Spring Security

Application Security

Application security refers to the protective measures built into the software applications to protect ***data***, ***resources*** and ***functionalities*** from *unauthorized access*, *malicious attacks* and other threats.

Organizations building digital products/services where the users will be accessing these services via HTTP protocol ex: Flipkart, Netflix, Youtube, Gmail

These products/services contain two critical resources

1. Customer data
2. Application functionality

The above can be further classified into unsecure and secure features

Insecure features/functionalities

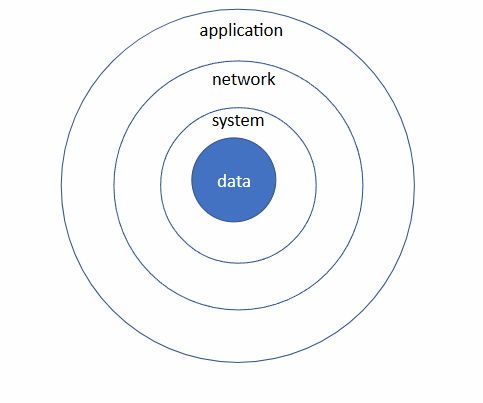
These features do not need security and are often intended for unauthorized access

* User registration
* Login/Logout
* About-us/portfolio/home page
* Contact-us

The features that needs security

* User profiles
* User preferences
* User data/resources

Different levels of security



1. Data Security:
   1. It is at the core of security layer
   2. Data should be stored in encrypted format
   3. Data should be store using a *one-way hash function*
   4. Sensitive information should be stored in the log files
2. System Security
   1. System/server where the data resides should also be secured
   2. *Rule of minimum privilege* should be followed to provide access to the server
   3. All activities should be logged (AWS - Cloudtrail)
   4. Root account should be disabled
   5. All the OS updates should be done periodically to ensure security vulnerabilities

are patched

1. Network Security
   1. Ensure that the firewall rules are configured correctly
   2. Only the necessary ports should be open and from a specific subnet
   3. All the backend non customer facing applications should be provisioned inside private subnets
   4. Follow HTPS protocol and certificates should be installed.
   5. External facing application, certificates from CA should be installed and for internal communication we can use self-signed certificates.
2. Application Security
   1. Two important things to be considered for application security – *Authentication and Authorization*
   2. Authentication refers to checking if you are the same person whom you claim to be.
   3. Authorization refers to checking if you are having the permissions to carry out the tasks.
   4. In HTTP world, the Authentication failure results in Http Status code – ***401***
   5. In HTTP world, the Authorization failure results in Http Status code – ***403***

Implementing security comes with a cost, at the same time, if there is a vulnerability, the advisories will exploit the vulnerability.

If the application is compromised, then the ***TRUST*** of the organization is at stake.

Application Security

There are broadly two ways in taking the approach to implement security:

Customer Centric application (Banking domain, Financial, Healthcare)

1. The applications should be less user friendly
2. Strict measures for login activity

Social Centric applications (Facebook, Google, Twitter)

1. The applications should be easier for the users to use the application
2. Force the users to authenticate for critical tasks (profile updating/ deletion)

Security should be implemented using *Defense in Depth* technique. i.e, even if one of the layers is compromised the next layer should be even more strong. And in case of worst case scenario where all the layers get compromised the data should be meaningless to the attacker.

Security domain

Principal:

A Principal is the entity that can be authenticated by a computer security system. It represents

1. An authenticated identity that can perform actions in the system. i.e, a logged in entity
2. Any entity that can be granted access rights and hold security permissions

Examples:

1. Individual users
2. Processes
3. Services/Applications
4. System/Machines
5. API/Service-Accounts
6. Roles/Groups

Anonymous Entity

1. A Non-logged in entity is referred to as Anonymous. An entity without identity. An anonymous entity after the login process becomes a Principal.

Authentication

Different types of Authentication

1. Username/password
2. Multi-Factor
3. Biometric
4. Single-Sign-On
5. OpenID/OAuth 2.0
6. SAML
7. Certificates
8. Token bases
9. API keys
10. Active directory
11. Hardware token
12. OTP/TOTP
13. Grid
14. Passkeys
15. Captcha

Authorization

1. Role-based Authorization
2. Attribute based Authorization
3. OAuth 2.0
4. Path based Authorization
5. Method level Authorization
6. Policy based Authorization
7. Claim based Authorization
8. Group based Authtorization

Frameworks to implement Application Security

1. JAAS API - Java API for implementing Application Security
   1. Java Authentication and Authorization Service
2. Spring Security
   1. Wide community adoption and support
   2. Natively integrates with Spring ecosystem
   3. Provides extensive supports for different types of authentication providers
   4. Very well documented and maintained
3. Apache Shiro

Spring Security

Spring Security is a comprehensive Application security framework built into Spring Ecosystem.

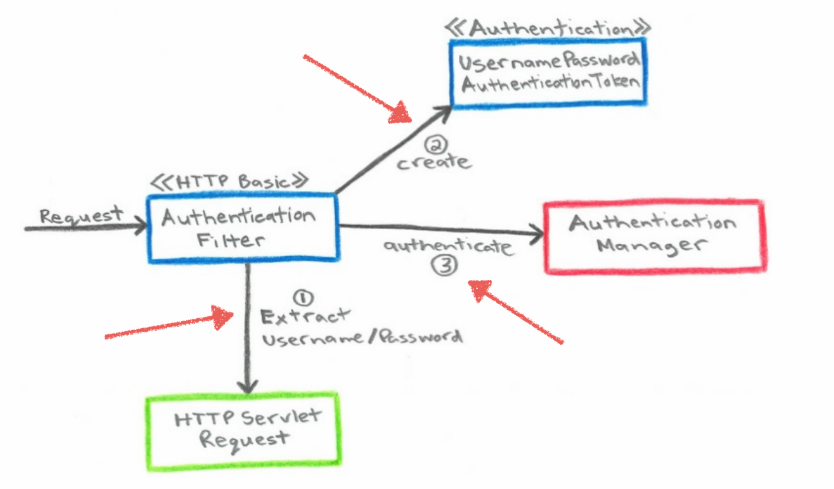
1. It integrates with many different Authentication providers, provides comprehensive solution to implement application security and can be extended.
2. All the heavy lifting is taken care by the Spring Security framework behind the scenes.
3. It lets developers write the application logic and takes care of seamlessly integrating the security of the applications.
4. It also handles application vulnerabilities like CSRF, CORS with less efforts from the developers.
5. It is implemented based on Filter pattern. It intercepts the requests, performs Authentication and Authorization before hitting the endpoints.
6. The application does not even know that the requests are being intercepted.

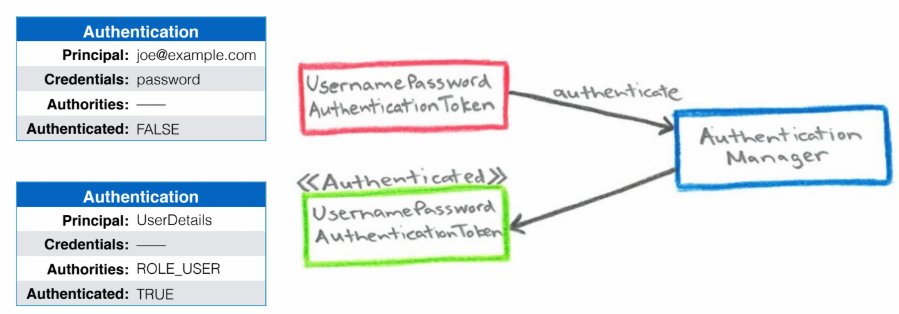
Authentication flow

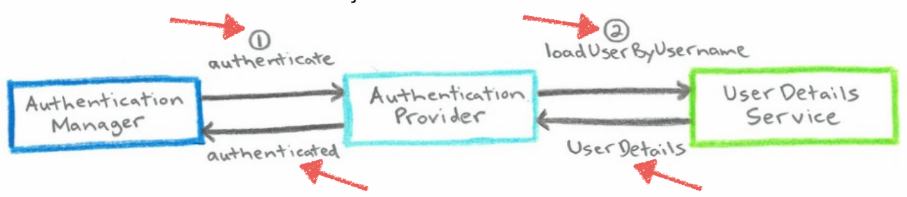
1. Anonymous user login to the application
2. Passes his credentials (*username and password*) using POST request
3. *UsernamePasswordAuthenticationFilter* intercepts the request
4. Extracts the username and password and create a *UsernamePasswordAuthenticationToken* object
5. The Filter then passes the authentication token object to *AuthenicationManager’s authenticate* method
6. The default implementation of AuthenticationManager is *ProviderManager* which provides the implementation for the *authenticate* method
7. The *authenticate* method takes the *Authentication* object and sends back *Authentication* object in response.
8. The ProviderManager calls the authenticate method on the list of AuthenticationProviders. From the list, the *AuthenticationProvider* which supports the type of authentication will be used for authentication.
9. The AuthenticationProvider implementation will implement the authentication and will be specific to the type of Authentication i.e, *DAOAuthenticationProvider* can authenticate against DBs, *OIDCAuthenicationProvider* can authenticate oidc login.

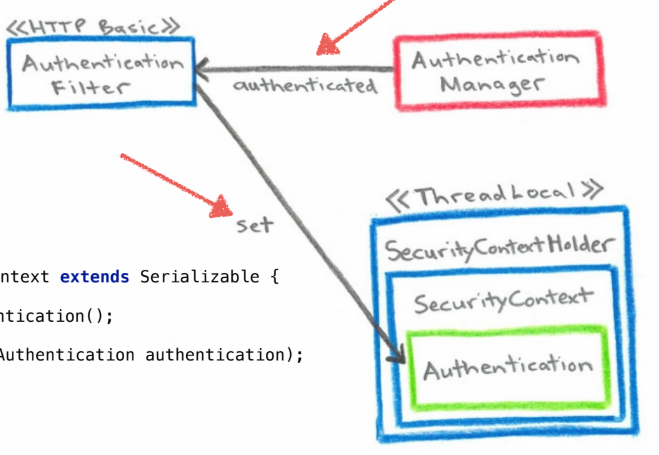
*Specific to Database authentication*

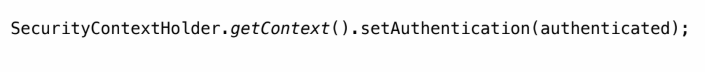
1. In case of Database authentication, we use *DAOAuthenticationProvider* which sends the *username* to *UserDetailsService loadUserByUsername* method.
2. We have to provide custom implementation for the *UserDetailsService loadUserByUsername* method and fetch the User from the DB corresponding to the *username* provided.
3. If the username is not present, we should throw an exception (*UsernameNotFoundException*) else, need to map the domain user to *UserDetails* object using the *adaptor design pattern*.
4. The *DAOAuthenticationProvider* verifies the user, matches the passwords, erases the credentials and return back the *UserDetails* object to the *AuthenticationManager.*
5. The *UsernamePasswordAuthenticationFilter* lastly stores the authentication object inside a *SecurityContext* which is ThreadLocal.
6. The security context will be available within any method where the requests flow.

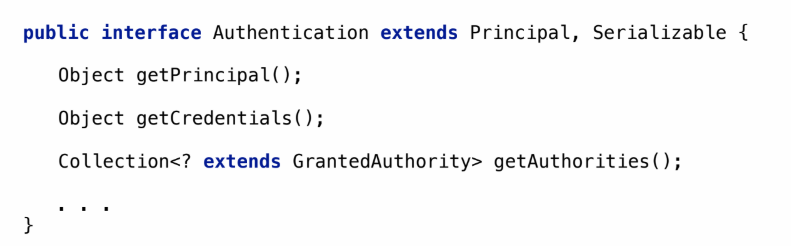
**

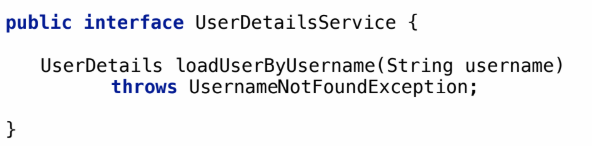
**

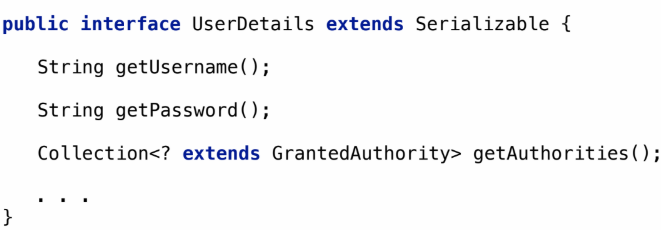
**

**

**

**

**

**

Lab: Setting up a Spring Boot project

Required starter dependencies

* *spring-boot-starter-web*
* *spring-boot-starter-data-jpa*
* *H2*
* *spring-boot-starter-validation*

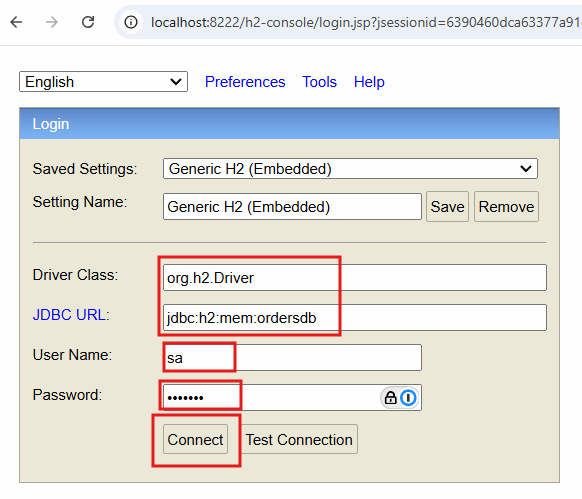
Steps to clone the repository

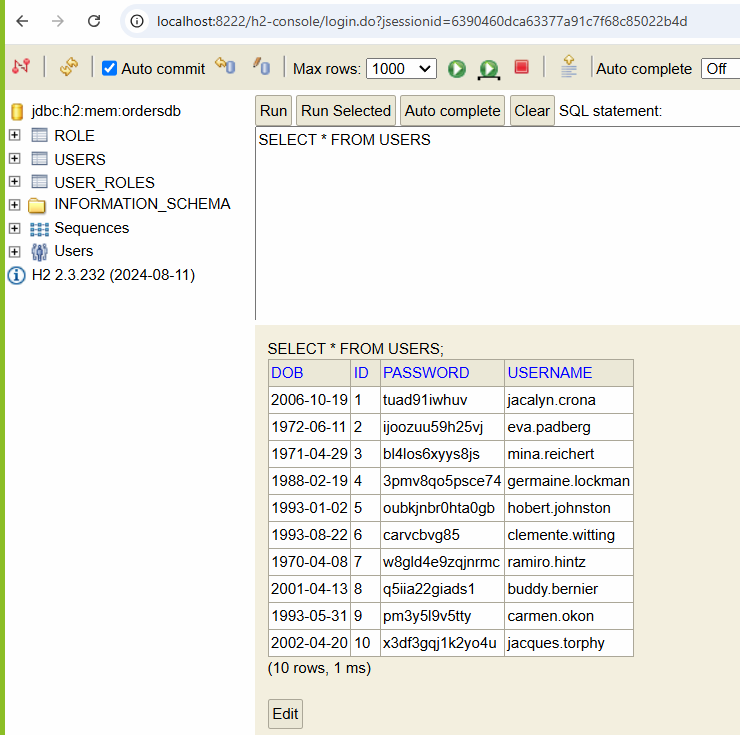
*git clone* [*https://github.com/pradeepkl/orders-application.git*](https://github.com/pradeepkl/orders-application.git)

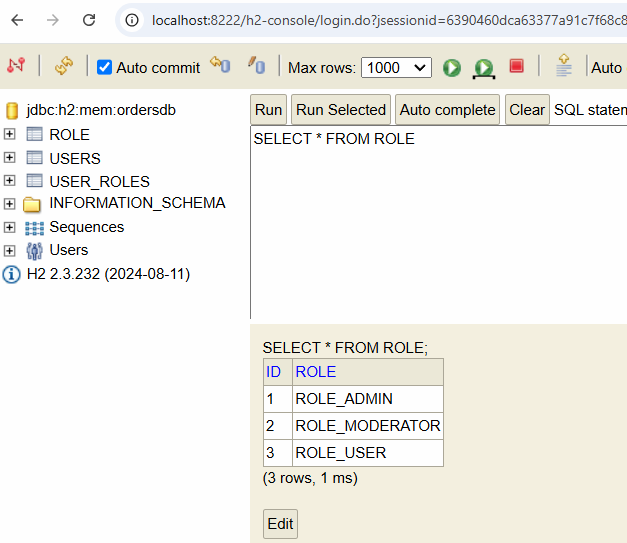
Import the project as maven project inside STS/Eclipse IDE

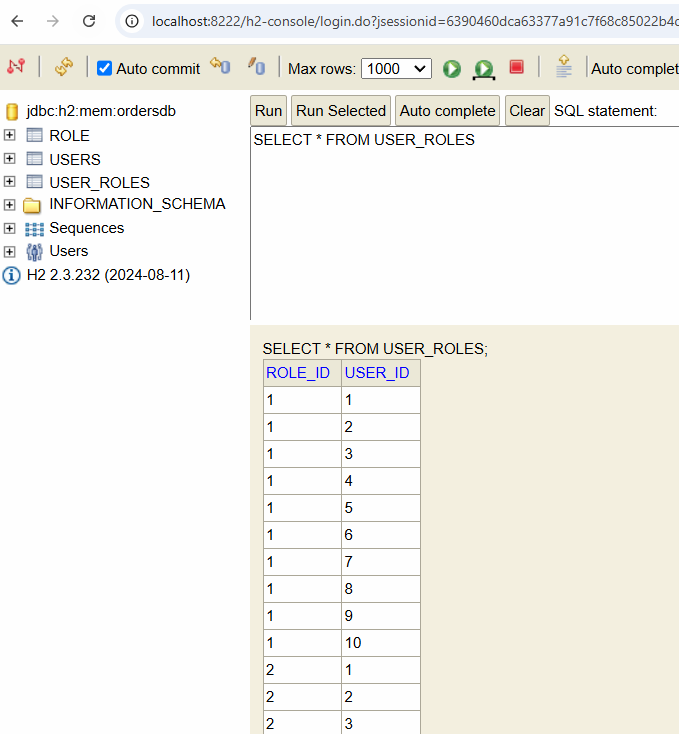
Open project in case of IntelliJ/VSCode editor

Run the application and visit the url in the browser: <http://localhost:8222/h2-console>









Day-2

Building a REST API: CRUD implemenations

Domain model: Order and LineItems

Relationships: One to Many (Bidirectional)

Endpoints for OrderRestController

* *GET - /api/orders*
* *GET - /api/orders/<id>*
* *POST - /api/orders*
* *PUT - /api/orders/<id>*
* *DELETE - /api/orders/<id>*

Order Payload:

{

    "customerName": "Lowell Moore",

    "totalAmount": 130.9,

    "orderDate": "1982-02-03",

    "lineItems": [

        {

            "itemName": "Rustic Granite Bag",

            "price": 18.18,

            "quantity": 2

        },

        {

            "itemName": "Small Copper Gloves",

            "price": 94.54,

            "quantity": 1

        }

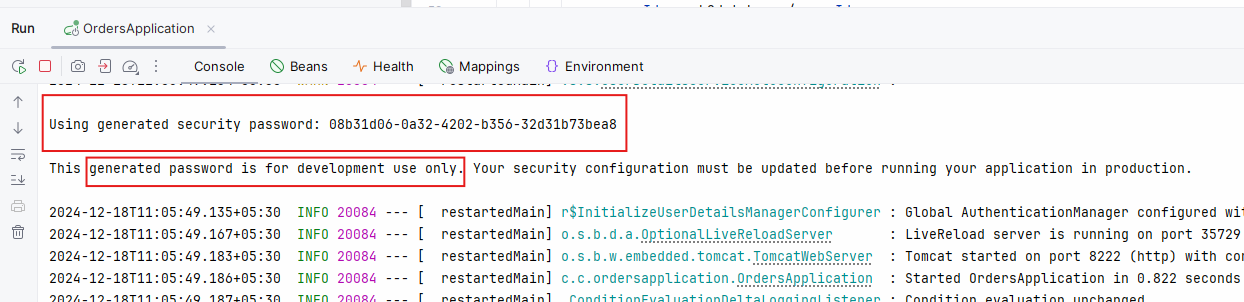
    ]

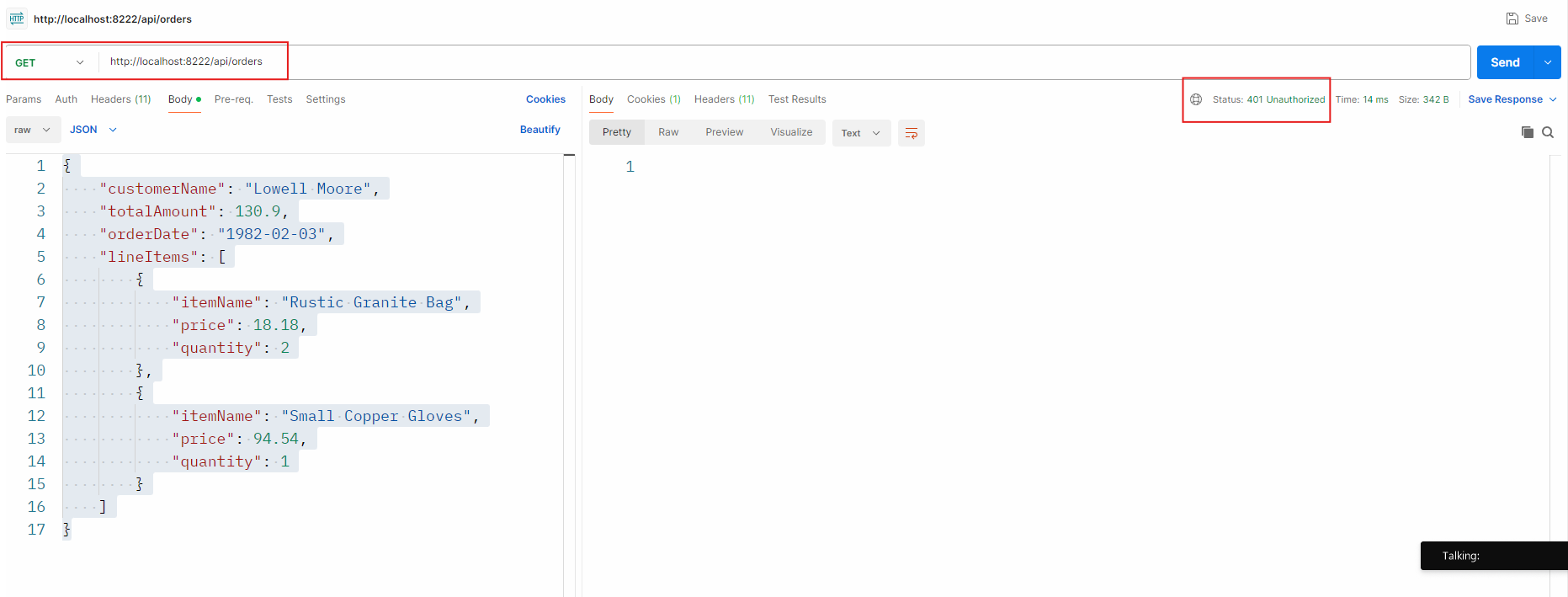
}

Securing the REST API’s using Spring Security (6.4.1)

pom.xml

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>





Spring Security Configuration

@Configuration  
@EnableMethodSecurity  
@RequiredArgsConstructor  
public class SecurityConfiguration {  
  
 private final UserDetailsService userDetailsService;  
 private final PasswordEncoder passwordEncoder;

SecurityFilterChain

@Bean  
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
 *//use the builder patern to configure the HttpSecurity object* http.csrf(csrf -> csrf.disable())  
 .authorizeHttpRequests(  
 authz -> authz  
 .requestMatchers("/h2-console/\*\*", "/login\*\*", "/logout", "/about-us").permitAll()  
 .requestMatchers(HttpMethod.*GET*, "/api/orders/\*\*").hasAnyRole("USER", "ADMIN", "MODERATOR")  
 .requestMatchers(HttpMethod.*POST*, "/api/orders/\*\*").hasAnyRole("ADMIN", "MODERATOR")  
 .requestMatchers(HttpMethod.*DELETE*, "/api/orders/\*\*").hasRole("SUPER\_ADMIN")  
 .anyRequest().fullyAuthenticated())  
 .headers( header -> header.frameOptions(frame -> frame.disable())) *// for h2-console* .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.*STATELESS*))  
 .httpBasic(httpBasic -> {}) *//use basic authentication.i.e username:password base64 encoded* .formLogin(form -> form *// form based authentication* .loginPage("/login")  
 .loginProcessingUrl("/login")  
 .defaultSuccessUrl("/home", true)  
 .failureUrl("/login?error=true")  
 .permitAll());  
 return http.build();  
}

DAOAuthenticationProvider

@Bean  
public DaoAuthenticationProvider authenticationProvider(){  
 DaoAuthenticationProvider provider = new DaoAuthenticationProvider();  
 provider.setUserDetailsService(userDetailsService);  
 provider.setPasswordEncoder(passwordEncoder);  
 return provider;  
}

PasswordEncoder

@Bean  
public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
}

Requirement for building User management

* Onboarding user and managing users is an integral part of secure application development
* There are challenges when building a comprehensive user management module

Challenges in building a user management module

1. User Registration/Signup workflow
2. Secure Storage of Password
3. Password management (complexity, strength, policy)
4. Password rotation
5. Password reset
6. Account lock/unlock

Day-3

Limitations of implementing the user management module

1. Reinventing the solution for a problem statement common to all application.
2. Lack of domain expertise in security domain.
3. Scanning for vulnerabilities and patching the applications to fix the vulnerabilities
4. Need to adapt to newer ways of authentication and authorization.

Solution: Delegate the authentication to a trusted 3rd party solution provider.

OAuth2.0, OpenID connect / SAML

OAuth 2.0

OAuth 2.0 is a delegation based framework to *delegate authentication and authorization* to a trusted provider.

Actors in OAuth 2.0 spec

1. Resource Server: The server in which the functionality (API’s) and sensitive data resides.
2. Resource Owner: The owner of the protected Resource
3. Authorization server: Performs Authentication
4. Client application: Trusted application which performs limited actions on the resource server on behalf of Resource owner

*“OAuth 2.0 is a delegation based protocol where the Resource owner delegates part of his responsibility to a trusted application to perform limited actions on behalf of him for a limited amount of time.”*

Classification of OAuth 2.0 Client application

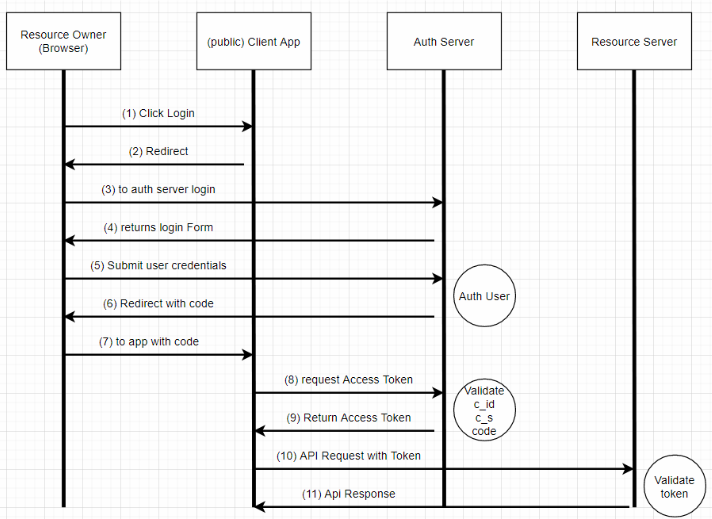
1. Backend Web applications (Java/Python/Ruby/nodejs)
   1. Hosted on servers which can be secured and the communication is over a trusted network also referred to as *back channel*
   2. Can be used to store sensitive credentials safely.
2. Front end applications/SPA/Javascript/Mobile applications
   1. These applications run on users browsers and communicate with backend application on an untrusted network (HTTP/HTTPS)
   2. They are run inside the browsers and can be vulnerable to various malicious scripts, browser extensions.
   3. Cannot store sensitive information on the browser and the security is limited with the OS level security.
3. Machine to machine communication
   1. Hosted on servers and there is no user to perform authentication.
   2. Machine to machine communication is safe since this happens over the back channel

OAuth 2.0 grant flows

These defines the steps involved to authenticate depending on the type of client. The following are the different types of grant flows

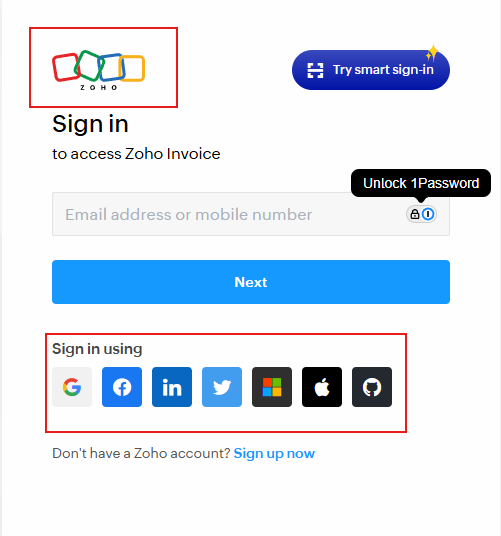
1. Authorization code grant flow: To be used with web backend application (Spring boot/ NodeJs/ Ruby/Python)
2. Proof of Key Exchange (PKCE): To be used with front end applications and mobile apps (React/Angular/Jquery/iOS/Android)
3. Client Credentials: Machine to machine communication, micro-services communication, application to application where there is no user involved.

Authorization code grant flow



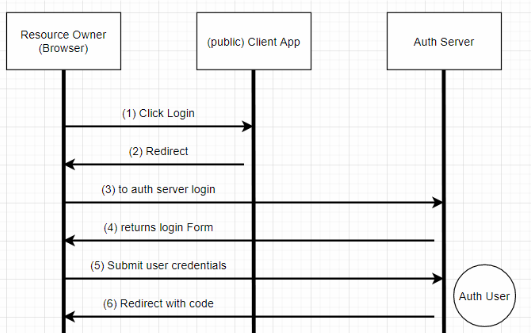
Step-0: Client application onboarding

1. The client application registers with the Authorization server
2. The registration process establishes the trust between the client application and auth server
3. The client application provides the input and get back the response
   1. Input: redirect\_url/callback\_url/ url of the client application to accept response from authorization server
   2. Output:
      1. Client-id – username and can be public, unique for that application within auth server
      2. Client credentials – password and is sensitive. Not to be shared.
   3. Many to many relationships with Auth server



* 1. The authorization server exposes a *public endpoint referred to as well-known/metadata* url
     1. <https://cognito-idp.ap-south-1.amazonaws.com/ap-south-1_zY1QIrdPp/.well-known/openid-configuration>
     2. https://dev-7858070.okta.com/oauth2/default/.well-known/oauth-authorization-server
  2. The metadata url consists of the following key details
     1. Authorization endpoint
     2. Token endpoint
     3. Issuer url

Step - 1

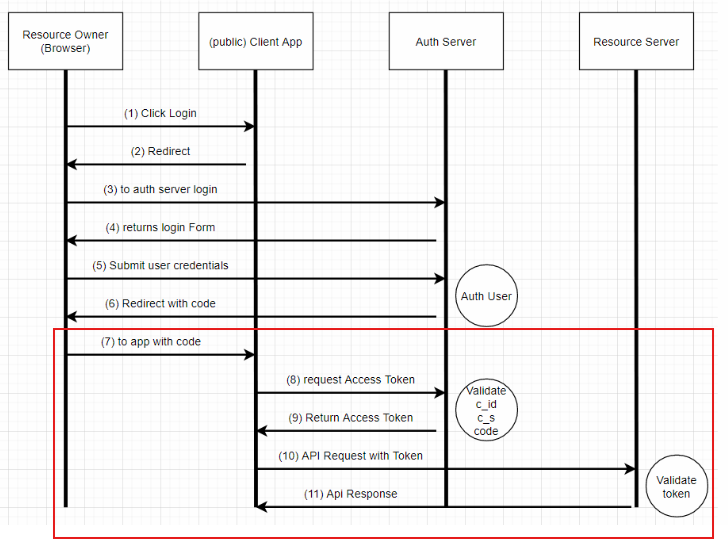


1. The Resource owner tries to login to the client application
2. The client application redirects the user to the auth server.
3. The client application makes use of the authorization endpoint to construct the login button
   1. Authorization endpoint
   2. Query parameters: client\_id, redirect\_url, scope, state, response\_type

<https://dev-7858070.okta.com/oauth2/default/v1/authorize?client_id=&scope=openid&response_type=code&redirect_url=&state=sdfsdfdsfdsf23234324>

1. The user now lands on the login screen of the authorization server.
2. The user will be presented with the login screen and the user enter his credentials.
3. The authorization server will authenticate the user and upon successful authentication, presents the user with a consent form asking the user to either accept or deny the request.
4. Once the user accepts the consent, the authorization server will generate a temporary code (auth\_code) and sends the code in the browser as query parameter to the redirect url for the client application.
5. The user will know the auth\_code, the client application will know the auth\_code and the advisories (hackers) will also know the auth\_code
6. The auth code sent via the browser is not safe since it is visible publically.
7. The auth code will have limited time validity and can be used only once.

Step – 2

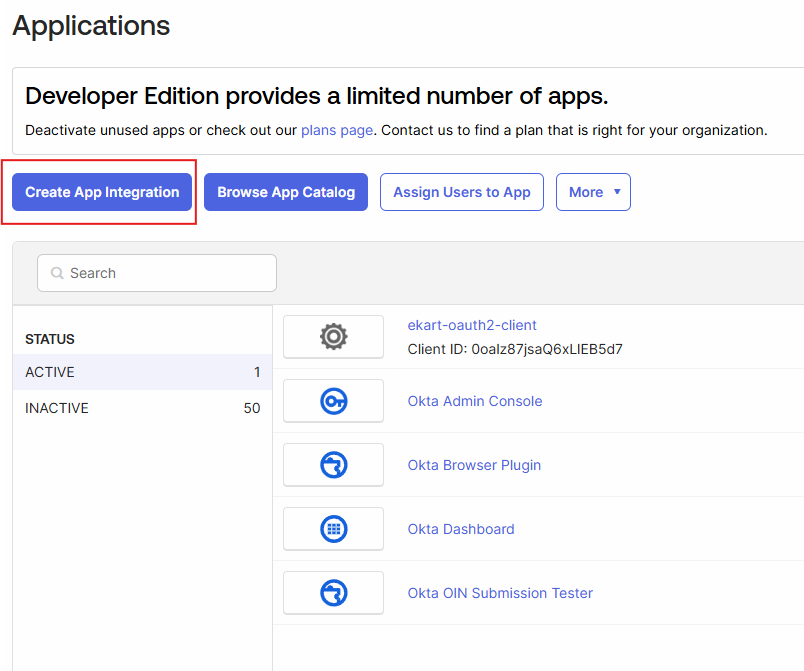


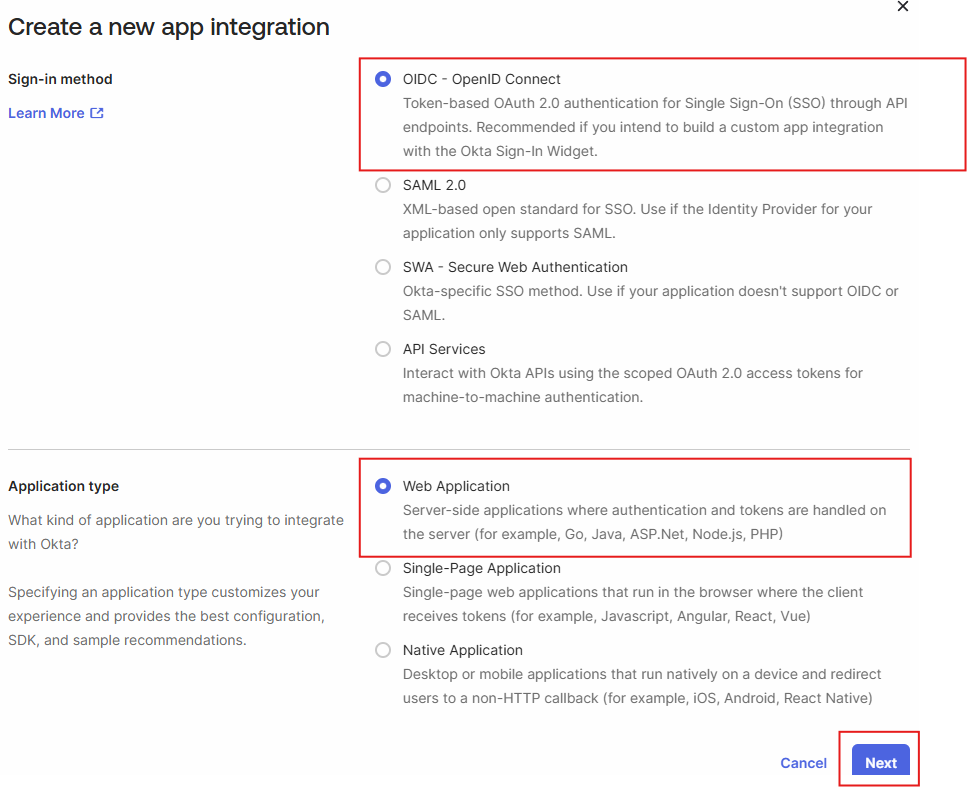
1. The client application received the auth\_code as a query parameter in the redirect url.
2. The client application extracts the auth\_code
3. The client application then makes a ***POST*** request to the Authorization server with the following parameters over the back channel
   1. Redirect uri
   2. Auth\_code
   3. Client id
   4. Client secret
4. The Auth server sends back access\_token, id\_token and refresh\_token as response
5. The access\_token can be either an opaque token or a JWT (Json Web Token)
6. The access token is a short lived token with time validity
7. The client application will use the refresh token to exchange for a new access token without going through the flow once again
8. The refresh token will have longer validity
9. The auth server also issues id\_token which should be used by the client application to get the user information
10. The id token is for the client application to fetch the user details and the access token should be sent to the resource server.

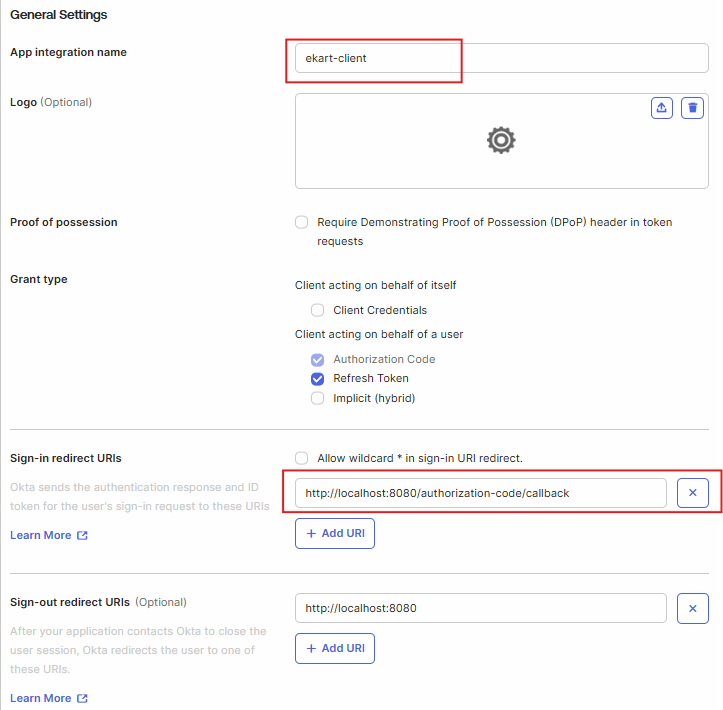
Authorization server

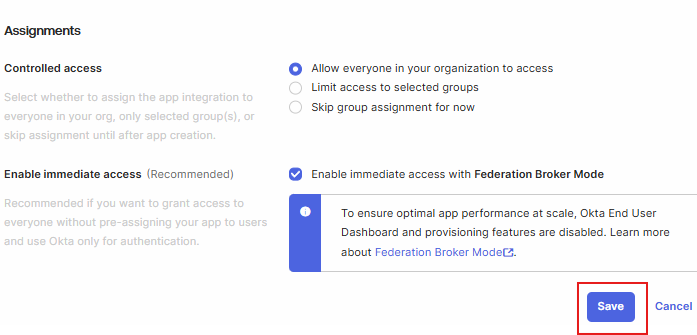
1. The authorization server should implement the OAuth2.0 spec
2. For social applications
   1. Google, facebook, Github
3. For enterprise
   1. Okta, Cognito, Auth0
4. On premises
   1. KeyCloak

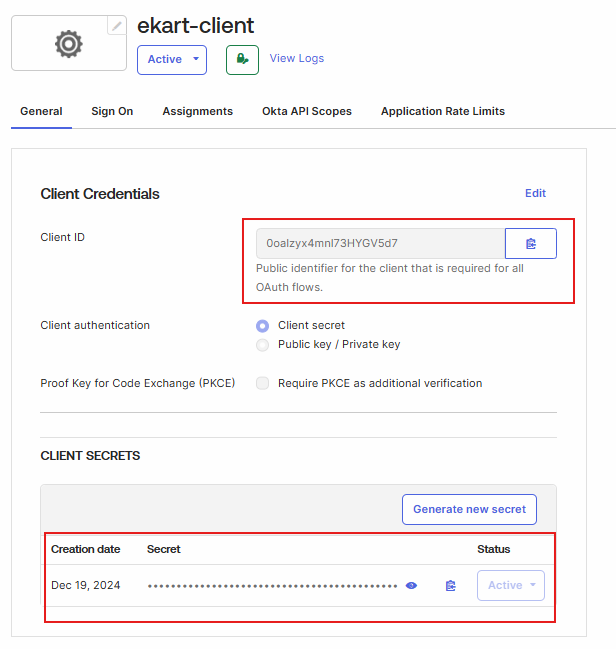
Lab:











Client id: ***0oalzyx4mnl73HYGV5d7***

Client secret: ***BdEtSWALyLPDjqwnGcl2fv1ylo1vmdLkzWDg08b\_zrJm-gPhb5L3Ypo2AQZS1ifi***

******

***Authorization endpoint –***

*Query parameters*

*client\_id -* ***0oalzyx4mnl73HYGV5d7***

*response\_type - code - fixed*

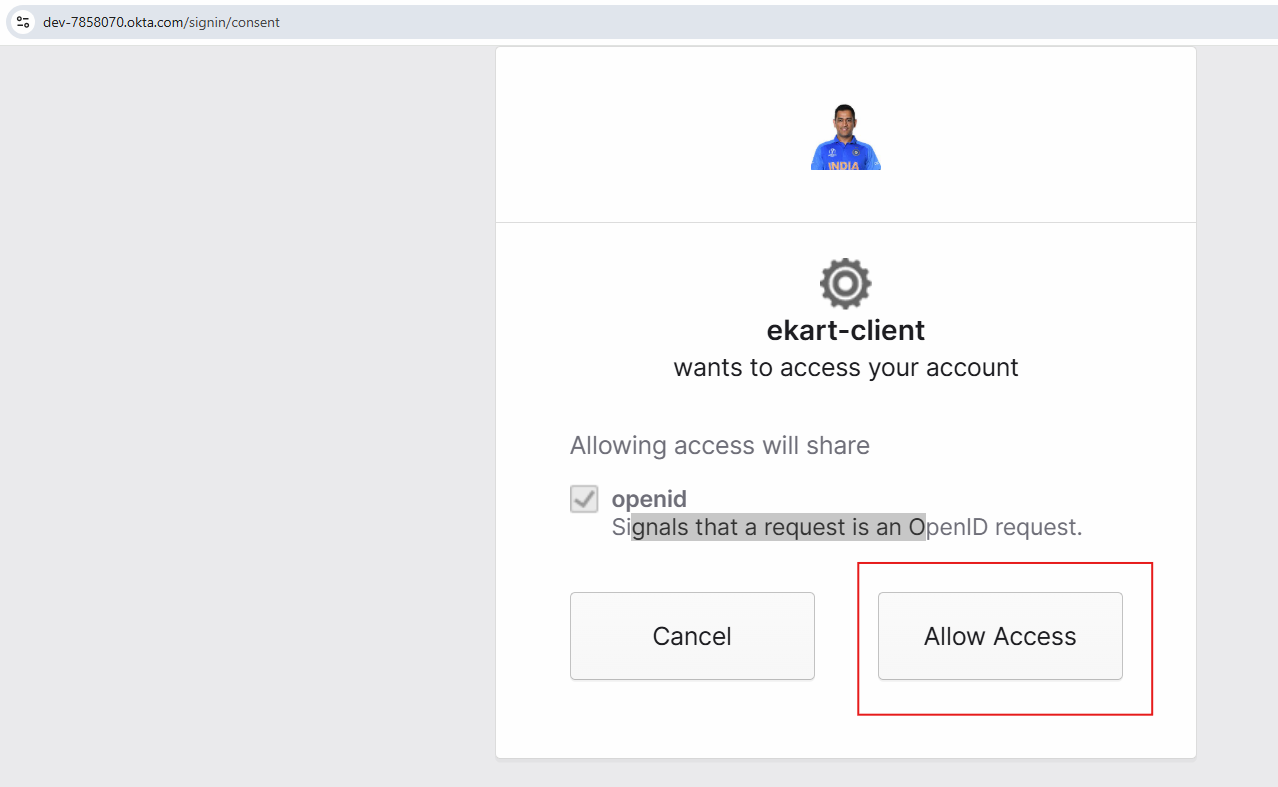
*scope - openid - fixed*

*redirect\_uri http://zoho-app/authorization-code/callback*

*state - 9fe74sdf91-346d-4b9b-8884-c2e59c2fcce2- a random number*

***https://dev-7858070.okta.com/oauth2/default/v1/authorize?client\_id=0oalzyx4mnl73HYGV5d7&response\_type=code&scope=openid&redirect\_uri=http%3A%2F%2Flocalhost%3A8080%2Fauthorization-code%2Fcallback&state=*** ***dc0ca87b-5daa-47ac-ba35-efc72cb88872***

***http://localhost:8080/authorization-code/callback?code=2Wv3\_wYkr0CwL7dSosKK4cKH--vBvZZ-\_xQTgFKu5nc&state=+dc0ca87b-5daa-47ac-ba35-efc72cb88872***

******

***POST endpoint:*** <https://dev-7858070.okta.com/oauth2/default/v1/token>

Body: www-form-url-encoded

grant\_type: authorization\_code

redirect\_url: <http://localhost:8080/authorization-code/callback>

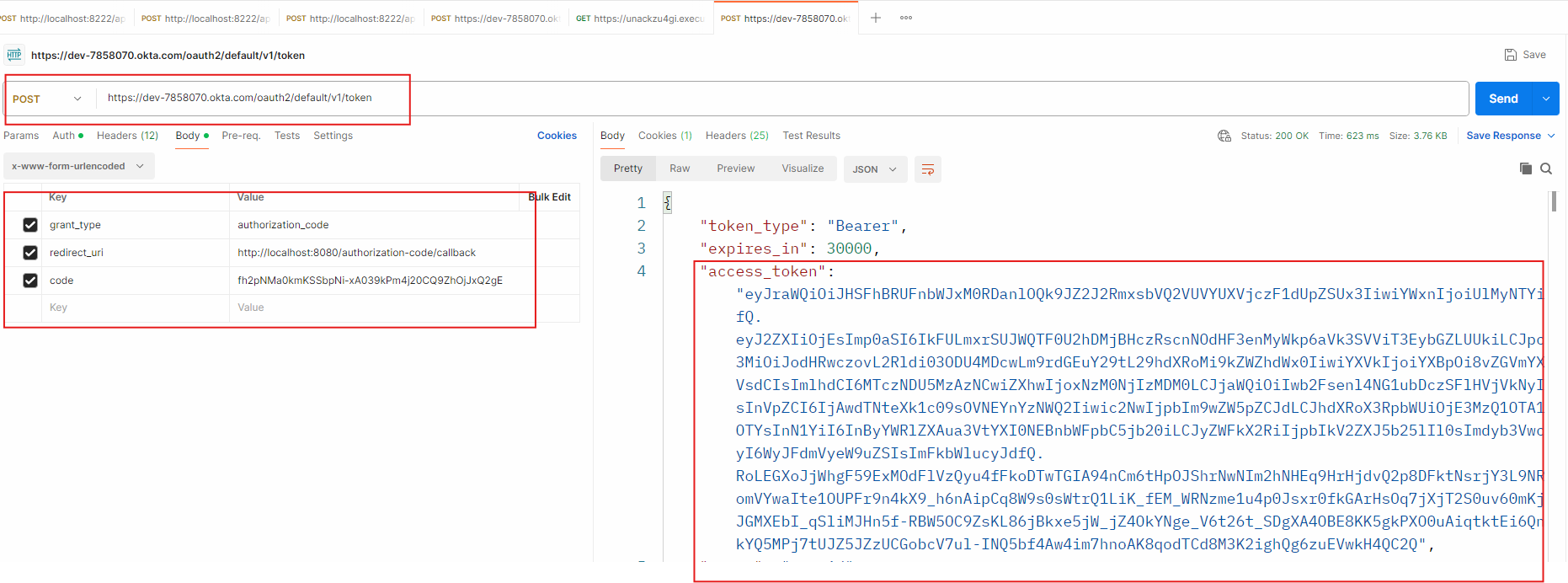
code: <code>

Auth

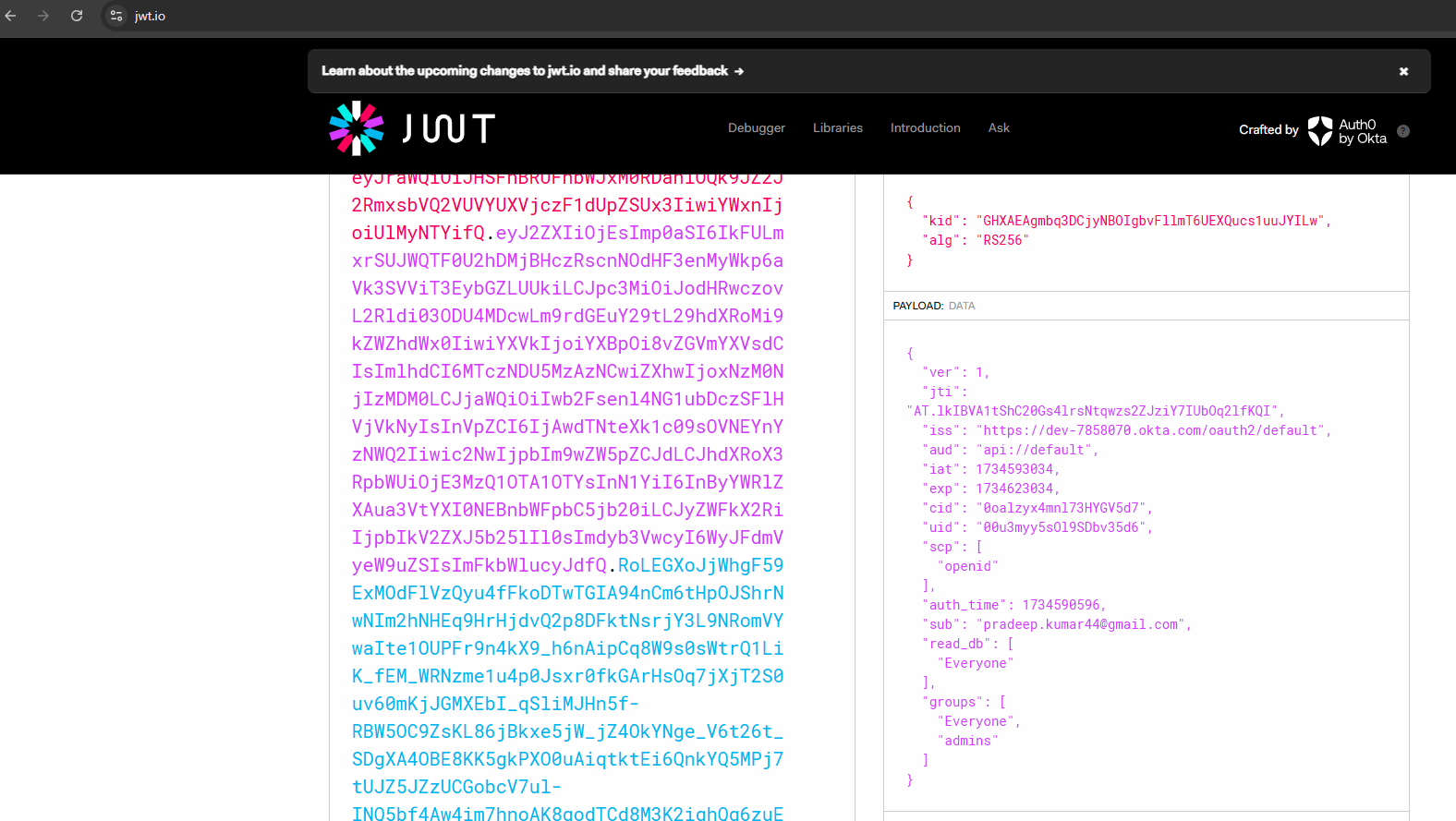
Type: Basic Auth

Username: 0oalzyx4mnl73HYGV5d7

Password: BdEtSWALyLPDjqwnGcl2fv1ylo1vmdLkzWDg08b\_zrJm-gPhb5L3Ypo2AQZS1ifi

******

******

******

JWT Token

1. Json Web Token is made up of three parts
   1. Header
   2. Body
   3. Signature
2. JWT tokens are like digital signatures and cannot be tampered
3. JWT tokens are self-contained information so that the resource-server can validate the token without communicating with the auth server.
4. Should not be used to store sensitive information.
5. To decode the JWT token we need the public keys
6. The auth server exposes the keys to decode the JWT token

Step-3

1. The client application sends the JWT token as bearer token to the resource server
2. The resource server fetches the keys from the issuer url and decodes the JWT token
3. The resources server will check the scopes and based on this will make access control decisions.
4. If the JWT token is invalid/tampered/expired, the resource server will send 401-UnAuthorized
5. If the token is valid but does not contain the valid scopes, then the resource server will send 403-Forbidden
6. Else the Resource server will send the response.

Spring Boot to build the OAuth2.0 Client application

1. Dependencies

<dependencies>  
 **<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-oauth2-client</artifactId>  
 </dependency>** <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.security</groupId>  
 <artifactId>spring-security-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

1. Application Configuration

security:  
 oauth2:  
 client:  
 provider:  
 okta:  
 issuer-uri: https://dev-7858070.okta.com/oauth2/default  
 authorization-uri: https://dev-7858070.okta.com/oauth2/default/v1/authorize  
 token-uri: https://dev-7858070.okta.com/oauth2/default/v1/token  
 user-info-uri: https://dev-7858070.okta.com/oauth2/default/v1/userinfo  
 jwk-set-uri: https://dev-7858070.okta.com/oauth2/default/v1/keys  
 registration:  
 okta:  
 client-id: 0oalzyx4mnl73HYGV5d7  
 client-secret: BdEtSWALyLPDjqwnGcl2fv1ylo1vmdLkzWDg08b\_zrJm-gPhb5L3Ypo2AQZS1ifi  
 scope: openid  
 redirect-uri: http://localhost:8080/authorization-code/callback  
 client-name: Ekart Client App  
 provider: okta  
  
server:  
 port: 8080

OAuth2Resource Server

1. Dependencies

<dependencies>  
 **<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>  
 </dependency>** <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.security</groupId>  
 <artifactId>spring-security-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

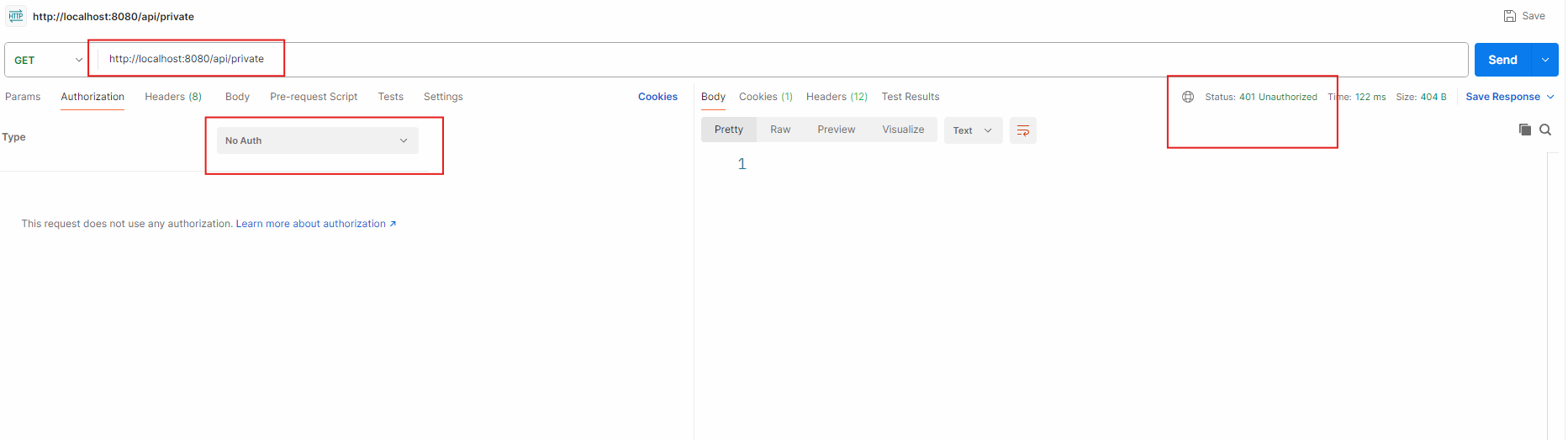
1. Configuration

security:  
 oauth2:  
 resourceserver:  
 jwt:  
 jwk-set-uri: https://dev-7858070.okta.com/oauth2/default/v1/keys  
 issuer-uri: <https://dev-7858070.okta.com/oauth2/default>

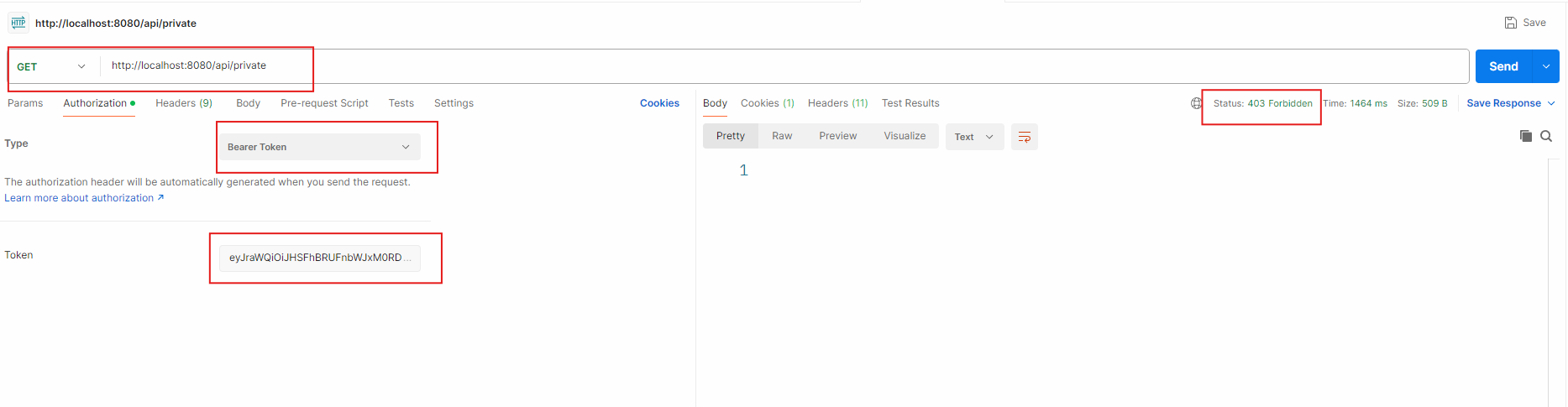
1. SecurityConfiguration

@Configuration  
public class SecurityConfig {  
  
 @Bean  
 public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity, JwtAuthenticationConverter jwtAuthenticationConverter) throws Exception {  
 httpSecurity  
 .authorizeHttpRequests(authorizeRequests ->  
 authorizeRequests  
 .requestMatchers("/api/public").permitAll()  
 .requestMatchers("/api/private").hasAnyRole("Evedsfdsfryone", "sdfsdfsdf")  
 .anyRequest().authenticated()  
 )  
 .oauth2ResourceServer(oauth2ResourceServer ->  
 oauth2ResourceServer.jwt( jwt ->  
 jwt.jwtAuthenticationConverter(jwtAuthenticationConverter())  
 )  
 );  
 return httpSecurity.build();  
 }  
  
 @Bean  
 public JwtAuthenticationConverter jwtAuthenticationConverter() {  
 JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();  
 JwtGrantedAuthoritiesConverter jwtGrantedAuthoritiesConverter = new JwtGrantedAuthoritiesConverter();  
 jwtGrantedAuthoritiesConverter.setAuthoritiesClaimName("groups");  
 jwtGrantedAuthoritiesConverter.setAuthorityPrefix("ROLE\_");  
 jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(jwtGrantedAuthoritiesConverter);  
 return jwtAuthenticationConverter;  
  
 }  
}

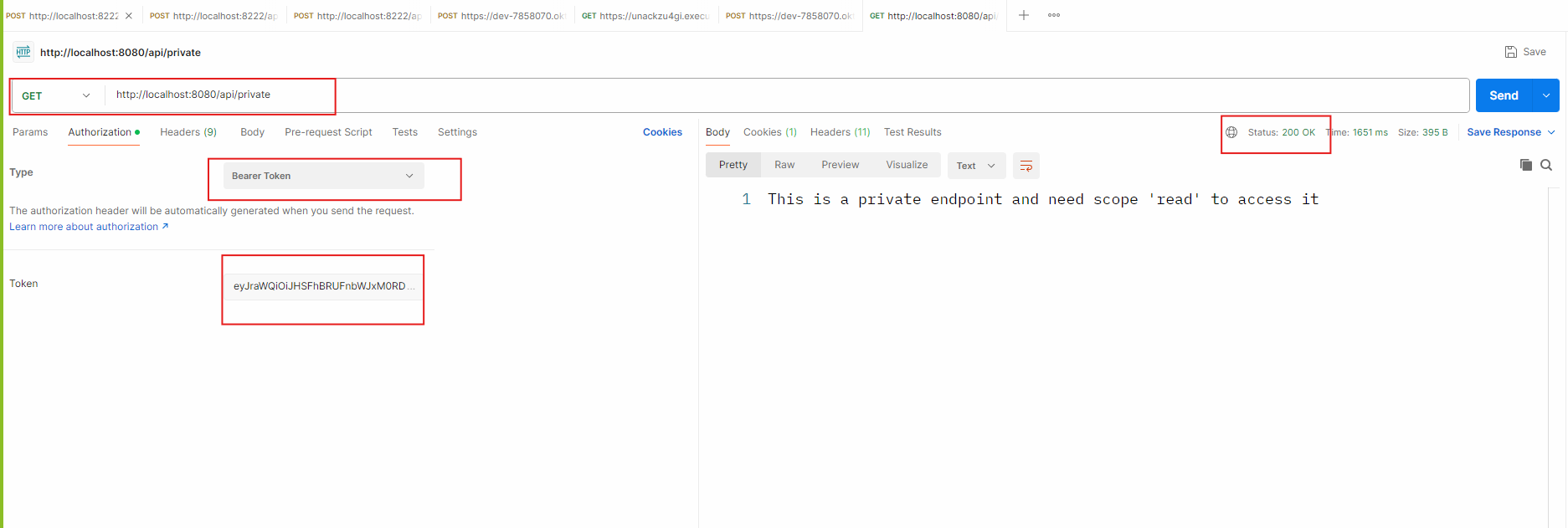
1. Lab:
   1. Without credentials/Invalid credentials



* 1. Valid credentials but not having permissions

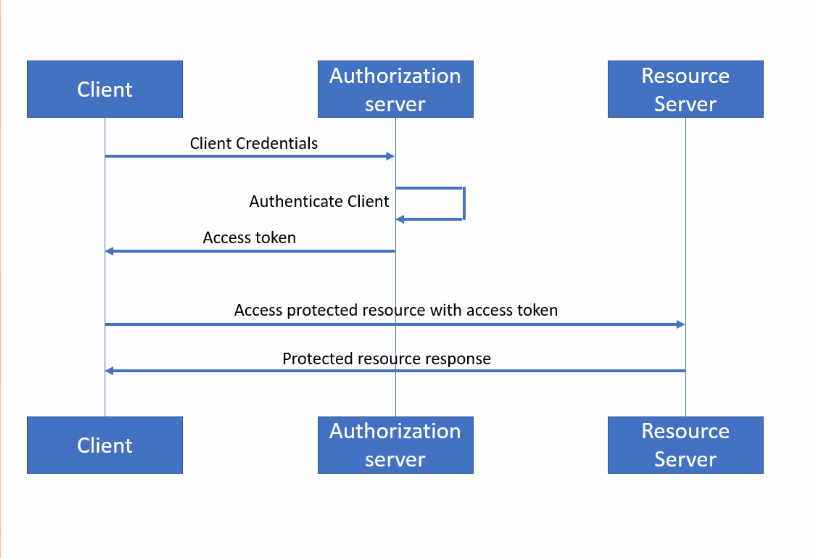


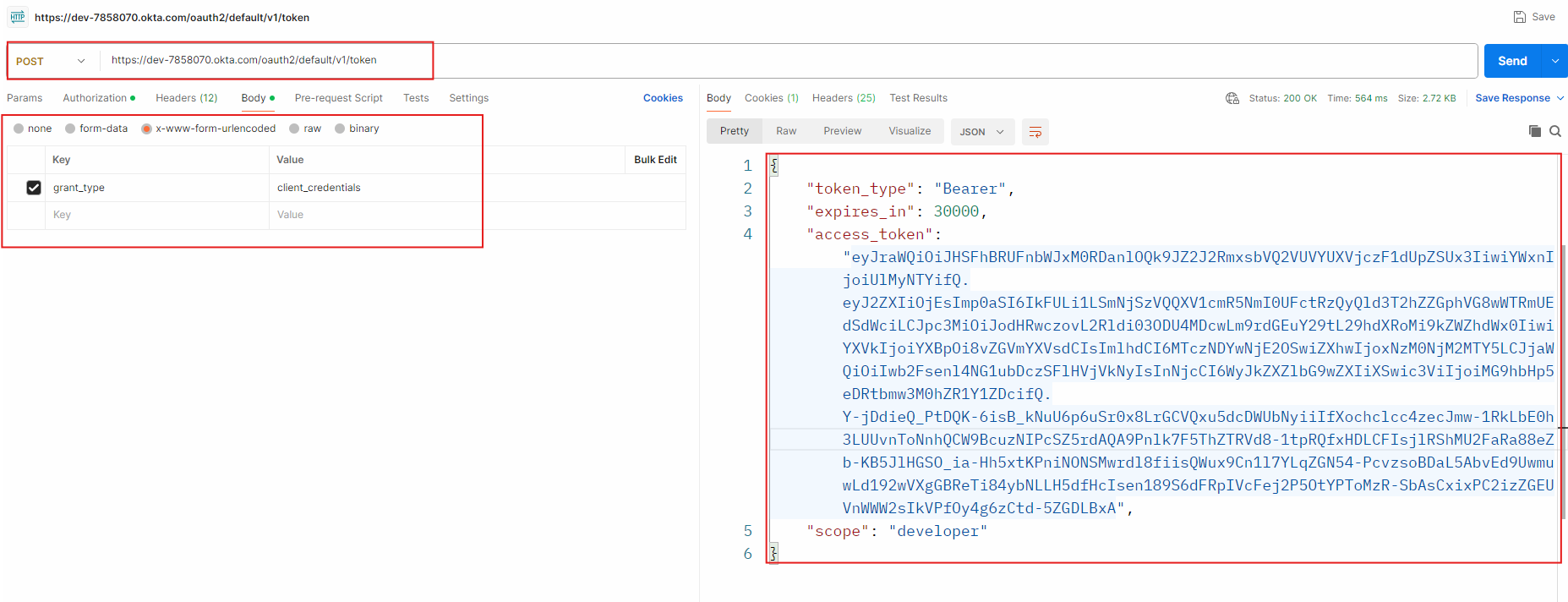
* 1. Valid Credentials and correct access rights



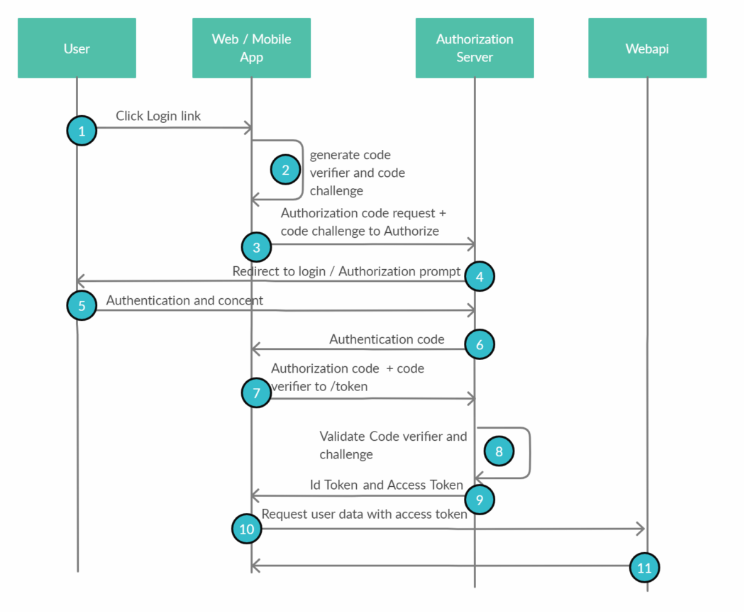
Service to Service Communication

1. There is no user involved when communicating between services.
2. We choose the client-credentials grant flow





PKCE workflow for SPA/Mobile app

****