up the grant code .It is only allowed to be used once and only during a short time period. This redirect\_uri should be configured to this client Id in the table oauth\_client\_details.WEB\_SERVER\_REDIRECT\_URI

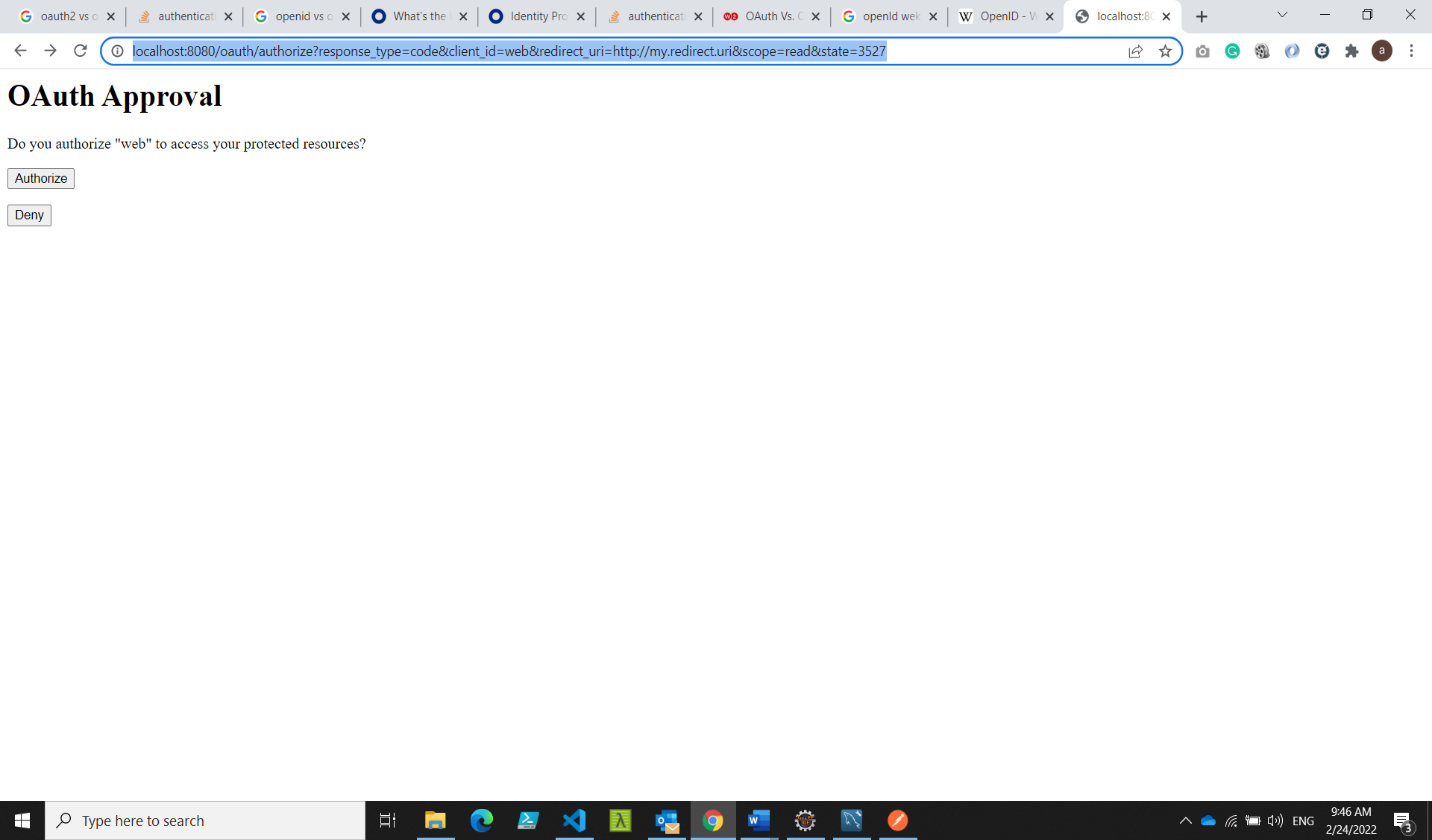
state: The application generates a random string and includes it in the request. It should then check that the same value is returned after the user authorizes the app. This is used to prevent CSRF attacks.

Note that the request above is a get request and you need to share neither the password of the client id nor the user credentials

The user is now asked to authenticate himself by username and password

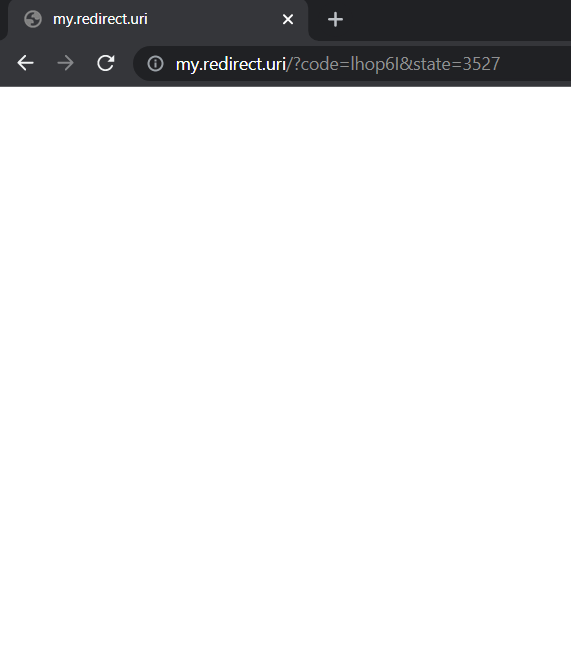


When user puts his credentials correctly, he will be asked to give his consent



if the user clicks on approve, he will be redirected (through browser) to the client app uri and gives this uri the grant code as below

<http://my.redirect.uri/?code=iqYqI1&state=3527>



Now the client app should use this grant code that was given to him and issue server-side call to **the token end point exposed by the authorization server** to get an access token. In this call the client app should provide the ouath server his client Id, his secret and the grant code

curl -k http://**web**:**123123**@localhost:8080/oauth/token -d **grant\_type**=authorization\_code -d **client\_id**=web -d **redirect\_uri**=http://my.redirect.uri -d **code**=iqYqI1

the result should be something like below

{

"access\_token":"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6ImJhZWwta2V5LWlkIn0.eyJhdWQiOlsicmVzb3VyY2Utc2VydmVyLXJlc3QtYXBpIl0sInVzZXJfbmFtZSI6InVzZXIxQGR4Yy5jb20iLCJzY29wZSI6WyJyZWFkIl0sImV4cCI6MTY0NTcwMDI3NSwiYXV0aG9yaXRpZXMiOlsiUk9MRV9FTVBMT1lFRSJdLCJqdGkiOiJmOGJiOTdiZC05YTk1LTQ4YWEtOTdjMi04ZGE4ZmRjYmQxMTYiLCJjbGllbnRfaWQiOiJ3ZWIifQ.bJQ0Muvq8n47sJ2oyrZkqsofvkF3gfZY3EOomN9uv6nBk6lxfqod7WwyjhG6ZebTi2wmgZW4E\_zfk5KByu6Dj10gdCh-gRI8-Ador1lDtUHXlxGWyv\_VODMktb2xiOZrnxMec9AQp7k5YIX4p3sCo\_l0B4e4R36UpUx\_S5KTamkfdvsFGG-gS9DtwJ6MWvs403lh1gjTP4bfsSu5NU547v7KrH8PESTy7LBRHCDlwnufBPs0SxDP6cQWnyKu-lk-TwZD40trKDVM5iz6dyJ1qWLVeFmFI8dmghfsjN05rlLc7CSrfMUxvVFRek472PzJT42EOSMjRb-mEsrJrJBPQA",

"token\_type":"bearer",

"refresh\_token":"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6ImJhZWwta2V5LWlkIn0.eyJhdWQiOlsicmVzb3VyY2Utc2VydmVyLXJlc3QtYXBpIl0sInVzZXJfbmFtZSI6InVzZXIxQGR4Yy5jb20iLCJzY29wZSI6WyJyZWFkIl0sImF0aSI6ImY4YmI5N2JkLTlhOTUtNDhhYS05N2MyLThkYThmZGNiZDExNiIsImV4cCI6MTY0ODI4MTQ3NSwiYXV0aG9yaXRpZXMiOlsiUk9MRV9FTVBMT1lFRSJdLCJqdGkiOiI2YWE3Yjk0NC1hZjE0LTQ5OGEtYjBhNi02YjVkZWMxMTUwYTYiLCJjbGllbnRfaWQiOiJ3ZWIifQ.cE-owivE0YmN\_swpvkiocbvr7XjxnhaIAdk7VW5m9-qx5miEnOmcX3QXCvLNvU8qdpz9F6pHkIdq07lbP\_HgJEv8toZyivmaouIYLdF2fvQZraC9utEI1d6CkEOZRWUY243HGwQKHgrVRbHIzJr4m9q4Ve2NKbSbPndCbfuK31gb2r35D1JgJjU9qZSDXbFYC8\_M-vIyuch9b35ZbyZsXnGJ4QpnHm48A7\_wWh5Vf2lnJn2sdhDw\_FxVhqYdLacZ-y1sCC2p35gvDNLB5erCU2oQs4W2GTxkRaM5O3XpseyE\_qRYbMNT0eDEK3Gcn90a6Iw0iJXs8bKsWdbjpnNNmA",

"expires\_in":10799,

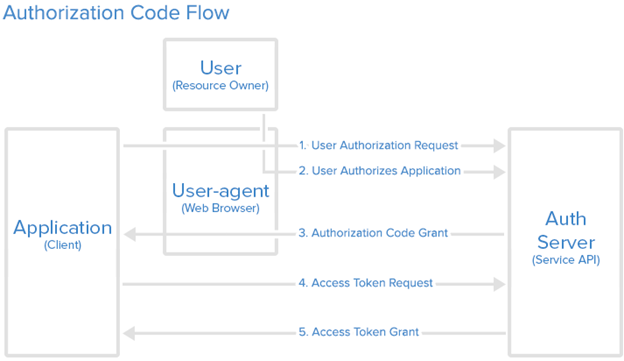
"scope":"read",

"jti":"f8bb97bd-9a95-48aa-97c2-8da8fdcbd116"

}

Now the client app can use this token to access the protected API or resource that is exposed by the resource server

* Note that redirect\_uri and code should be provided together, this is because the authorization server matches the code with the redirect URI that is configured to the client. Also, the authorization code is for one-time use. If you try to get the access token again using the same code value, you will get “Invalid authorization code” message
* Note that when the request was initiated on browser client\_id was only provided. At the nd when getting the token using the authorization\_code client\_id/password were provided



One additional specification that is worth some extra attentions is RFC7636-proof key for code exchange by OAuth Public Clients PKCE.This specification describes how an otherwise insecure public clients such as mobile native app or desktop application can utilize the code grant flow by adding extra layer of security

1. **Implicit grant flow**: this is also web based and is like the authorized code grant flow but less secure. It gets the access token directly from authorization server instead of grant code

This grant flow is intended for client application that is not be able to keep client secret protected. For example, single-page app (java or type script app)

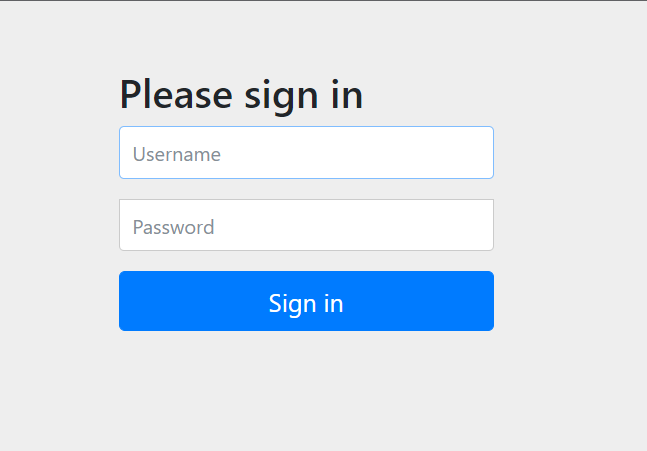
In practice, any benefit gained from the initial simplicity is lost in the other factors required to make this flow secure

First, the client app will ask the authorization server to get the access as below

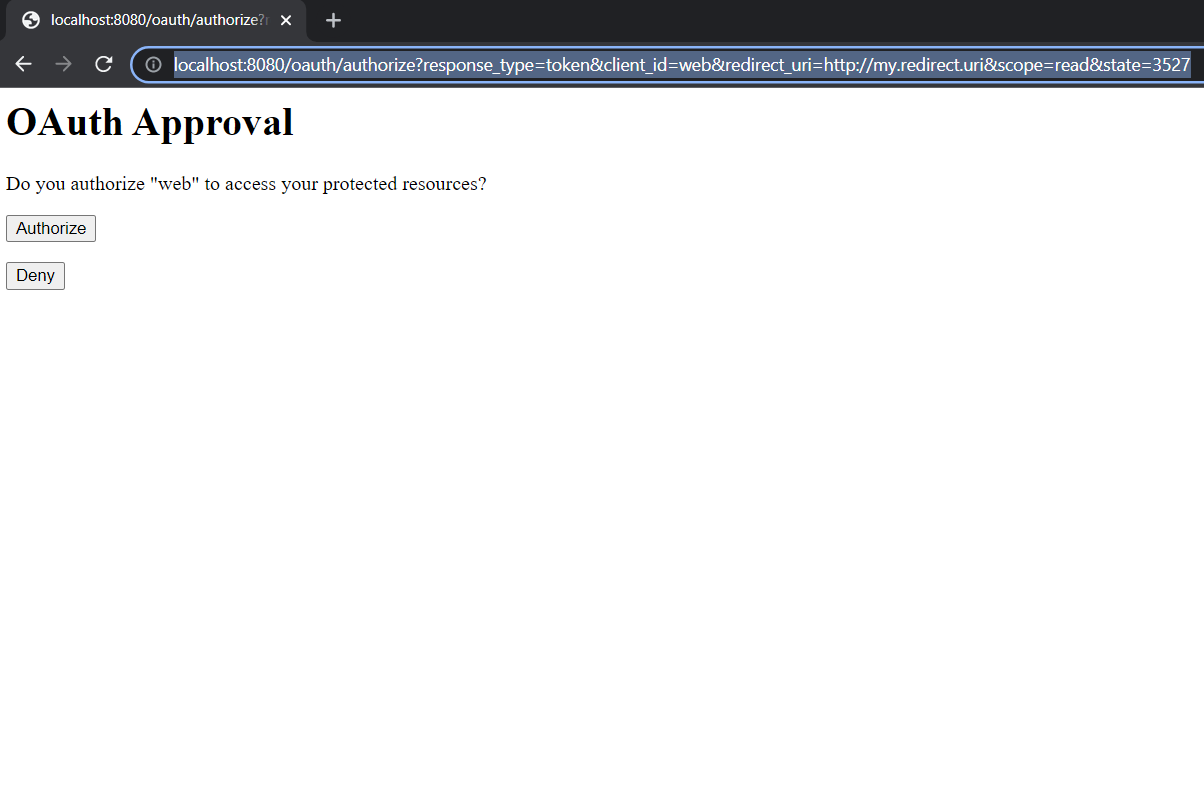
<http://localhost:8080/oauth/authorize?response_type=token&client_id=web&redirect_uri=http://my.redirect.uri&scope=read&state=3527>

it is the same URL but with response\_type=token instead of response\_type=code

The user is now asked to authenticate himself by username and password

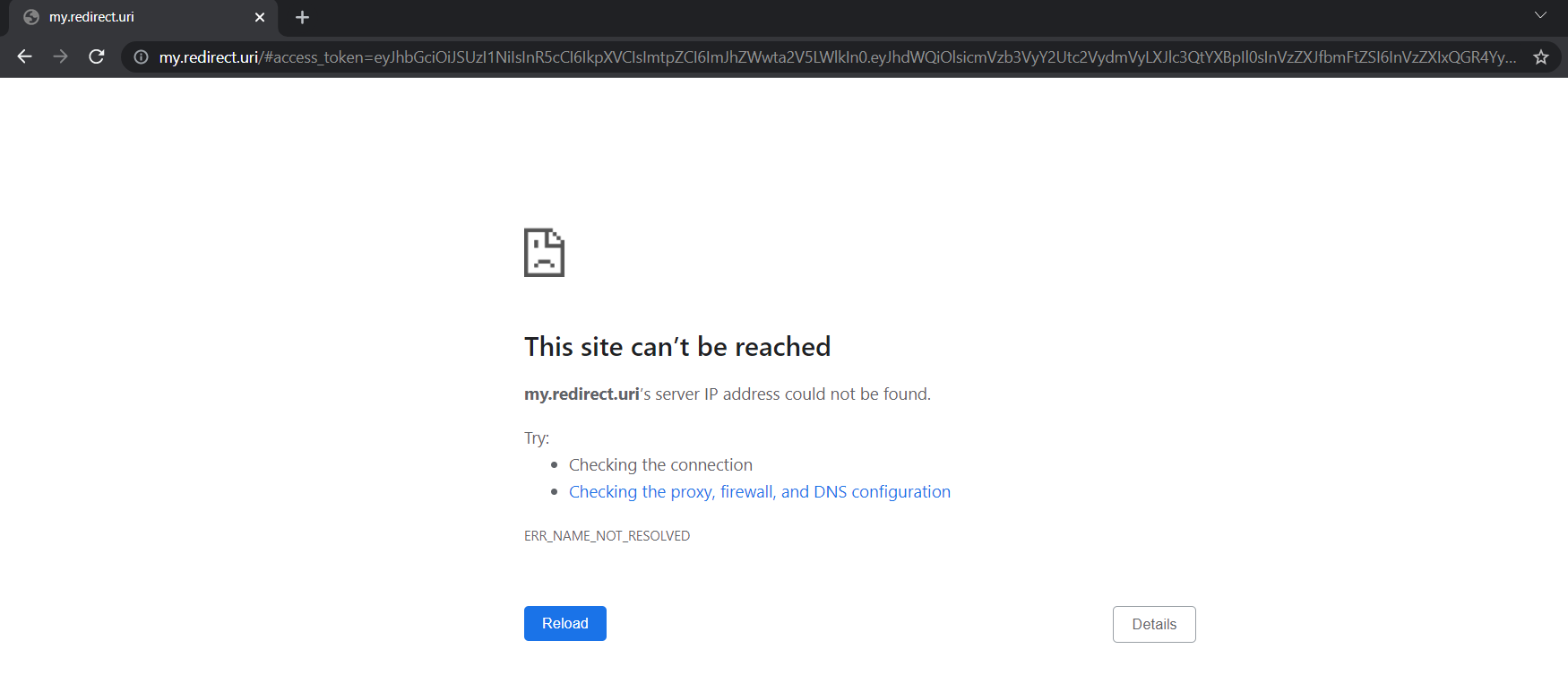


When user puts his credentials correctly, he will be asked to give his consent



Once user approves, he will be redirected to the configured redirect uri with the access token in the url

http://my.redirect.uri/#access\_token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6ImJhZWwta2V5LWlkIn0.eyJhdWQiOlsicmVzb3VyY2Utc2VydmVyLXJlc3QtYXBpIl0sInVzZXJfbmFtZSI6InVzZXIxQGR4Yy5jb20iLCJzY29wZSI6WyJyZWFkIl0sImV4cCI6MTY0NTg3MTIxMywiYXV0aG9yaXRpZXMiOlsiUk9MRV9FTVBMT1lFRSJdLCJqdGkiOiI5NDJmNDJjYy1kODJlLTQ4OGUtOGVjNy0wMTJmYzE0ODAzNjAiLCJjbGllbnRfaWQiOiJ3ZWIifQ.KbRS6zdUi1X3PzLsDQu9j2Nozi7y1thuWI\_d3b0jZQoPoiexG3\_nq83wrZjOmpP-Ub4IS8WtnSoynfld2juaLNRDIYuAO5PiSUJety\_UJVJ8kvgx6BuMXqv0P5jtNi9CX37WA\_Jtg6OdBC81jDI1HhaC5k-L3h9SLsG9x5wxIXPVEyJjeYX3yf427IqrttrlaguaBfNjelwyTz0wqVrDLno3SDfent27cDjgqCpxZMk3cBuH6bQBD8524V\_um84gkSgMk1wVjQvLVpdgOJk\_eck96Ribf83hJytJUkAZtjAZlbLIbsfTTXkRWla08trtSCJviwGH6fqMCI0OZ3nTgg&token\_type=bearer&state=3527&expires\_in=10799&jti=942f42cc-d82e-488e-8ec7-012fc1480360



1. **Password credentials grant flow:** If the client app cannot interact with browser, it will not fit with the above grant flows. In this grant flow the user will share his credentials with the client app. Here the client app will use the use credentials to directly get the access token by making direct call to /oauth/token. Below is an example of this call

curl -k http://web:123123@localhost:8080/oauth/token -d **grant\_type=password** -d **username=user1@dxc.com** -d **password=pass**

This API call looks the same as the call the authorization code grant flow but there is a major difference. Here the user credentials were shared and passed to get the access token. In the

We do not share these user credentials but the grant code that was obtained when the user was redirected to the browser to authenticate himself

1. **Client credential grant flow:** In case where a client application needs to call an API unrelated to a specific user, it can use this grant flow to acquire an access token using its own client Id and secret.

* Once the client app gets the access token from the authorization server, he should pass this access token to the resource owner to access the protected resource. The job of the resource owner is to validate this token that is passed to him to make sure that is valid.

If the access was expired, the client app will use the refresh token to call the authorization server again to get another valid access token without further interaction with the user

* When an OAuth 2.0 client makes a request to the resource server, the resource server needs some way to verify the access token. The OAuth 2.0 core spec doesn’t define a specific method of how the resource server should verify access tokens, just mentions that it requires coordination between the resource and authorization servers. In some cases, especially with small services, both endpoints are part of the same system, and can share token information internally such as in a database. In larger systems where the two endpoints are on different servers, this has led to proprietary and non-standard protocols for communicating between the two servers.

<https://www.oauth.com/oauth2-servers/token-introspection-endpoint/>

**The**[**OAuth 2.0 Token Introspection extension**](https://datatracker.ietf.org/doc/html/rfc7662) defines a protocol that returns information about an access token, intended to be used by resource servers or other internal servers

The token introspection endpoint needs to be able to return information about a token, so you will most likely build it in the same place that the token endpoint lives. The two endpoints need to either share a database, or if you have implemented self-encoded tokens, they will need to share the secret. For example, the below end points that is availed by spring authorization server are examples of these Introspection endpoints:

* + 1. GET oauth/token\_key

curl client-id:client-secret@localhost:8080/**oauth/token\_key**

* + 1. POST /oauth/check\_token

curl client-id:client-secret@localhost:8080/**oauth/check\_token** -d token="vdvdvdvdv"

* An alternative to token introspection is to use a structured token format that is recognized by both the authorization server and resource server. The JWT Profile for OAuth 2.0 Access Tokens is a recent RFC that describes a standardized format for access tokens using JWTs. This enables a resource server to validate access tokens without a network call, by validating the signature and parsing the claims within the structured token itself.
* Which grant flow we should use depends on the client application. Below are links that put some lights on that

<https://auth0.com/docs/get-started/authentication-and-authorization-flow/which-oauth-2-0-flow-should-i-use>

* OpenID Connect (OIDC) is an add-on to OAuth 2.0 that enable client applications to verify the identity of users. OIDC adds an extra token, an ID token, that the client application gets back from the authorization server after a completed grant flow
* OpenID Connect (OIDC): is an identity layer built on top of the OAuth 2.0 protocol and supported by some OAuth 2.0 providers, such as Google and Azure Active Directory. It defines a sign-in flow that enables a client application to authenticate a user, and to obtain information (or "claims") about that user, such as the username, email, and so on. User identity information is encoded in a secure JSON Web Token (JWT), called ID token.
* ID token is encoded as JWT, and it contains number of claims such as ID and email address of the user. Also, the ID token is digitally signed using JSON web signature. This makes it possible for client application to trust the information(claims) in the ID token by validations the digital signature using the public keys from the authorization server
* Optionally, access token can be encoded and signed in the same way as ID token, but it is not mandatory according to the specification
* OpenID Connect defines a discovery mechanism, called OpenID Connect Discovery, where an OpenID server publishes its metadata at a well-known URL, typically

<https://server.com/.well-known/openid-configuration>

* This URL returns a JSON listing of the OpenID/OAuth endpoints such as
  1. initiating grant flow
  2. getting the public keys to verify a digitally signed JWT token (JWKS) (introspection end points)
  3. user-info endpoint

Also, it provides information regarding the supported scopes and claims

* The field names and values are defined in the OpenID Connect Discovery is something standard and it has specs <https://openid.net/specs/openid-connect-discovery-1_0.html>

Now let us have a look at the code and how to implement this using spring boot

**Spring boot Authorization server:**

1. The below dependency should be added to your pom.xml file

<dependency>

<groupId>org.springframework.security.oauth</groupId>

<artifactId>spring-security-oauth2</artifactId>

<version>2.3.8.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework.security.oauth.boot</groupId>

<artifactId>spring-security-oauth2-autoconfigure</artifactId>

<version>2.3.8.RELEASE</version>

</dependency>

<!--JWT support token if needed-->

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-jwt</artifactId>

<version>1.1.1.RELEASE</version>

</dependency>

<!—spring actuator to perform some checks-->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

* spring-security-oauth2 and spring-security-oauth2-autoconfigure are in maintenance mode. Also Note that you need to specify the version for spring-security-oauth2-autoconfigure, since it is not managed by Spring Boot any longer, though it should match Boot’s version anyway.The below link discuss this topic:

<https://stackoverflow.com/questions/59280271/authorizationserverconfigureradapter-is-deprecated>

Based on the link above, a **new Spring Authorization Server project** (https://github.com/spring-projects/spring-authorization-server) was announced(https://spring.io/blog/2020/04/15/announcing-the-spring-authorization-server).

Also, In Spring Security OAuth 2.0 Roadmap Update they recommend using Keycloak as an open-source implementation of the authorization server. Below links could be useful:

<https://www.baeldung.com/spring-boot-keycloak> (A Quick Guide to Using Keycloak with Spring Boot)

<https://www.baeldung.com/keycloak-embedded-in-spring-boot-app> (Keycloak Embedded in a Spring Boot Application)

1. Annotate your application to be @EnableAuthorizationServer

And that is set. This is what you need to make it run. Also, you will notice the following lines are printed when you start your application



From the console, we can see that spring auto configuration generated random values for us

Using generated security password: f5cb4203-1bcf-46ab-90fb-2258580a3da9

security.oauth2.client.client-id = 75e28575-7509-44e2-919c-6f226a714aec

security.oauth2.client.client-secret = 1ad6a147-1f16-4844-b3cf-9292c6281b1e

And you can use this default values to issue token as below

curl **75e28575-7509-44e2-919c-6f226a714aec:1ad6a147-1f16-4844-b3cf-9292c6281b1e**@**localhost:8080/oauth/token** -d **grant\_type**=password -d **username**=user -d **password**=f5cb4203-1bcf-46ab-90fb-2258580a3da9 -d **scope**=read,write,trust

That is set! This is all what you need but we have the following concerns if we will take this default approach

Notes/concerns:

* 1. these random values will be changed every time you start the application
  2. The basic auth credentials for the /oauth/token endpoint are the client-id and client-secret. The user credentials are the normal Spring Security user details (which default in Spring Boot to “user” and a random password).
  3. The autoconfiguration in **spring-security-oauth2-autoconfigure** dependency and **spring-security-oauth2** are responsible to generate these random settings
  4. We do not have a way to set client scope, client grant, client redirect URI so you will be able to get token for these workflows that does not interact with browser, so this is why we used password grant flow in above
  5. Also, introspection end points are forbidden by default so get 403 when try to access /oauth/token\_key and /oauth/check\_token
  6. Spring Security OAuth retrieves a random string value by default, not JWT-encoded: The result would be something like “

{"access\_token":"zZjE2rLJZB\_kUauZGH\_xkrHmg2o","token\_type":"bearer","refresh\_token":"hYHttMCRuYFHnoT8JFo82ThEiYI","expires\_in":43120,"scope":"read,write,trustcurl"}{"access\_

token":"zZjE2rLJZB\_kUauZGH\_xkrHmg2o","token\_type":"bearer","refresh\_token":"hYHttMCRuYFHnoT8JFo82ThEiYI","expires\_in":43120,"scope":"read,write,trustcurl"}

And the value “zZjE2rLJZB\_kUauZGH\_xkrHmg2o” is not JWT token

**Enhance Spring boot Authorization server:**

In the below points we will discuss how we can enhance the default values that comes from the spring auto configuration of spring security

1. Adjust application.properties

Here we will do some enhancements regarding client details configuration by using application.properties

#Basic authentication settings( user details)

spring.security.user.name=user

spring.security.user.password=pass

# client secret configuration

security.oauth2.client.client-id =client-id

security.oauth2.client.client-secret = client-secret

security.oauth2.client.scope= read,write,trust

security.oauth2.client.grant-type=password,authorization\_code,refresh\_token,implicit

security.oauth2.client.registered-redirect-uri=http://my.redirect.uri

# Allow the access to Introspection end points /oauth/token\_key and /oauth/check\_token

security.oauth2.authorization.token-key-access=permitAll()

security.oauth2.authorization.check-token-access=permitAll()

# use JWT instead of the default string value, if spring-security-jwt was found in class path. If not set and use the default then will get 404 when access curl client-id:client-secret@localhost:8080/oauth/token\_key

security.oauth2.authorization.jwt.key-value=JWTKey@123

* Using application.properties is the easiest way to do the enhancements but you can do the same with more flexibility using the **WebSecurityConfigurerAdapter** and **AuthorizationServerConfigurerAdapter.** [**https://stackoverflow.com/questions/31673143/spring-boot-how-to-configure-spring-security-using-application-proerties**](https://stackoverflow.com/questions/31673143/spring-boot-how-to-configure-spring-security-using-application-proerties)

1. Enable security filter (Adjust WebSecurityConfigurerAdapter)

Also, we can enhance more to support the other grant flows that require browser interaction, we can do this by just enable the authentication security filter to

forward the request ““/oauth/authorize?response\_type=code” to the default login page .

Unfortunately, this cannot be accomplished using settings in application.properties

Just extending this abstract class and annotate it with @Configuration is enough enable the security filter that will forward to the default login page

Below is an example

@Configuration

**public** **class** CustomWebSecurityConfigurerAdapter **extends** WebSecurityConfigurerAdapter {

}

Now Our Authorization server supports the following:

1. /oauth/ authorize?response\_type with different grant flow
2. Introspection Endpoints GET oauth/token\_key and POST /oauth/check\_token
3. Our token is JWT

Now the below URL should be running

1. curl client-id:client-secret@localhost:8080/oauth/token -d grant\_type=password -d username=user -d password=pass -d scope=read
2. curl client-id:client-secret@localhost:8080/oauth/token\_key
3. curl client-id:client-secret@localhost:8080/oauth/check\_token -d token="eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2NDY2MDc2ODMsInVzZXJfbmFtZSI6InVzZXIiLCJqdGkiOiJtVXlOWG1NOUxpMV9XcWtIeWoyNDVuWE04MVEiLCJjbGllbnRfaWQiOiJjbGllbnQtaWQiLCJzY29wZSI6WyJyZWFkIl19.1Mu51ZdTMv0fnNjZj8dC3edDVjcXyTFeUZ0uu3KdbMI"
4. browser grant flow using the below

<http://localhost:8080/oauth/authorize?response_type=code&client_id=client-id&redirect_uri=http://my.redirect.uri&scope=read&state=3527>

<http://localhost:8080/oauth/authorize?response_type=token&client_id=client-id&redirect_uri=http://my.redirect.uri&scope=read&state=3527>

Now we are in a good shape but as mentioned we need to cover two important classes that can be used to enhance and configure spring security and authorization server. These are the WebSecurityConfigurerAdapter and **AuthorizationServerConfigurerAdapter**

**Enhance Spring boot Authorization using** **WebSecurityConfigurerAdapter**

* WebSecurityConfigurerAdapter is a convenience class that allows customization to both WebSecurity and HttpSecurity
* <https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/configuration/WebSecurityConfigurerAdapter.html>
* WebSecurityConfigurerAdapter is an abstract class, and it has default implementation
* WebSecurityConfigurerAdapter is not for authorization sever but it is for spring security
* WebSecurityConfigurerAdapter lead spring security module to create security filter(s) that applied to the application.
* Many configuration can be done using application.properties as mentioned above but the java configuration is WebSecurityConfigurerAdapter class gives you more flexibility and more control
* There are many methods in this class with default implementation that can be overwritten but the main important methods
  1. **protected** void configure (AuthenticationManagerBuilder auth) throws Exception

This method is used by the default implementation of authenticationManager() method to attempt to obtain an AuthenticationManager.

If overridden, the AuthenticationManagerBuilder should be used to specify the AuthenticationManager which can be built using different methods such as

**jdbc(auth.inMemoryAuthentication())** **, in memory(auth.inMemoryAuthentication())** or using **UserDetailsService(auth.userDetailsService())**

The result of this method is creating the **AuthenticationManager** bean which can be exposed later as bean using authenticationManagerBean() method as below

authenticationManagerBean()-🡪 authenticationManager()->configure(AuthenticationManagerBuilder auth)

@Bean

@Override

* 1. **public** AuthenticationManager authenticationManagerBean() **throws** Exception {

**return** **super**.authenticationManagerBean();

}

Override this method to expose the AuthenticationManager from configure (AuthenticationManagerBuilder) to be exposed as a Bean

As of above this method calls internally the configure (AuthenticationManagerBuilder auth) mentioned above

* 1. **protected** **void** configure(HttpSecurity http) **throws** Exception

General use of WebSecurity ignoring() method omits Spring Security and none of Spring Security’s features will be available. WebSecurity is based above HttpSecurity.

configure (WebSecurity) is used for configuration settings that impact global security (ignore resources, set debug mode, reject requests by implementing a custom firewall definition). For example

@Override

**public** **void** configure(WebSecurity web) **throws** Exception {

// Disable all security settings for actuator end points

web.ignoring()

.antMatchers(HttpMethod.***OPTIONS***)

.antMatchers("/actuator/\*\*");

}

As of above we ignored the URL <http://localhost:8080/actuator> from any security constraints including the login form

configure (WebSecurity web) Endpoint used in this method ignores the spring security filters, security features (secure headers, csrf protection etc) are also ignored and no security context will be set and cannot protect endpoints for Cross-Site Scripting, XSS attacks, content-sniffing.

* 1. **public** void configure(HttpSecurity http) throws Exception

configure(HttpSecurity) allows configuration of web-based security at a resource level, based on a selection match and here you can enable/disable different security options and user rules

The default configuration is:

http.authorizeRequests().anyRequest().authenticated().and().formLogin().and().httpBasic();

Below example is an example and it is equivalent to what we did in the previous point

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.authorizeRequests()

.antMatchers(HttpMethod.***OPTIONS***,"/actuator/\*\*").permitAll()

.and().formLogin().and().httpBasic();

}

More about the diff between the two configure methods can be found in the link below

<https://stackoverflow.com/questions/56388865/spring-security-configuration-httpsecurity-vs-websecurity>

1. Enhance authentication

* To use the database to carry user information first we need to add the below dependency

<!-- mysql database driver -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<!-- spring data for DB access -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

* Add the spring data source configuration to tell spring security where he can look at the tables

######DB configuration

spring.jpa.hibernate.ddl-auto=none

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

spring.datasource.url=jdbc:mysql://localhost:3306/oauthdb

spring.datasource.username=mydxc

spring.datasource.password=mydxc

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

logging.level.org.hibernate=DEBUG

* Remove all use details settings in the application.properties

#Basic authentication settings

#spring.security.user.name=user1

#spring.security.user.password=pass1

* What you need to do is to extend WebSecurityConfigurerAdapter and annotate it with @configuration

@Configuration

**public** **class** CustomWebSecurityConfigurerAdapter **extends** WebSecurityConfigurerAdapter{

**//**

**}**

* Based on <https://docs.spring.io/spring-security/site/docs/4.2.x/reference/html/appendix-schema.html> there is a default DB scheme that spring security expects. The standard JDBC implementation of the **UserDetailsService** (JdbcDaoImpl) requires tables to load the password, account status (enabled or disabled) and a list of authorities (roles) for the user. You will need to adjust this schema to match the database dialect you are using.
* If you will use the jdbc authentication, you have deleted all user details settings in application properties then add the below

// autowire the datasource

@Autowired

DataSource ds;

// overwrite configure with the injected AuthenticationManagerBuilder

@Override

@Autowired

**protected** **void** configure (AuthenticationManagerBuilder auth) **throws** Exception {

//auth.inMemoryAuthentication().withUser("usr").password("{noop}pass").roles("ROLE\_USER");

//auth.userDetailsService(userDetailsService);

auth.jdbcAuthentication().dataSource(ds);

}

Notes:

Here the configure method with the autowired (**AuthenticationManagerBuilder auth)** which can be used in this configure method todo **in-memory** authentication, **jdbc** authentication or even custom **userDetailsService**

Here we did not use any password encoder bean, so all password values value should be preceded with “{noop}”. For example, “pass” should be saved “{noop}pass”

1. Using password encoder:

* It is highly recommended to encrypt passwords if it is stored into database and avoid saving plain text into database. You need to create PasswordEncoder bean and use it in AuthenticationManagerBuilder auth. Below is an example:

@Bean

**public** PasswordEncoder passwordEncoder() {

PasswordEncoder encoder = PasswordEncoderFactories.*createDelegatingPasswordEncoder*();

**return** encoder;

}

Most probably you can create this bean in the main application config class Then you case use it later as below

@Autowired

DataSource ds;

@Autowired

PasswordEncoder passwordEncoder;

// And then update the configure method to use this passwordEncoder

@Override

@Autowired

**protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {

auth.jdbcAuthentication().dataSource(ds)

.passwordEncoder(passwordEncoder);

}

Bcrypt is the default password encoder of spring security and the password value should be precerded with {bcrypt} for example “pass” will be saved into database as {bcrypt}$2a$10$z5e/19vUBUHF8AKGKREFL.5MD0RiIC3MzB2m.G0GIMAIhdZxs8bpe

* org.springframework.security.core.userdetails.UserDetailsService is an interface ,its main purpose is to abstract how and where spring security can find the user details. Based on this fact you can provide different implementations to get the user details from database or files or external systems etc
* For example, If you have your own DB scheme for user details, you can build your custom queries in a repository class then use it in a new service class that implements org.springframework.security.core.userdetails.UserDetailsService and assign it to AuthenticationManagerBuilder auth as below

@Autowired

**private** UserDetailsService userDetailsService;

@Bean

**public** PasswordEncoder passwordEncoder() {

PasswordEncoder encoder = PasswordEncoderFactories.*createDelegatingPasswordEncoder*();

**return** encoder;

}

@Override

@Autowired

**protected** **void** configure (AuthenticationManagerBuilder auth) **throws** Exception {

auth.userDetailsService(userDetailsService)

.passwordEncoder(passwordEncoder);

}

Above we did use neither @Autowired DataSource ds nor auth.jdbcAuthentication().dataSource(ds), the custom UserDetailsService should encapsulate all of these details

* Many important methods can be overridden to customize application security. All of them are explained in the <https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/configuration/WebSecurityConfigurerAdapter.html#authenticationManagerBean()>
* The most relevant methods in WebsecurityConfigurerAdapter which commonly overridden to custom your security settings are the below:
  1. **public** **void** configure(WebSecurity web) **throws** Exception
  2. **protected** **void** configure(HttpSecurity http) **throws** Exception
  3. @Autowired

**protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception

* 1. **public** AuthenticationManager authenticationManagerBean() **throws** Exception
* Based on the configuration we did above the below URL should :

**Getting token key**

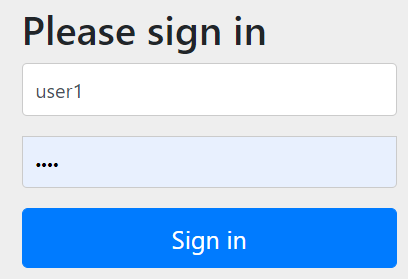
curl client-id:client-secret@localhost:8080/oauth/token\_key

{"alg":"HMACSHA256","value":"JWTKey@123"}

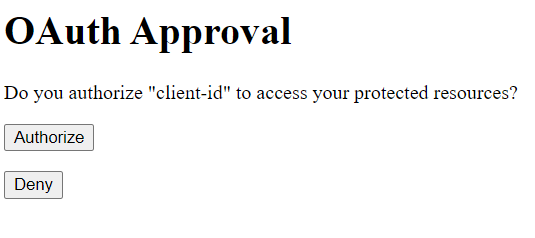
**Getting token(grant\_type=authorization\_code From browser)**

http://localhost:8080/oauth/authorize?response\_type=code&client\_id=client-id&redirect\_uri=http://my.redirect.uri&scope=read&state=3527

Then user is asked to login http://localhost:8080/login



Then user is asked to approve http://localhost:8080/oauth/authorize?response\_type=code&client\_id=client-id&redirect\_uri=http://my.redirect.uri&scope=read&state=3527



Then user will be forwarded with code to <http://my.redirect.uri/?code=xaNa2y&state=3527>

Finally getting token

curl -k http://client-id:client-secret@localhost:8080/oauth/token -d grant\_type=authorization\_code -d redirect\_uri=http://my.redirect.uri -d code=xaNa2y

{"access\_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2NTE3ODUyOTcsInVzZXJfbmFtZSI6InVzZXIxIiwiYXV0aG9yaXRpZXMiOlsiUk9MRV9VU0VSIl0sImp0aSI6ImJQc1ZNandIeXdDSTlfdWx3ZXBvWWxiNkZTdyIsImNsaWVudF9pZCI6ImNsaWVudC1pZCIsInNjb3BlIjpbInJlYWQiXX0.K4ELNaRoOAV5koQhvsvmdBoh9ENyK0D6BBDJeT1H7No","token\_type":"bearer","refresh\_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX25hbWUiOiJ1c2VyMSIsInNjb3BlIjpbInJlYWQiXSwiYXRpIjoiYlBzVk1qd0h5d0NJOV91bHdlcG9ZbGI2RlN3IiwiZXhwIjoxNjU0MzM0MDk3LCJhdXRob3JpdGllcyI6WyJST0xFX1VTRVIiXSwianRpIjoiaVRIUEhySXFkcmhsS2hfZ0dmT21BM3M1MDRvIiwiY2xpZW50X2lkIjoiY2xpZW50LWlkIn0.9A38cAvKgnf-Frd-TXHAsmWt6LcM6ZzvOLVpTkqnXRw","expires\_in":43199,"scope":"read","jti":"bPsVMjwHywCI9\_ulwepoYlb6FSw"}

**Getting token(grant\_type=refresh\_token)**

curl -k http://client-id:client-secret@localhost:8080/oauth/token -d grant\_type=refresh\_token -d redirect\_uri=http://my.redirect.uri -d refresh\_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX25hbWUiOiJ1c2VyMSIsInNjb3BlIjpbInJlYWQiXSwiYXRpIjoiYlBzVk1qd0h5d0NJOV91bHdlcG9ZbGI2RlN3IiwiZXhwIjoxNjU0MzM0MDk3LCJhdXRob3JpdGllcyI6WyJST0xFX1VTRVIiXSwianRpIjoiaVRIUEhySXFkcmhsS2hfZ0dmT21BM3M1MDRvIiwiY2xpZW50X2lkIjoiY2xpZW50LWlkIn0.9A38cAvKgnf-Frd-TXHAsmWt6LcM6ZzvOLVpTkqnXRw

{"access\_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2NTE3ODc1NDIsInVzZXJfbmFtZSI6InVzZXIxIiwiYXV0aG9yaXRpZXMiOlsiUk9MRV9VU0VSIl0sImp0aSI6IlctUGR3UjkyYWVrZkd1aUR

SczlSbUZxam1zZyIsImNsaWVudF9pZCI6ImNsaWVudC1pZCIsInNjb3BlIjpbInJlYWQiXX0.fj24y\_1McI4fzdAhgzsp4DTkz11Mky1-shhwREvZNWg","token\_type":"bearer","refresh\_token":"eyJhbGciOiJIUzI1

NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX25hbWUiOiJ1c2VyMSIsInNjb3BlIjpbInJlYWQiXSwiYXRpIjoiVy1QZHdSOTJhZWtmR3VpRFJzOVJtRnFqbXNnIiwiZXhwIjoxNjU0MzM0MDk3LCJhdXRob3JpdGllcyI6WyJST0xFX1VT

RVIiXSwianRpIjoiaVRIUEhySXFkcmhsS2hfZ0dmT21BM3M1MDRvIiwiY2xpZW50X2lkIjoiY2xpZW50LWlkIn0.GmfLv-Lt5o7-BEuYzM6xsYd9l457YEyPORnBZBlS0K0","expires\_in":43199,"scope":"read","jti":

"W-PdwR92aekfGuiDRs9RmFqjmsg"}

**Check token**

curl client-id:client-secret@localhost:8080/oauth/check\_token -d token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2NTE3ODUyOTcsInVzZXJfbmFtZSI6InVzZXIxIiwiYXV0aG9yaXRpZXMiOlsiUk9MRV9VU0VSIl0sImp0aSI6ImJQc1ZNandIeXdDSTlfdWx3ZXBvWWxiNkZTdyIsImNsaWVudF9pZCI6ImNsaWVudC1pZCIsInNjb3BlIjpbInJlYWQiXX0.K4ELNaRoOAV5koQhvsvmdBoh9ENyK0D6BBDJeT1H7No

{"user\_name":"user1","scope":["read"],"active":true,"exp":1651785297,"authorities":["ROLE\_USER"],"jti":"bPsVMjwHywCI9\_ulwepoYlb6FSw","client\_id":"client-id"}

**Enhance Spring boot Authorization using** **AuthorizationServerConfigurerAdapter**

* AuthorizationServerConfigurerAdapter is the class that can be used to customize and configure your spring authorization server
* This use of this class is different than the WebSecurityConfigurerAdapter. It is only for the spring authorization server and the WebSecurityConfigurerAdapter is for general security configuration
* AuthorizationServerConfigurerAdapter is used by the same way as the WebSecurityConfigurerAdapter, you need to create a class that extends it and start to override its methods

@Configuration

**public** **class** CustomAuthorizationServerConfiguration **extends** AuthorizationServerConfigurerAdapter {

}

* As WebSecurityConfigurerAdapter, the AuthorizationServerConfigurerAdapter has some common configuration methods that could be overwritten to customize your settings. These are highlighted in this link <https://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/config/annotation/web/configuration/AuthorizationServerConfigurerAdapter.html>

1. Enhance Client details

Instead of using static values in application.properties, we will extends the AuthorizationServerConfigurerAdapter and configure it to get the client configuration from the DB as below:

// autowire the datasource

@Autowired

DataSource ds;

// autowire the password encoder

@Autowired

PasswordEncoder passwordEncoder;

/\*\*

\* Client details configuration could be in memory or DB

\* But it is implemented using DB

\*/

@Override

**public** **void** configure(~~ClientDetailsServiceConfigurer~~ clients) **throws** Exception

{

clients.~~jdbc~~(ds).~~passwordEncoder~~(passwordEncoder);

/\* if you want in-memory configuration

clients.inMemory().withClient("client").authorizedGrantTypes("password",

"code", "authorization\_code", "implicit", "refresh\_token",

"check\_token").redirectUris("http://my.redirect.uri")

.secret("{noop}secret").scopes("all");

\*/

}

The code is similar and has the same concept as we did above for WebsecurityConfigurerAdapter. We used the data source bean and the password encoder bean to implement getting the client details (client id& secrets) from the database

And now we will comment all client details configuration in the application.properties

# client secret configuration

#security.oauth2.client.client-id =client-id

#security.oauth2.client.client-secret = client-secret

#security.oauth2.client.scope=read,write,trust

#security.oauth2.client.grant-type=password,authorization\_code,refresh\_token,implicit

#security.oauth2.client.registered-redirect-uri=http://my.redirect.uri

#security.oauth2.authorization.token-key-access=permitAll()

#security.oauth2.authorization.check-token-access=permitAll()

#security.oauth2.authorization.jwt.key-value=JWTKey@123

Note that once you declared your custom AuthorizationServerConfigurerAdapter java class, all related settings in application properties will be ignored even if they are not commented such as the access permission of token\_key and check\_token and JWT token conversion. Based on this fact, you must implement all these settings in your custom AuthorizationServerConfigurerAdapter .

1. Integrate Client details with the authentication(continued)

As above you will be able to get token using browser scenario but if try the password scenario as below:

curl -k http://web:123123@localhost:8080/oauth/token -d grant\_type=password -d username=user1 -d password=pass

you will get

{"error":"unsupported\_grant\_type","error\_description":"Unsupported grant type"}

That is why you need to declare the AuthenticationManagerBean in your custom CustomWebSecurityConfigurerAdapter and autowire it in your custom AuthorizationServerConfiguration

First, expose AuthenticationManager as bean in your CustomWebSecurityConfigurerAdapter as below

@Override

@Bean

**public** AuthenticationManager authenticationManagerBean() **throws** Exception {

**return** **super**.authenticationManagerBean();

}

Second, autowire that bean in CustomAuthorizationServerConfiguration

@Autowired

**private** AuthenticationManager authenticationManager;

Finally, assign this autowired bean to be the AuthenticationManger of authorization server endpoints by overriding the configure(~~AuthorizationServerEndpointsConfigurer~~ endpoints) as below:

@Override

**public** **void** configure(~~AuthorizationServerEndpointsConfigurer~~ endpoints) **throws** Exception {

endpoints.~~authenticationManager~~(authenticationManager);

}

Now you can get the token using below request

curl -k http://web:123123@localhost:8080/oauth/token -d grant\_type=password -d username=user1 -d password=pass

And the response should be something like

{"access\_token":"9M\_oL01Gp7U51SdI\_yNB7L\_5ICY","token\_type":"bearer","refresh\_token":"U2fvAKWu\_G0SgmhNxDoRFyoamvA","expires\_in":9340,"scope":"read"}

1. JWT token settings using symmetric key

Convert from the plain token to JWT token is so simple and can be implemented in two different ways

**First way,** you will determine the signing key that is used to sign the token as in the code below

@Bean

**public** ~~JwtAccessTokenConverter~~ jwtAccessTokenConverter() {

~~JwtAccessTokenConverter~~ converter = **new** ~~JwtAccessTokenConverter~~();

// set the signing key

converter.~~setSigningKey~~("JWTKey@123");

**return** converter;

}

And this is equivalent to

[security.oauth2.authorization.jwt.key-value=JWTKey@123](mailto:security.oauth2.authorization.jwt.key-value=JWTKey@123) in application.properties file

**Second way** is to use the random signing key created by spring default.

@Bean

**public** ~~JwtAccessTokenConverter~~ jwtAccessTokenConverter() {

~~JwtAccessTokenConverter~~ converter = **new** ~~JwtAccessTokenConverter~~();

// Here spring will create a random key

// converter.setSigningKey("JWTKey@123");

**return** converter;

}

**Note**

1. In the second way, spring will create random signing key for you, but you will get a new signing key every time the application is restated
2. You will not be able to call the Introspection end points oauth\check\_token and oauth\token\_key. By default, they are all forbidden and you will get forbidden message from the server later we will learn how to fix that.

curl -k http://web:123123@localhost:8080/oauth/token\_key

{"timestamp":"2022-09-05T11:29:08.090+00:00","status":403,"error":"Forbidden","message":"Forbidden","path":"/oauth/token\_key"}

1. To be able to call to call the Introspection end points. You need to configure this using by overriding configure(~~AuthorizationServerSecurityConfigurer~~ security) as below

// This is java config that equivalent to the below settings in the application.properties

// security.oauth2.authorization.token-key-access=permitAll()

// security.oauth2.authorization.check-token-access=permitAll()

@Override

**public** **void** configure(~~AuthorizationServerSecurityConfigurer~~ security) **throws** Exception {

security.~~checkTokenAccess~~("isAuthenticated()");

security.~~tokenKeyAccess~~("isAuthenticated()");

}

The above java settings are the equivalent way to the below settings in the application.properties:

And now the result should be something as below:

curl -k http://web:123123@localhost:8080/oauth/token\_key

{"alg":"HMACSHA256","value":"JWTKey@123"}

curl -k http://web:123123@localhost:8080/oauth/check\_token -d token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsicmVzb3VyY2Utc2VydmVyLXJlc3QtYXBpIl0sInVzZXJfbmFtZSI6InVzZXIxIiwic2NvcGUiOlsicmVhZCJdLCJleHAiOjE2NjIzODg1MTksImF1dGhvcml0aWVzIjpbIlJPTEVfVVNFUiJdLCJqdGkiOiJPV2QzS3V2T3Yzd0NpS0x3YlVSY2p5RDhiajgiLCJjbGllbnRfaWQiOiJ3ZWIifQ.WQ3igVdylQaFmPPpketfUxcQQeYk1ryBihoy2AW\_AsA -d username=user1 -d password=pass

{"aud":["resource-server-rest-api"],"user\_name":"user1","scope":["read"],"active":true,"exp":1662388519,"authorities":["ROLE\_USER"],"jti":"OWd3KuvOv3wCiKLwbURcjyD8bj8","client\_id":"web"}

1. if you do not set the the signing key as mentioned spring will create a random key, you can see this as below

curl -k http://web:123123@localhost:8080/oauth/token\_key

{"alg":"HMACSHA256","value":"uWOeOE"}

1. Generally speaking, and by default, Spring signs the header and payload using a Message Authentication Code (MAC) approach. We can verify this by analyzing the JWT in one of the many JWT decoder/verifier online tools such as <https://kjur.github.io/jsrsasign/tool/tool_jwtveri.html>

<https://kjur.github.io/jsrsasign/tool/tool_jwtveri.html>

If we decode the JWT we obtained, we'll see that the value of the alg attribute is HS256, which indicates an HMAC-SHA256 algorithm was used to sign the token

1. JWT token settings using asymmetric key

Using symmetric key is not always the best approach for security. Nowdays , asymmetric keys is the PRD approach.

Here we will see how to use asymmetric key for signing the token. Asymmetric keys Cryptography means that there is a pair of keys that are mathematically related. If you use one of them keys to encrypt a message, then the other and only the other key can be used to decrypt this message.

Now we will go to the next level to secure our token and signing key. In this case we need to understand the below concepts:

**JWS** is a specification created by the IETF that describes different cryptographic mechanisms to verify the integrity of data, namely the data in a JSON Web Token (JWT). It defines a JSON structure that contains the necessary information to do so.

It's a key aspect in the widely-used JWT spec since the claims need to be either signed or encrypted in order to be considered effectively secured.

In the first case, the **JWT** is represented as a **JWS**. While if it's encrypted, the JWT will be encoded in a JSON Web Encryption (**JWE**) structure.

The most common scenario when working with OAuth is having just signed JWTs. This is because we don't usually need to “hide” information but simply verify the integrity of the data.

Of course, whether we're handling signed or encrypted JWTs, we need formal guidelines to be able to transmit public keys efficiently. This is the purpose of JWK, a JSON structure that represents a cryptographic key, defined also by the IETF.

Many Authentication providers offer a “JWK Set” endpoint, also defined in the specifications. With it, other applications can find information on public keys to process JWTs.

For instance, a Resource Server uses the kid (Key Id) field present in the JWT to find the correct key in the JWK set.

1. Let us start this step by step. The first step is just to use asymmetric key public and private keys instead of using the single key . you can create your key pair using java utility as follow:

cd $JAVA\_HOME/bin

./keytool -genkeypair -alias bael-oauth-jwt -keyalg RSA -keypass bael-pass -keystore bael-jwt.jks -storepass bael-pass

Notice we used an RSA algorithm here, which is asymmetric. answer the interactive questions and generate the keystore file. At the end , the result would be bael-jwt.jks file which represents your key pair

Note: this ” bael-jwt.jks” is a binary file. That means it can't be filtered please have a look at <https://maven.apache.org/plugins/maven-resources-plugin/examples/filter.html> , or it'll become corrupted.

If we're using Maven, one alternative is to put the text files in a separate folder and configure the pom.xml accordingly:

<**build**>

<**resources**>

<**resource**>

<**directory**>src/main/resources</**directory**> <**filtering**>false</**filtering**>

</**resource**>

<**resource**>

<**directory**>src/main/resources/filtered</**directory**> <**filtering**>true</**filtering**>

</**resource**>

</**resources**>

</**build**>

You can now put the file at src/main/resources.

1. Once you have your bael-jwt.jks at your resource folder .you now ready to replace our syemtric key by this key pair as below

// Here we created keypair as bean

@Bean

**public** KeyPair keyPair() {

ClassPathResource ksFile = **new** ClassPathResource("bael-jwt.jks");

~~KeyStoreKeyFactory~~ ksFactory = **new** ~~KeyStoreKeyFactory~~(ksFile, "bael-pass".toCharArray());

KeyPair keyPair = ksFactory.~~getKeyPair~~("bael-oauth-jwt");

**return** keyPair;

}

// change the jwtAccessTokenConverter to use this key pair

@Bean

**public** ~~JwtAccessTokenConverter~~ jwtAccessTokenConverter() {

~~JwtAccessTokenConverter~~ converter = **new** ~~JwtAccessTokenConverter~~();

converter.~~setKeyPair~~(keyPair());

**return** converter;

}

Note you do not have to create keypair in form of bean(@Bean) if you do not intend to use it later. We can create it when needed as below:

@Bean

**public** ~~JwtAccessTokenConverter~~ jwtAccessTokenConverter() {

~~JwtAccessTokenConverter~~ converter =

**new** ~~JwtAccessTokenConverter~~();

ClassPathResource ksFile =

**new** ClassPathResource("bael-jwt.jks");

~~KeyStoreKeyFactory~~ ksFactory =

**new** ~~KeyStoreKeyFactory~~(ksFile, "bael-pass".toCharArray());

KeyPair keyPair = ksFactory.~~getKeyPair~~("bael-oauth-jwt");

converter.~~setKeyPair~~(keyPair);

**return** converter;

}

1. Now let us see the result of the above change

request

curl -k http://web:123123@localhost:8080/oauth/token -d grant\_type=password -d username=user1 -d password=pass

Response

{"access\_token":"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsicmVzb3VyY2Utc2VydmVyLXJlc3QtYXBpIl0sInVzZXJfbmFtZSI6InVzZXIxIiwic2NvcGUiOlsicmVhZCJdLCJleHAiOjE2NjIzOTcxODUsImF1dGhvcml0aWVzIjpbIlJPTEVfVVNFUiJdLCJqdGkiOiJKU24tU0xISUpBaXd5NTFySGJJT3VHNDYwN3MiLCJjbGllbnRfaWQiOiJ3ZWIifQ.Tn8UqCIODjf8ztvUseLRIf\_-lzS96kgHd2iZwqjqJo9\_-T0TvA75hGFmkUu7WeNpcQFo0X8yeL7K4eZoOUJXnD9F8AmCsuol5MjQdGcG3WYcWi6Lm5fWFE1HBRPd1rk2H12jsThBmbzKH0NuAOIrrGzcwMVOYns2CUNpgzHWtGOovDlgOfgao3MHlX7Q8HN0w7kO6zFWR4ZUxvZP6IuaswGTtIbvg1\_aSNL6yCpfGdV-\_hTrqZq8F\_MKC2EDH8wMg\_cJQuGtkYWyxuYF2xLG\_RMMDyeRXxlwSxqXJt34veJnh5o36eUgdpkPrILhwsKaSjfJ-KWs8pZ65Q5NsioFRA","token\_type":"bearer","refresh\_token":"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsicmVzb3VyY2Utc2VydmVyLXJlc3QtYXBpIl0sInVzZXJfbmFtZSI6InVzZXIxIiwic2NvcGUiOlsicmVhZCJdLCJhdGkiOiJKU24tU0xISUpBaXd5NTFySGJJT3VHNDYwN3MiLCJleHAiOjE2NjQ5NzgzODUsImF1dGhvcml0aWVzIjpbIlJPTEVfVVNFUiJdLCJqdGkiOiJFM0w2ZDdJYWJrdFdHc1NTZExaNWF6N0xFdDQiLCJjbGllbnRfaWQiOiJ3ZWIifQ.pHPKd27pmKp1wYwdJGZUbxNRkBXEMNqo\_adGKwD80l6klMIymbbhScWCgaxtHRj65rU60rmwgu9dczpLqi\_Uj8RctRE\_6qn3vD6vdfJEnPQgp0m8suAEVh9JowOmULwSlCMotPfF6qdkVv5EX2os4W1wOKVDRY7M-QfEHb8ZF3FP7b7wwSJSb1JEVWBQEQwFmHBdMaqHj3s\_5ih1Ow\_pAs8Gfuk1yWVGmXp0Yq1s\_XGriBhN9QZTORy12GIWTOD7gYcEjyxwJfM4CTs\_NN2qqEKkecVsRevR5CaBFARVlEQDr5KtiFHbp2tHVdaKuDBGPznaV2tDy6jLb8FziYgL6w","expires\_in":10799,"scope":"read","jti":"JSn-SLHIJAiwy51rHbIOuG4607s"}

request

curl -k http://web:123123@localhost:8080/oauth/token\_key

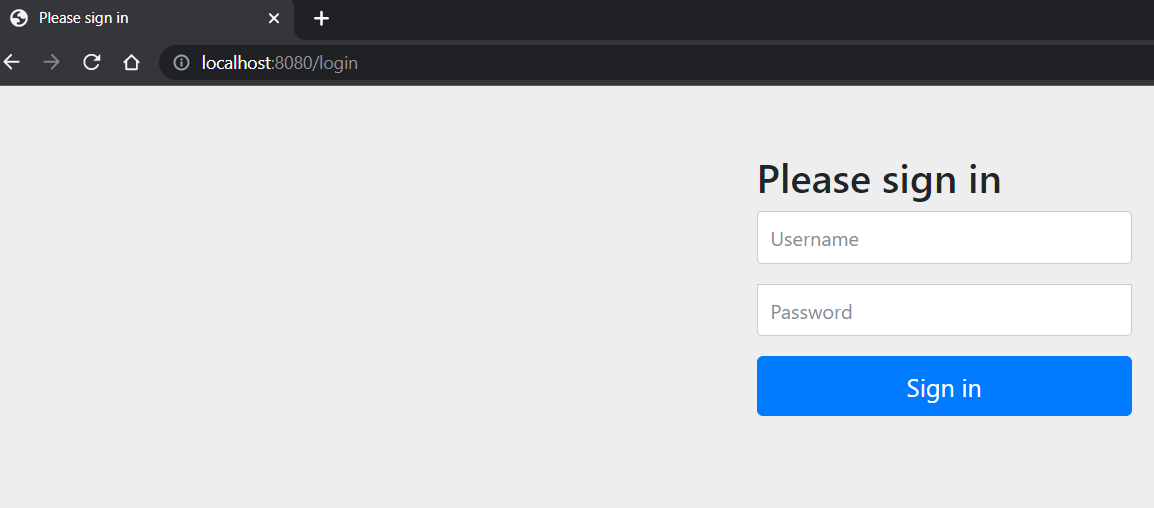
response

{"alg":"SHA256withRSA","value":"-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzmxFNVqHS679BuDGDkJhqs/HLKXPB7w/jyAVgnmag8Hc2PU4FzUONTm2hJYiChZNrGRkpwwjfFcS/c4pMtnsWDKmwvzKtxpvx5IPLU5wIkwOJpkaqZg6cH88VV9al3hqxgFB76emRf0Bp7B2s6kOiTJjjHhsY4PqXdQL5HH7naNuq+ZfL0jVAg1stBpGiN8ygMONiOB9/Ix4B17xbgvmvBuuBBvJBBq6pLOy6Ar0bpXOKgD0fou7IYofwFDXm0xSl0mllP0Ho/GWx5mR+7cncZEmQ0ultDvlXz4EZsqdgNZ9j/KTMi2UfHoxIMnUkDGSsjV5JUAprtTd2uVuAginvQIDAQAB\n-----END PUBLIC KEY-----"}

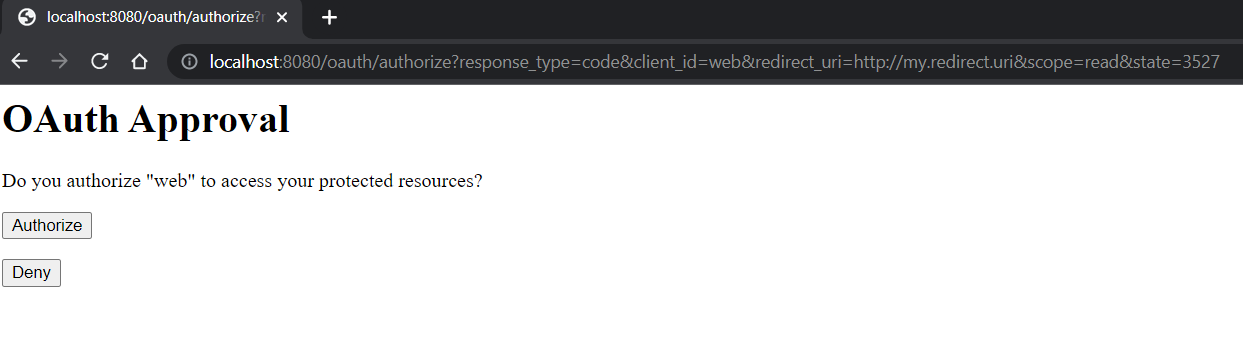
* From the above two request you can notice that our token become longer more complex due to the use of the asymmetric key pais . Also , you can notice that token\_key returns the public key.
* Also, you are able to use the other authentication scenarios to generate the token. For example, the browser based scenario (authorization\_code) as below:

<http://localhost:8080/oauth/authorize?response_type=code&client_id=web&redirect_uri=http://my.redirect.uri&scope=read&state=3527>

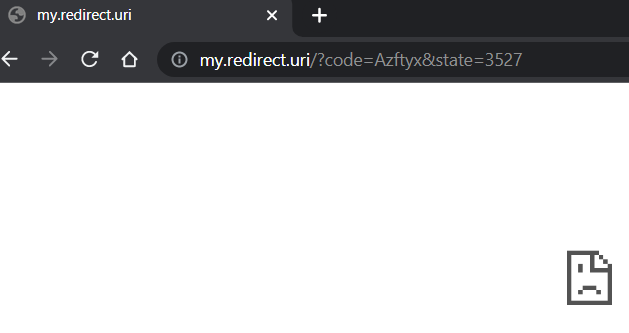
Enter your credentials



Give your consent



pick the grant code(the authorization code) value code=Azftyx



Finally, get the token based on the grant code

Request:

curl -k http://web:123123@localhost:8080/oauth/token -d grant\_type=authorization\_code -d redirect\_uri=http://my.redirect.uri -d code=Azftyx

response:

{"access\_token":"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsicmVzb3VyY2Utc2VydmVyLXJlc3QtYXBpIl0sInVzZXJfbmFtZSI6InVzZXIxIiwic2NvcGUiOlsicmVhZCJdLCJleHAiOjE2NjI0NjQyNzksImF1dGhvcml0aWVzIjpbIlJPTEVfVVNFUiJdLCJqdGkiOiJKem5STTF6Qk9wMjFTaERwUlI0WGZSQy05elUiLCJjbGllbnRfaWQiOiJ3ZWIifQ.IXkXDavGqkoWOODEdXNy\_eo6m4CAFIcNWymPeAF3tBZzq3S73IfzS\_BVvXkNA1qZDgxAq7LNlxmJOTnVu8USCU8I78aEmwPhdfiyTrYHDqelQC8MLEZjseh2Sf4ywMdTWttwInPjDl2sxrS-ceFGiWWWpiC5e9WgwJ63Z-U5o-pW0rXAHJmEee-pc4CugVB7amVudXJhXhyN1iKEL860eoGc4rrzMyPKOcUrRHn9dOSSkU-KDjxJEuKYuPcdLQuUscbEX5DfZpn69SrNh8xdOC9ghu3oxIQG6-VMSkuVZvMAV7KOwnIb4p6RilL53UDsuEPAeeVNwjOlaYPSZgozXQ","token\_type":"bearer","refresh\_token":"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsicmVzb3VyY2Utc2VydmVyLXJlc3QtYXBpIl0sInVzZXJfbmFtZSI6InVzZXIxIiwic2NvcGUiOlsicmVhZCJdLCJhdGkiOiJKem5STTF6Qk9wMjFTaERwUlI0WGZSQy05elUiLCJleHAiOjE2NjUwNDU0NzksImF1dGhvcml0aWVzIjpbIlJPTEVfVVNFUiJdLCJqdGkiOiJpenlvSnFHdUlfTGNMU0JTbXR3NDY1bEhVT2ciLCJjbGllbnRfaWQiOiJ3ZWIifQ.xdLG\_lGSARqdKvJxqFKg0Raa0yK7BODTVqnp4P7Bj2NZWkcUTxpkKZTBLcngnKSbdeJc5YSmvjdWH7FPziMzawnD4FprTDDAPxKEyhvYFeFri7q6rdBMd4MIK9sLrlhtjMscINADE4umPRh-f5Uq8gllX27GDgZo6fhvdwBJm96dV-YTTyYDhSs70H5V9H9rna7-R-jtlL-lsV6NCmNGBsRmjnwEbta4XkxWMWOIt-7wtkgeibJW8x5cbt1CJJjmUO\_yGaNXNyLOaqIbWIXH1Jknxene97PyaDJyNME54cbz\_PwP701-TIKDlDNrsVOtYWHkR0DRXtfQ457Hqi7olw","expires\_in":10799,"scope":"read","jti":"JznRM1zBOp21ShDpRR4XfRC-9zU"}

1. JWT token settings using asymmetric key and JWK set

JWK is the key that is used to sign the JWT.

JWK could be symmetric or asymmetric as we see above.

Your authorization server should avail a way or API(URL) to the client to validate the token. This will be discussed later but one of them is the JWK Set end point.

JWKs are a set of keys shared between different services and are used to verify the JWT token from the authorization server. JWK Set exposes the public keys to all the clients who need to validate signatures that the signing parties use.

This means the JWK Set api would return a list of public keys that are available on the authorization server and in this case each public key is associated by kid (key id)

The validation process in this case (in case of using JWK set end point) is to extract the kid value from the token header then get the corresponding public key from the JWK set endpoint based on this kid value