OAuth 2.0 Implementation with Spring Security and Spring Boot

We will make authorisation server which will hold the authorising part and a resource server which will serve the endpoints.

OAuth 2.0 stands for “Open Authorisation” is a standard designed to allow a website or application to access resources hosted by other web apps on behalf of a user.

OAuth 2.0 is an open standard protocol for authorisation, often used in conjunction with modern web and mobile applications to allow secure access to resources without sharing user credentials. It provides a framework for users to grant access to their resources on one site to another site without sharing their credentials, typically using OAuth tokens.

It is an authorisation protocol and not authentication protocol (we connect openId with it), this means we are giving it permission to use resources.
It is deigned primarily as a means of granting access to a set of resources for example remote apis or user’s data
Oauth 2.0 uses Access tokens

/**

OAuth 2.0 is an open standard protocol for authorization, often used in conjunction with modern web and mobile applications to allow secure access to resources without sharing user credentials. It provides a framework for users to grant access to their resources on one site to another site without sharing their credentials, typically using OAuth tokens.

Key concepts of OAuth 2.0 include:

1. **Authorization Server**: This is responsible for authenticating the user and issuing access tokens after successful authentication. It verifies the identity of the resource owner and grants access to the client application.

2. **Resource Server**: This hosts the protected resources that the client application wants to access. It accepts and responds to requests using access tokens to grant access to these resources.

3. **Client**: This is the application that wants to access the user's resources. It requests authorization from the resource owner through the authorization server and presents the access token to the resource server to access

protected resources.

4. **Authorization Grant**: This is the mechanism used by the client to obtain an access token from the authorization server. Common grant types include authorization code, implicit, password, and client credentials.

5. **Access Token**: This is a credential that represents the user's authorization to access protected resources. The client includes this token in each request to the resource server to authenticate itself.

OAuth 2.0 is widely used for enabling secure, delegated access to web APIs and services. It is commonly employed by social media platforms, cloud services, and other web applications to enable third-party access to user data without exposing user credentials.

When you use Google Login, OAuth 2.0 is indeed involved in the background to securely authenticate users and grant access to their Google account data without sharing their credentials. Here's how it typically works:

1. **User Initiates Login**: When a user wants to log in to a website or application using their Google account, they are redirected to the Google login page.

2. **Request Authorization**: The website or application sends a request to Google's OAuth 2.0 authorization server, asking for permission to access the user's Google account data.

3. **User Grants Permission**: The user is presented with a consent screen, where they can review the requested permissions (such as access to profile information, email, contacts, etc.). If they approve, they authenticate themselves with Google by providing their credentials.

4. **Authorization Code**: After successful authentication, Google sends an authorization code back to the website or application's redirect URI.

5. **Token Exchange**: The website or application exchanges this authorization code for an access token and, optionally, a refresh token by sending a request to Google's token endpoint. This access token represents the user's authorization to access their Google account data.

6. **Accessing Resources**: The website or application can now use the access token to make requests to Google's API on behalf of the user. For example, it can retrieve the user's profile information, email, calendar events, etc.

7. **Token Expiry and Refresh**: Access tokens have a limited validity period. If the access token expires, the website or application can use the refresh token

(if provided) to obtain a new access token without requiring the user to log in again.

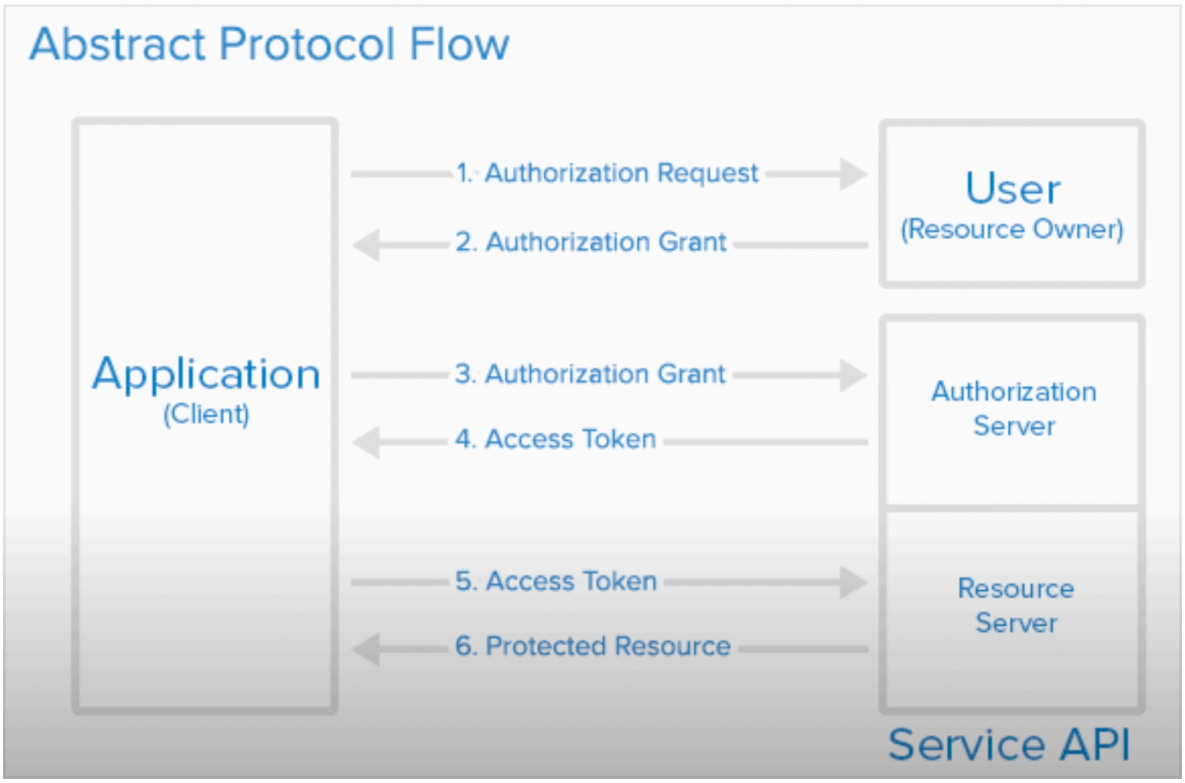
In summary, Google Login uses OAuth 2.0 to securely authenticate users and authorize access to their Google account data, enabling seamless integration with third-party websites and applications. This allows users to sign in using their existing Google credentials without sharing their username and password directly.

*/

OAuth 2.0 Roles
Resource Owner (the user of system that owns the protected resources and can grant access to them)
Client (The system that requires access to the protected resources. To access resources, the Client must hold the appropriate Access Token)
Authorisation Server (This server receives requests from the Client for Access Tokens and issues them upon successful authentication and consent by the Resource Owner)
Resource Server (A server that protects the user's resources and receives access requests from the client. It accepts and validates an access token from the client and returns the appropriate resources to it.)

OAuth 2.0 Scopes

Scopes are an important concept in OAuth 2.0. They are used to specify the reason for which access to resources may be granted.
Define what we are allowed to use (like role based access).
e.g. when we click on login by google we are redirected to google page where we feed the username and password and they ask what want google account data to share with the app.



First the app requests the user for authorisation request and then user gives the authorisation grant. Then app makes authorisation grant with authorisation server which sends back the token. Then with that token we can access protected resources from Resource Server.

Go to OAuth playground to understand better:
https://www.oauth.com/playground/

Whenever we register our app for google auth, we get client secret and id.

There are different OAuth 2.0 flow like connect OpenId, authorisation code, etc.

/**

OAuth 2.0 defines several authorization flows, each designed for different use cases and security requirements. These flows specify how access tokens are obtained and used to access protected resources. The main OAuth 2.0 flows are:

1. **Authorization Code Flow (or Web Server Flow)**:
   - This is the most commonly used flow for server-side applications.
   - It involves multiple steps:
     1. The client (application) redirects the user to the authorization server (e.g., Google) to authenticate.
     2. After authentication, the authorization server redirects the user back to

the client with an authorization code.
    3. The client exchanges this authorization code for an access token by sending a request to the authorization server's token endpoint.
    4. The authorization server validates the code and issues an access token to the client.

2. **Implicit Flow (or Client-Side Flow)**:
   - This flow is suitable for browser-based applications (JavaScript single-page apps) where the client is unable to securely store client secrets.
   - It involves a simplified authorization process without exchanging an authorization code for a token.
   - The access token is returned directly to the client after the user grants permission, via the browser's redirect URI.

3. **Resource Owner Password Credentials Flow**:
   - In this flow, the user's username and password are exchanged directly for an access token.
   - It is primarily used when the client application trusts the user and can securely collect their credentials.
   - However, this flow is less secure because the client has direct access to the user's credentials.

4. **Client Credentials Flow**:
   - This flow is suitable for machine-to-machine communication where the client is acting on its own behalf (not on behalf of a user).
   - The client authenticates itself directly to the authorization server using its own credentials (client ID and client secret).
   - The authorization server issues an access token directly to the client, without involving a user.

5. **Device Authorization Flow** (also known as Device Flow or OAuth for Browserless and Input Constrained Devices):
   - This flow is designed for devices with limited input capabilities (such as smart TVs, media consoles, etc.) that cannot easily enter credentials.
   - It involves a two-step process where the device initially obtains a verification code from the authorization server and then exchanges it for an access token.

Each OAuth 2.0 flow has its own set of security considerations and is suited for different types of applications and devices. It's important to choose the appropriate flow based on the requirements and constraints of your application.

*/

https://authorization-server.com/authorize?
  response_type=code
  &client_id=bfB-zN2tQj2ISBRopbq8qdVf

```
&redirect_uri=https://www.oauth.com/playground/authorization-code.html
&scope=photo+offline_access
&state=AmIBPr7HP5ODGsUL
```

response_type depending on the type of flow.
redirect_uri is the uri to redirect the client after the authorisation

Before authorization begins, it first generates a random string to use for the state parameter. The client will need to store this to be used in the next step.

Make an authorisation server with dependencies jpa, driver, web, security, Lombok

Now add the dependency
Spring-security-oauth2-authorisation-server

We have to have same user entity, services and repositories as in client app.

To let spring security track the users, we need custom UserDetailsService, we make inside the service package

```
@Service
@Transactional
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(11);
    }

    @Override
    public UserDetails loadUserByUsername(String email) throws
UsernameNotFoundException {
        User user = userRepository.findByEmail(email);
        if(user == null) {
            throw  new UsernameNotFoundException("No User Found");
        }
        return new org.springframework.security.core.userdetails.User(
                user.getEmail(),
                user.getPassword(),
                user.isEnabled(),
                true,
                true,
```

```
            true,
            getAuthorities(List.of(user.getRole()))
        );
    }

    private Collection<? extends GrantedAuthority> getAuthorities(List<String>
roles) {
        List<GrantedAuthority>  authorities = new ArrayList<>();
        for(String role: roles) {
            authorities.add(new SimpleGrantedAuthority(role));
        }
        return authorities;
    }
}
```

User looks like

```
package com.dailycodebuffer.oauthserver.entity;

import lombok.Data;

import javax.persistence.*;

@Entity
@Data
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String email;

    @Column(length = 60)
    private String password;

    private String role;
    private boolean enabled = false;
}
```

Now we need to create AuthorisationServerConfig class under config package

This helps in exchange of authorisation code and access token

```
@Configuration(proxyBeanMethods = false)
public class AuthorizationServerConfig {
```

```
@Autowired
    private PasswordEncoder passwordEncoder;

/**
```

1. SecurityFilterChain **Configuration**:
    - Inside the method, the OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http) method is invoked to apply default security configuration for an OAuth 2.0 authorization server.
    - This includes setting up authorization endpoints, token endpoints, and other security measures required for OAuth 2.0 functionality.
2. http.formLogin(Customizer.withDefaults()).build():
    - This configures the HTTP security for the authorization server using a form-based login mechanism.
    - The Customizer.withDefaults() method applies default customizations to the form login configuration.
    - Finally, the build() method builds the HttpSecurity instance with the configured settings.

```
*/
@Bean
    @Order(Ordered.HIGHEST_PRECEDENCE)
    public SecurityFilterChain authServerSecurityFilterChain(HttpSecurity http)
throws Exception {
        // we are having default auth to authorisation functionality
        // default slash authorised api, slash token api, all the default configuration
will be added
        // like jwt decoder, rsa keys
        OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);

        return http.formLogin(Customizer.withDefaults()).build();
        // we will be setup with default configuration with OAuth 2.0
    }

// now code to register the client
// this is code for just one client
// to manage multiple client we can make our own implementation of
RegisteredClientRepository

@Bean
    public RegisteredClientRepository registeredClientRepository() {
        RegisteredClient registeredClient =
RegisteredClient.withId(UUID.randomUUID().toString())
                .clientId("api-client") // giving the client id and secret id
                .clientSecret(passwordEncoder.encode("secret"))
                .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SE
```

```
CRET_BASIC)
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_C
ODE)
            .authorizationGrantType(AuthorizationGrantType.PASSWORD)
            .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)
            .redirectUri("http://127.0.0.1:8080/login/oauth2/code/api-client-oidc")
            .redirectUri("http://127.0.0.1:8080/authorized")
            .scope(OidcScopes.OPENID)
            .scope("api.read")
            .clientSettings(ClientSettings.builder().requireAuthorizationConsent(tr
ue).build()) // we are allowing to give consent back to client
            .build();
            // there are 2 scopes
            // one scope is for the open id connect because we are authenticating
as well
            // grant type allows the client to access through
        return new InMemoryRegisteredClientRepository(registeredClient);// this
registeredClient will be in memory of this server
    }
```

The code snippet you provided configures an OpenID Connect (OIDC) client, specifying various settings such as scopes and client settings. Let's break down what each part does:

1. **`.scope(OidcScopes.OPENID)`**:
   - This line sets the scope of the OIDC client to include the `openid` scope. The `openid` scope indicates that the client is requesting access to the user's OpenID, which includes information such as the user's unique identifier and authentication details.

2. **`.scope("api.read")`**:
   - Additionally, this line adds a custom scope to the OIDC client, namely `api.read`. Custom scopes are used to define specific permissions or access levels that the client requires. In this case, `api.read` suggests that the client needs read access to an API.

3. **`.clientSettings(ClientSettings.builder().requireAuthorizationConsent(true).build())`**:
   - Here, the `clientSettings` method is used to specify settings for the client, such as whether authorization consent is required. In this case, `requireAuthorizationConsent(true)` indicates that the client requires explicit consent from the user before accessing their information or resources.
   - By setting `requireAuthorizationConsent` to `true`, the client is configured to prompt the user to consent to the requested scopes and permissions during the authentication process. This ensures that users have control over what

information they share with the client.

In summary, the provided code configures an OIDC client with scopes for OpenID and custom access to an API, while also requiring explicit user consent for authorization. This setup is common in scenarios where clients need to access user information or resources securely and transparently.

The `clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)` configuration specifies the method that the client uses to authenticate itself to the authorization server during the OAuth 2.0 authentication flow. Let's break down its significance:

1. **Client Authentication Method**:
   - In OAuth 2.0, clients need to authenticate themselves to the authorization server to obtain an access token. This authentication is crucial for the authorization server to verify the identity of the client and ensure that only authorized clients can access protected resources.

2. **CLIENT_SECRET_BASIC**:
   - `CLIENT_SECRET_BASIC` is one of the client authentication methods defined by the OAuth 2.0 specification. It involves sending the client credentials (client ID and client secret) in the HTTP request headers using the Basic authentication scheme.
   - When the `CLIENT_SECRET_BASIC` method is used, the client ID and client secret are Base64-encoded and sent as part of the Authorization header in the HTTP request.

3. **Purpose**:
   - By specifying `clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)`, the OAuth 2.0 client is configured to authenticate itself using the client ID and client secret provided by the authorization server.
   - This ensures that the client credentials are securely transmitted to the authorization server, allowing it to verify the client's identity and grant access tokens accordingly.

In summary, `clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)` is used to configure the OAuth 2.0 client to authenticate itself using the `CLIENT_SECRET_BASIC` method, where the client ID and client secret are sent securely in the HTTP request headers during the authentication process.

Now we require to to the private and public key configuration

```
@Bean
   public JWKSource<SecurityContext> jwkSource() {
      RSAKey rsaKey = generateRsa();
      JWKSet jwkSet = new JWKSet(rsaKey);
      return (jwkSelector, securityContext) -> jwkSelector.select(jwkSet);
   }

   private static RSAKey generateRsa() {
      KeyPair keyPair = generateRsaKey();
      RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
      RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
      return new RSAKey.Builder(publicKey)
            .privateKey(privateKey)
            .keyID(UUID.randomUUID().toString())
            .build();
   }

   private static KeyPair generateRsaKey() {
      KeyPair keyPair;
      try {
         KeyPairGenerator keyPairGenerator =
KeyPairGenerator.getInstance("RSA");
         keyPairGenerator.initialize(2048);
         keyPair = keyPairGenerator.generateKeyPair();
      } catch (Exception ex) {
         throw new IllegalStateException(ex);
      }
      return keyPair;
   }


// authorisation server provider configuration

 @Bean
   public ProviderSettings providerSettings() {
      return ProviderSettings.builder()
            .issuer("http://auth-server:9000")
            .build();
   }

// issuer contains the url to the auth server only
// actually we can configure to create this in  hosts  file in our machine (that is
create domain name for that server so that auth-server refresh to ip address
```

localhost)
// to see the hosts available type command cat /private/etc/hosts
// we can make changes in this file

In this code snippet, a bean named `providerSettings` is declared using the `@Bean` annotation. This method creates and configures a `ProviderSettings` object, which likely represents settings related to the identity provider (IDP) or authorization server used in the application. Let's break down what each part of the method does:
ProviderSettings Configuration:
- The `ProviderSettings` class (presumably from a library or framework) is used to configure settings related to the identity provider or authorization server.
- The `issuer("http://auth-server:9000")` method configures the issuer URL of the provider. The issuer URL is the URL of the IDP or authorization server that issues authentication tokens.
- Other settings related to the provider can be configured similarly, although they are not shown in this snippet.

`build()` Method:
- The `build()` method is called to build the `ProviderSettings` object with the configured settings. This method returns an instance of `ProviderSettings` with the specified configurations.

Overall, this `providerSettings()` method creates a bean that provides settings for the identity provider or authorization server used in the application, specifically setting the issuer URL. This bean can then be injected into other components that require access to these provider settings.

- In some cases, hitting the URL may trigger a redirect to another URL or prompt you to authenticate or authorize yourself.
- For example, if the URL is the authorization endpoint of an OAuth 2.0 server, hitting it may redirect you to a login page where you can authenticate yourself and authorize access to your resources.

Now add the default spring security configuration

```
@EnableWebSecurity
public class DefaultSecurityConfig {

    @Autowired
    private CustomAuthenticationProvider customAuthenticationProvider;

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws
```

```
Exception {
        http.authorizeRequests(authorizeRequests ->
                        authorizeRequests.anyRequest().authenticated()
                )
                .formLogin(Customizer.withDefaults());
        return http.build();
    }

    @Autowired
    public void bindAuthenticationProvider(AuthenticationManagerBuilder
authenticationManagerBuilder) {
        authenticationManagerBuilder
                .authenticationProvider(customAuthenticationProvider);
    }
}
```

Now to manage our authentication we can declare our own (custom) authentication provider
Validating the input credentials with that of user.

```
@Service
public class CustomAuthenticationProvider implements AuthenticationProvider
{

    @Autowired
    private CustomUserDetailsService customUserDetailsService;

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Override
    public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
        String username = authentication.getName();
        String password = authentication.getCredentials().toString();
        UserDetails user=
customUserDetailsService.loadUserByUsername(username);
        return checkPassword(user,password);
    }

    private Authentication checkPassword(UserDetails user, String rawPassword)
{
        if(passwordEncoder.matches(rawPassword, user.getPassword())) {
            return new
UsernamePasswordAuthenticationToken(user.getUsername(),
```

```
                user.getPassword(),
                user.getAuthorities());
        }
        else {
            throw new BadCredentialsException("Bad Credentials");
        }
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return
UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication
);
    }
}

// now we need to configure the client

In application.yml

spring:
  security:
    oauth2:
      client:
        registration:
          api-client-oidc: (name of the client)
            provider: spring
            client-id: api-client
            client-secret: secret
            authorization-grant-type: authorization_code
            redirect-uri: "http://127.0.0.1:8080/login/oauth2/code/{registrationId}"
(registrationId is same as name of the client)
            scope: openid (for authentication )
            client-name: api-client-oidc
          api-client-authorization-code:
            provider: spring
            client-id: api-client
            client-secret: secret
            authorization-grant-type: authorization_code
            redirect-uri: "http://127.0.0.1:8080/authorized" (once authorised we will
be redirected to)
            scope: api.read
            client-name: api-client-authorization-code
        provider:
          spring:
            issuer-uri: http://auth-server:9000
```

These urls are standards to be used.

Now we have to configure the web security config

```
package com.dailycodebuffer.client.config;

import org.springframework.context.annotation.Bean;
import org.springframework.security.config.Customizer;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebS
ecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@EnableWebSecurity
public class WebSecurityConfig {

    private static final String[] WHITE_LIST_URLS = {
            "/hello",
            "/register",
            "/verifyRegistration*",
            "/resendVerifyToken*"
    };

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(11);
    }

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .cors()
                .and()
                .csrf()
                .disable()
                .authorizeHttpRequests()
                .antMatchers(WHITE_LIST_URLS).permitAll()
                .antMatchers("/api/**").authenticated()
                .and()
```

```
            // add the configuration for auth server
            // for that we have to add dependency of spring-boot-starter-oauth2-
client
            .oauth2Login(oauth2login ->
                  oauth2login.loginPage("/oauth2/authorization/api-client-oidc"))
            .oauth2Client(Customizer.withDefaults());

      return http.build();
   }
}
```

Actually we have certain endpoints in client which needs to be authenticated to access the protected resources

So now if we call localhost:8080/api/hello we will be redirected to http://auth-server:9000/login

To get the user details logged in we can inject Principal principal object in controller

If we write localhost:8080/api/hello then after authentication we will be redirected to home page
But if we hit 127.0.0.1/api/hello we will be redirect to same page after login

Now make a resource server with dependency spring web, security, OAuth2 Resource server

To configure as a resource server:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://auth-server:9000
```

```
@EnableWebSecurity
public class ResourceServerConfig {

   @Bean
   SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
      http
            .authorizeRequests()
            .mvcMatchers("/api/**")
            .access("hasAuthority('SCOPE_api.read')") // only allowing the client
with scope allowed to access this
```

```
            .and()
            .oauth2ResourceServer()
            .jwt();
        return http.build();
    }
}
```

We have to configure the web client in client to access the protected endpoints of resource server

We need to add 2 more dependency in client server in order to use web client. Spring-webflux, reactor-netty

```
@Configuration
public class WebClientConfiguration {

    @Bean
    WebClient webClient(OAuth2AuthorizedClientManager
authorizedClientManager) {
        ServletOAuth2AuthorizedClientExchangeFilterFunction oauth2Client =
                new
ServletOAuth2AuthorizedClientExchangeFilterFunction(authorizedClientManag
er);
        return WebClient.builder()
                .apply(oauth2Client.oauth2Configuration()) // web client is configured
to use oauth2
                .build();
    }

    @Bean
    OAuth2AuthorizedClientManager authorizedClientManager(
            ClientRegistrationRepository clientRegistrationRepository,
            OAuth2AuthorizedClientRepository authorizedClientRepository) {

        OAuth2AuthorizedClientProvider authorizedClientProvider =
                OAuth2AuthorizedClientProviderBuilder.builder()
                        .authorizationCode()
                        .refreshToken()
                        .build();
        DefaultOAuth2AuthorizedClientManager authorizedClientManager = new
DefaultOAuth2AuthorizedClientManager(
                clientRegistrationRepository, authorizedClientRepository);

authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider
);
```

```
        return authorizedClientManager;
    }
}
```

Now in client we can access the resource api

```
@GetMapping("/api/users")
    public String[] users(
            @RegisteredOAuth2AuthorizedClient("api-client-authorization-code")
                    OAuth2AuthorizedClient client){
        return this.webClient
                .get()
                .uri("http://127.0.0.1:8090/api/users")
                .attributes(oauth2AuthorizedClient(client)) //
ServletOAuth2AuthorizedClientExchangeFilterFunction.oauth2AuthorizedClient(
Client)
                .retrieve()
                .bodyToMono(String[].class)
                .block();
    }
```

We are trying to access this endpoint

```
@GetMapping("/api/users")
    public String[] getUser() {
        return new String[]{"Shabbir", "Nikhil","Shivam"};
    }
```