

OAuth 2.0

1. Introduction

In the traditional client-server authentication model, the client requests an access-restricted resource (protected resource) on the server by authenticating with the server using the resource owner's credentials. In order to provide third-party applications access to restricted resources, the resource owner shares its credentials with the third party. This creates several problems and limitations:

- o Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear-text.
- o Servers are required to support password authentication, despite the security weaknesses inherent in passwords.
- o Third-party applications gain overly broad access to the resource owner's protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.
- o Resource owners cannot revoke access to an individual third party without revoking access to all third parties, and must do so by changing the third party's password.
- o Compromise of any third-party application results in compromise of the end-user's password and all of the data protected by that password.

OAuth addresses these issues by introducing an authorization layer and separating the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server, and is issued a different set of credentials than those of the resource owner. Instead of using the resource owner's credentials to access protected resources, the client obtains an access token -- a string denoting a specific scope, lifetime, and other access attributes. Access tokens are issued to third-party clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server.

For example, an end-user (resource owner) can grant a printing service (client) access to her protected photos stored at a photo-sharing service (resource server), without sharing her username and password with the printing service. Instead, she authenticates directly with a server trusted by the photo-sharing service (authorization server), which issues the printing service delegation-specific credentials (access token). This specification is designed for use with HTTP ([\[RFC2616\]](#)). The use of OAuth over any protocol other than HTTP is out of scope.

1.1. Roles

OAuth defines four roles:

Resource owner

An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

Resource server

The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.

Client

An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).

Authorization server

The server issues access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

The interaction between the authorization server and resource server is beyond the scope of this specification. The authorization server may be the same server as the resource server or a separate entity. A single authorization server may issue access tokens accepted by multiple resource servers.

1.2. Protocol Flow

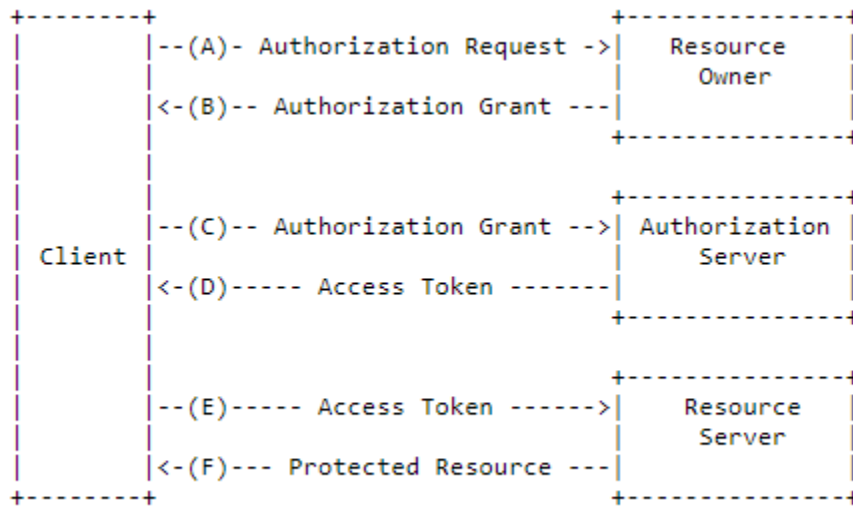


Figure 1: Abstract Protocol Flow

The abstract OAuth 2.0 flow illustrated in Figure 1 describes the interaction between the four roles and includes the following steps:

- (A) The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or preferably indirectly via the authorization server as an intermediary.
- (B) The client receives an authorization grant, which is a credential representing the resource owner's authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.
- (C) The client requests an access token by authenticating with the authorization server and presenting the authorization grant.
- (D) The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token.
- (E) The client requests the protected resource from the resource server and authenticates by presenting the access token.
- (F) The resource server validates the access token, and if valid, serves the request.

The preferred method for the client to obtain an authorization grant from the resource owner (depicted in steps (A) and (B)) is to use the authorization server as an intermediary

[1.3.](#) Authorization Grant

An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token. This specification defines four grant types -- authorization code, implicit, resource owner password credentials, and client credentials -- as well as an extensibility mechanism for defining additional types.

[1.3.1.](#) Authorization Code

The authorization code is obtained by using an authorization server as an intermediary between the client and resource owner. Instead of requesting authorization directly from the resource owner, the client directs the resource owner to an authorization server (via its user-agent as defined in [\[RFC2616\]](#)), which in turn directs the resource owner back to the client with the authorization code.

Before directing the resource owner back to the client with the authorization code, the authorization server authenticates the resource owner and obtains authorization. Because the resource owner only authenticates with the authorization server, the resource owner's credentials are never shared with the client.

The authorization code provides a few important security benefits, such as the ability to authenticate the client, as well as the transmission of the access token directly to the client without passing it through the resource owner's user-agent and potentially exposing it to others, including the resource owner.

[1.3.2.](#) Implicit

The implicit grant is a simplified authorization code flow optimized for clients implemented in a browser using a scripting language such as JavaScript. In the implicit flow, instead of issuing the client an authorization code, the client is issued an access token directly (as the result of the resource owner authorization). The grant type is implicit, as no intermediate credentials (such as an authorization code) are issued (and later used to obtain an access token).

When issuing an access token during the implicit grant flow, the authorization server does not authenticate the client. In some cases, the client identity can be verified via the redirection URI used to deliver the access token to the client. The access token may be exposed to the resource owner or other applications with access to the resource owner's user-agent.

Implicit grants improve the responsiveness and efficiency of some clients (such as a client implemented as an in-browser application), since it reduces the number of round trips required to

obtain an access token. However, this convenience should be weighed against the security implications of using implicit grants.

[1.3.3. Resource Owner Password Credentials](#)

The resource owner password credentials (i.e., username and password) can be used directly as an authorization grant to obtain an access token. The credentials should only be used when there is a high degree of trust between the resource owner and the client (e.g. The client is part of the device operating system or a highly privileged application), and when other authorization grant types are not available (such as an authorization code).

Even though this grant type requires direct client access to the resource owner credentials, the resource owner credentials are used for a single request and are exchanged for an access token. This grant type can eliminate the need for the client to store the resource owner credentials for future use, by exchanging the credentials with a long-lived access token or refresh token.

[1.3.4. Client Credentials](#)

The client credentials (or other forms of client authentication) can be used as an authorization grant when the authorization scope is limited to the protected resources under the control of the client, or to protected resources previously arranged with the authorization server. Client credentials are used as an authorization grant typically when the client is acting on its own behalf (the client is also the resource owner) or is requesting access to protected resources based on an authorization previously arranged with the authorization server.

[1.4. Access Token](#)

Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.

The token may denote an identifier used to retrieve the authorization information or may self-contain the authorization information in a verifiable manner (i.e., a token string consisting of some data and a signature). Additional authentication credentials, which are beyond the scope of this specification, may be required in order for the client to use a token.

The access token provides an abstraction layer, replacing different authorization constructs (e.g., username and password) with a single token understood by the resource server. This abstraction

enables issuing access tokens more restrictive than the authorization grant used to obtain them, as well as removing the resource server's need to understand a wide range of authentication methods.

Access tokens can have different formats, structures, and methods of utilization (e.g., cryptographic properties) based on the resource server security requirements. Access token attributes and the methods used to access protected resources are beyond the scope of this specification and are defined by companion specifications such as [[RFC6750](#)].

1.5. Refresh Token

Refresh tokens are credentials used to obtain access tokens. Refresh tokens are issued to the client by the authorization server and are used to obtain a new access token when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope (access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner). Issuing a refresh token is optional at the discretion of the authorization server. If the authorization server issues a refresh token, it is included when issuing an access token (i.e., step (D) in Figure 1).

A refresh token is a string representing the authorization granted to the client by the resource owner. The string is usually opaque to the client. The token denotes an identifier used to retrieve the authorization information. Unlike access tokens, refresh tokens are intended for use only with authorization servers and are never sent to resource servers.

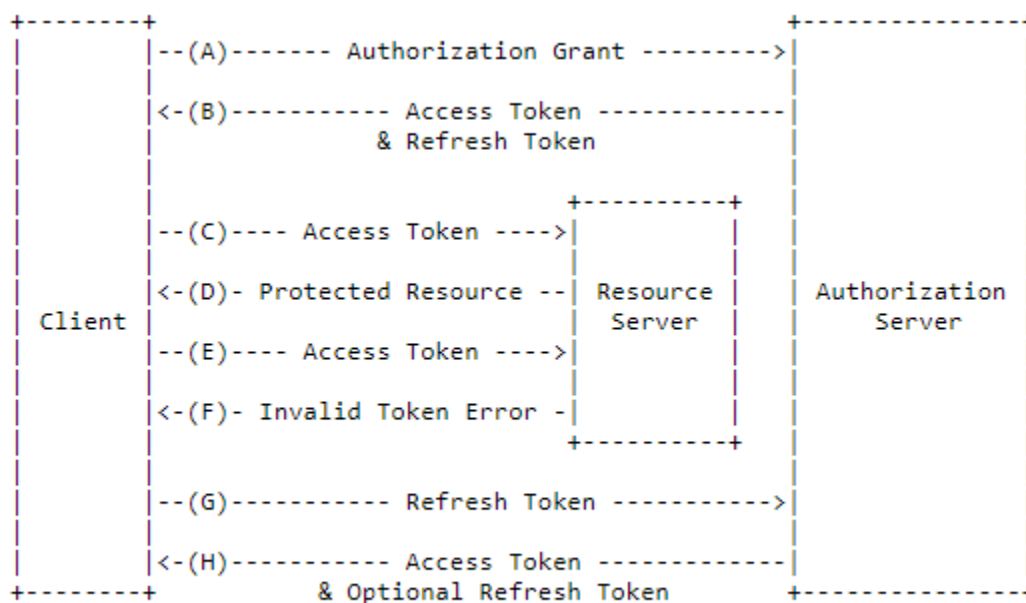


Figure 2: Refreshing an Expired Access Token

The flow illustrated in Figure 2 includes the following steps:

- (A) The client requests an access token by authenticating with the authorization server and presenting an authorization grant.
- (B) The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token and a refresh token.
- (C) The client makes a protected resource request to the resource server by presenting the access token.
- (D) The resource server validates the access token, and if valid, serves the request.
- (E) Steps (C) and (D) repeat until the access token expires. If the client knows the access token expired, it skips to step (G); otherwise, it makes another protected resource request.
- (F) Since the access token is invalid, the resource server returns an invalid token error.
- (G) The client requests a new access token by authenticating with the authorization server and presenting the refresh token. The client authentication requirements are based on the client type and on the authorization server policies.
- (H) The authorization server authenticates the client and validates the refresh token, and if valid, issues a new access token (and, optionally, a new refresh token).

2. Implementations

2.1 Using Google Sign-in (Google OAuth Server)

Through this example, we will implement single sign on functionality with Google accounts for an existing Spring Boot Application, using Spring OAuth2 Client library (Allowing the end users to login using their own Google Accounts instead of application-managed credentials)

2.1.1 Creating client id and client secret

1. Open the [Google API Console Credentials page](#).
2. Click **Select a project**, then **NEW PROJECT**, and enter a name for the project, and optionally, edit the provided project ID. Click **Create**.
3. On the Credentials page, select **Create credentials**, then **OAuth client ID**.

4. You may be prompted to set a product name on the Consent screen; if so, click **Configure consent screen**, supply the requested information, and click **Save** to return to the Credentials screen.
5. Select **Web Application** for the **Application Type**. Follow the instructions to enter JavaScript origins, redirect URIs, or both.
6. Click **Create**.
7. On the page that appears, copy the **client ID** and **client secret** to your clipboard, as you will need them when you configure your client library.



The screenshot shows a dialog box titled "OAuth client". It contains two text input fields. The first field is labeled "Here is your client ID" and has a copy icon to its right. The second field is labeled "Here is your client secret" and also has a copy icon to its right. At the bottom of the dialog is a blue button labeled "OK".

Ref <https://developers.google.com/adwords/api/docs/guides/authentication>
<https://www.youtube.com/watch?v=xH6hAW3EqLk>

Note that you need to add an authorized redirect URI like this:

<http://localhost:8080/login/oauth2/code/google>

In case your application is hosted with its own context path, e.g. */Shopme* - then specify the redirect URI like this:

<http://localhost:8080/Shopme/login/oauth2/code/google>

2.1.2 Declare Dependency for Spring Boot OAuth2 Client

Besides Spring Security Dependency, add OAuth2 Client dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

2.1.3 Configure Spring OAuth2 Properties for Google

Update application.yml file with following properties

```
spring:
  security:
    oauth2:
```



```

client:
  registration:
    google:
      clientId: YOUR_GOOGLE_CLIENT_ID
      clientSecret: YOUR_GOOGLE_CLIENT_SECRET
      scope:
        - email
        - profile

```

Update CLIENT_ID and CLIENT_SECRET

```

spring:
  security:
    oauth2:
      client:
        registration:
          github:
            clientId: |
            clientSecret:
          google:
            clientId:
            clientSecret:

```

(We can add other OAuth client id's)

2.1.4 Updating Security Configuration

Update security configuration as follows

```

@Configuration
@EnableWebSecurity
public class SecurityConfig {

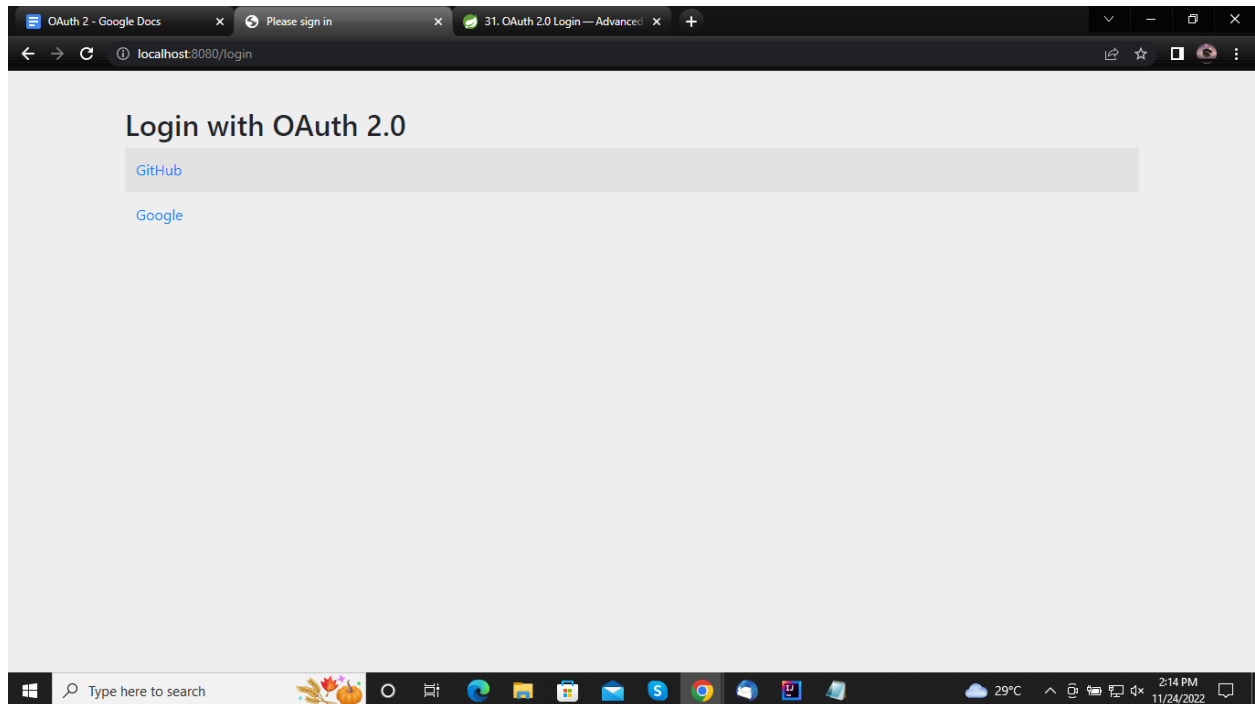
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .anyRequest() ExpressionUrlAuthorizationConfigurer<...>.AuthorizedUrl
            .authenticated() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .and() HttpSecurity
            .logout() LogoutConfigurer<HttpSecurity>
            .and() HttpSecurity
            .oauth2Login();
        return http.build();
    }
}

```

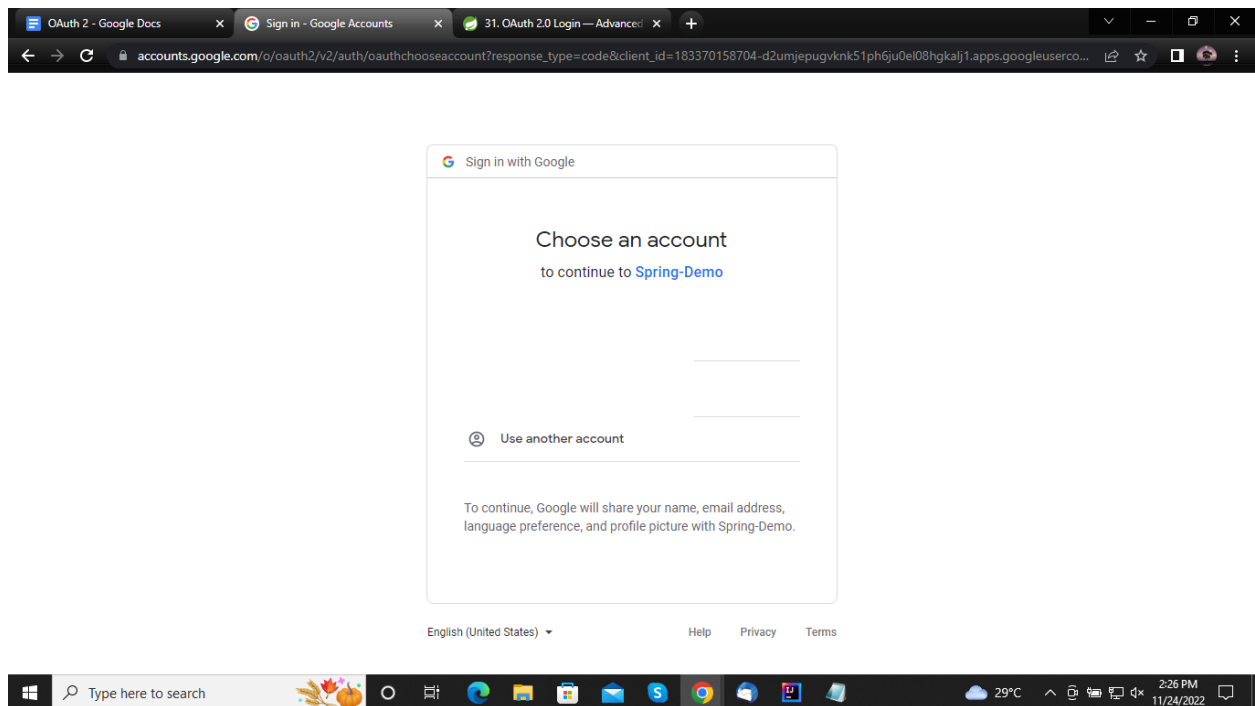
2.1.5 Testing our project

Accessing Spring Boot Application from browser

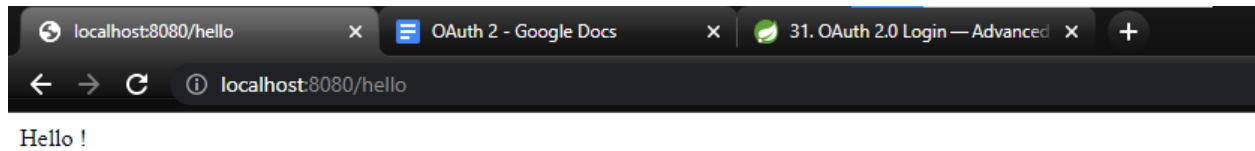
<http://localhost:8080>



After Clicking on Google we will be redirected to Google Sign in as follows:



After Signing in with Google we will be redirected back to our endpoint:



2.2 Simple OAuth application using the Spring Security Authorization Server (client-server application)

2.2.1 Authorization Server Implementation

Dependencies

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>2.5.4</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
  <version>2.5.4</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-oauth2-authorization-server</artifactId>
  <version>0.2.0</version>
</dependency>
```

Configuration

application.yml

```
server:
  port: 9000
```

Let's create @Configuration class

Defining RegisteredClientRepository bean

```

@Bean
public RegisteredClientRepository registeredClientRepository() {
    RegisteredClient registeredClient = RegisteredClient.withId(UUID.randomUUID().toString())
        .clientId("articles-client")
        .clientSecret("{noop}secret")
        .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
        .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
        .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)
        .redirectUri("http://127.0.0.1:8080/login/oauth2/code/articles-client-oidc")
        .redirectUri("http://127.0.0.1:8080/authorized")
        .scope(OidcScopes.OPENID)
        .scope("articles.read")
        .build();

    return new InMemoryRegisteredClientRepository(registeredClient);
}

```

A Spring Security filter chain for the [Protocol Endpoints](#).

The properties we're configuring are:

- Client ID – Spring will use it to identify which client is trying to access the resource
- Client secret code – a secret known to the client and server that provides trust between the two
- Authentication method – in our case, we'll use basic authentication, which is just a username and password
- Authorization grant type – we want to allow the client to generate both an authorization code and a refresh token
- Redirect URI – the client will use it in a redirect-based flow
- Scope – this parameter defines authorizations that the client may have. In our case, we'll have the required *OidcScopes.OPENID* and our custom one, *articles.read*

Configuring a bean to apply the default OAuth security and generate a default form login page:

```

@Bean
@Order(Ordered.HIGHEST_PRECEDENCE)
public SecurityFilterChain authServerSecurityFilterChain(HttpSecurity http) throws Exception {
    OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);
    return http.formLogin(Customizer.withDefaults()).build();
}

```

A Spring Security filter chain for [authentication](#).

Each authorization server needs its signing key for tokens to keep a proper boundary between security domains.

```

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

1 usage
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();
}

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

```


Except for the signing key, each authorization server needs to have a unique issuer URL as well. We'll set it up as a localhost alias for *http://auth-server* on port 9000 by creating the *ProviderSettings* bean:

```

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

```

We will have to add “127.0.0.1 auth-server” entry in our /etc/hosts file

 hosts - Notepad

File Edit Format View Help

```
# Copyright (c) 1993-2009 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#       102.54.94.97       rhino.acme.com           # source server
#       38.25.63.10       x.acme.com               # x client host
#
# localhost name resolution is handled within DNS itself.
#       127.0.0.1         localhost
#       ::1               localhost
127.0.0.1 auth-server
```

Enabling the Spring Web Security module with `@EnableWebSecurity` annotated configuration

```
@EnableWebSecurity
public class DefaultSecurityConfig {

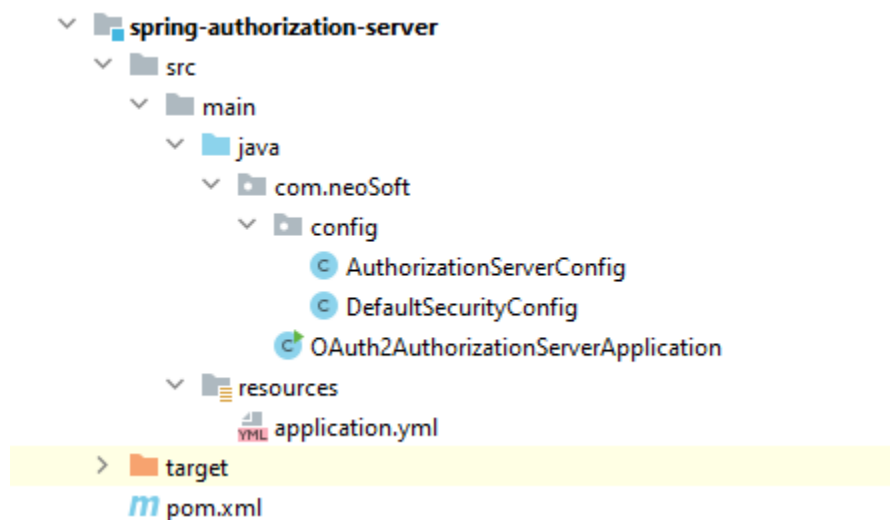
    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeRequests(authorizeRequests ->
            | authorizeRequests.anyRequest().authenticated()
            | )
            .formLogin(withDefaults());
        return http.build();
    }

    @Bean
    UserDetailsService users() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("admin")
            .password("password")
            .roles("USER")
            .build();
        return new InMemoryUserDetailsManager(user);
    }
}
```

Here we're calling `authorizeRequests.anyRequest().authenticated()` to require authentication for all requests. We're also providing a form-based authentication by invoking the `formLogin(defaults())` method.

Defining set of example users in `UserDetailsService` bean

Final Structure of spring-authorization-server



2.2.2 Resource Server

Creating a resource server that will return a list of articles from the GET endpoint. The endpoints should allow only requests that are authenticated against our OAuth server.

Dependencies

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>2.5.4</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
  <version>2.5.4</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
  <version>2.5.4</version>
</dependency>
```

Configuration

We need to set up the proper URL for our authentication server with the host and the port we've configured in the *ProviderSettings* bean earlier:

```
server:
  port: 8090

logging:
  level:
    root: INFO
    org.springframework.web: INFO
    org.springframework.security: INFO
    org.springframework.security.oauth2: INFO

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


Configuring web security with @EnableWebSecurity

To explicitly state that every request to article resources should be authorized and have the proper articles.read authority

```
@EnableWebSecurity
public class ResourceServerConfig {

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.mvcMatcher( mvcPattern: "/articles/**") HttpSecurity
            .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .mvcMatchers( ...patterns: "/articles/**") ExpressionUrlAuthorizationConfigurer<...>.MvcMatchersAuthorizedUrl
            .access( attribute: "hasAuthority('SCOPE_articles.read')") ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .and() HttpSecurity
            .oauth2ResourceServer() OAuth2ResourceServerConfigurer<HttpSecurity>
            .jwt();
        return http.build();
    }
}
```

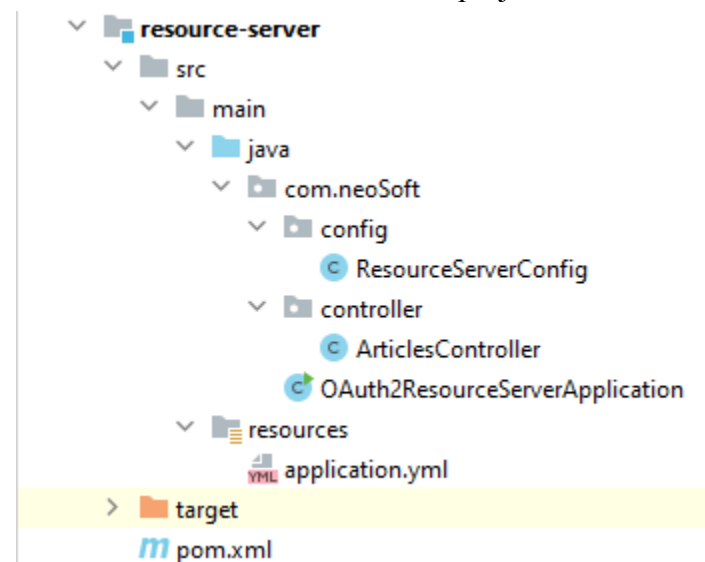
The `oauth2ResourceServer()` method will configure the OAuth server connection based on the application.yml configuration.

Creating REST controller that will return a list of articles under the GET /articles endpoint:

```
@RestController
public class ArticlesController {

    @GetMapping("/articles")
    public String[] getArticles() { return new String[]{"Article 1", "Article 2", "Article 3"}; }
}
```

Final structure of Resource Server project



2.2.3 API Client

Creating REST API client that will fetch the list of articles from the resource server

Dependencies

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>2.5.4</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
  <version>2.5.4</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-client</artifactId>
  <version>2.5.4</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webflux</artifactId>
  <version>5.3.9</version>
</dependency>
<dependency>
  <groupId>io.projectreactor.netty</groupId>
  <artifactId>reactor-netty</artifactId>
  <version>1.0.9</version>
</dependency>
```

Configuration

```
server:
  port: 8080

logging:
  level:
    root: INFO
    org.springframework.web: INFO
    org.springframework.security: INFO
    org.springframework.security.oauth2: INFO

spring:
  security:
    oauth2:
      client:
        registration:
          articles-client-oidc:
            provider: spring
```

```

        client-id: articles-client
        client-secret: secret
        authorization-grant-type: authorization_code
        redirect-uri:
"http://127.0.0.1:8080/login/oauth2/code/{registrationId}"
        scope: openid
        client-name: articles-client-oidc
articles-client-authorization-code:
    provider: spring
    client-id: articles-client
    client-secret: secret
    authorization-grant-type: authorization_code
    redirect-uri: "http://127.0.0.1:8080/authorized"
    scope: articles.read
    client-name: articles-client-authorization-code
provider:
spring:
    issuer-uri: http://auth-server:9000

```

Creating WebClient instance to perform HTTP requests to our resource server

```

@Bean
WebClient webClient(OAuth2AuthorizedClientManager authorizedClientManager) {
    ServletOAuth2AuthorizedClientExchangeFilterFunction oauth2Client =
        new ServletOAuth2AuthorizedClientExchangeFilterFunction(authorizedClientManager);
    return WebClient.builder()
        .apply(oauth2Client.oauth2Configuration())
        .build();
}

```

The WebClient requires an *OAuth2AuthorizedClientManager* as a dependency. Following is default implementation

```

@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 configuring web security

```
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorizeRequests ->
                authorizeRequests.anyRequest().authenticated()
            )
            .oauth2Login(oauth2Login ->
                oauth2Login.loginPage("/oauth2/authorization/articles-client-oidc"))
            .oauth2Client(withDefaults());
        return http.build();
    }
}
```

Configuring WebClient to send an HTTP request to our resource server

```
@RestController
public class ArticlesController {

    1 usage
    @Autowired
    private WebClient webClient;

    @GetMapping(value = "/articles")
    public String[] getArticles(
        @RegisteredOAuth2AuthorizedClient("articles-client-authorization-code") OAuth2AuthorizedClient authorizedClient
    ) {
        return this.webClient
            .get() RequestHeadersUriSpec<capture of ?>
            .uri(s: "http://127.0.0.1:8090/articles") capture of ?
            .attributes(oauth2AuthorizedClient(authorizedClient))
            .retrieve() ResponseSpec
            .bodyToMono(String[].class) Mono<String[]>
            .block();
    }
}
```

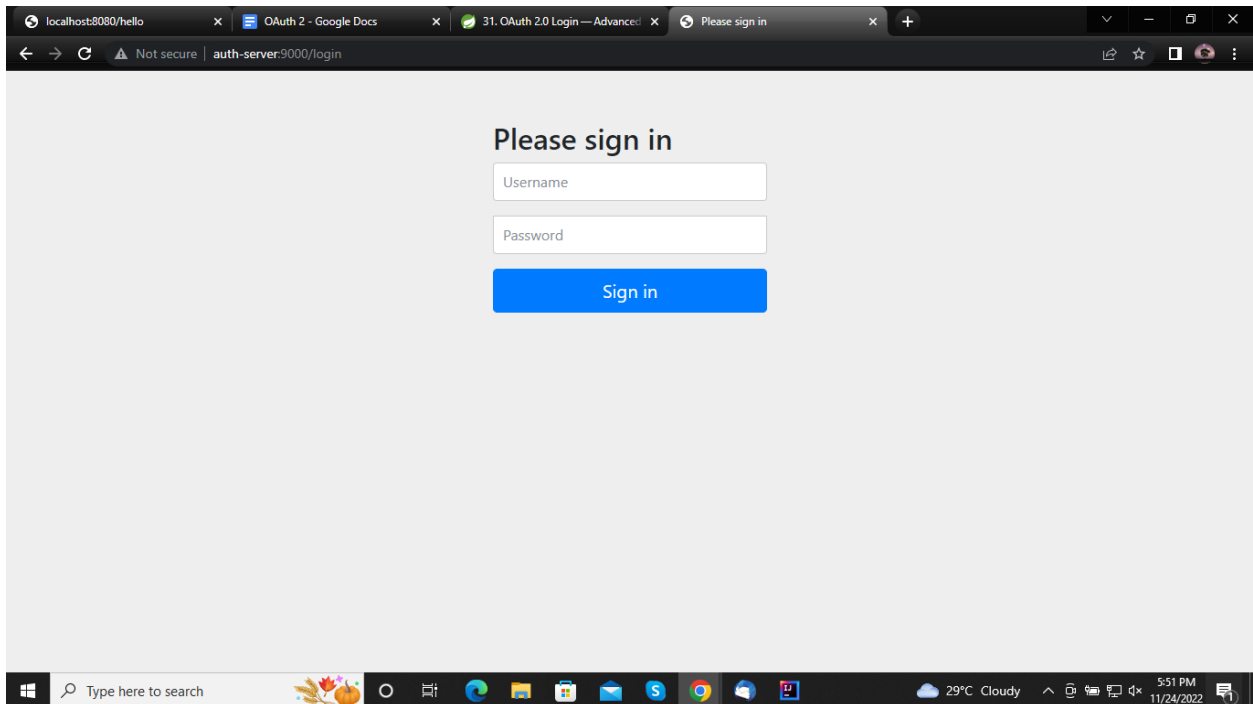
In the above example, we're taking the OAuth authorization token from the request in a form of *OAuth2AuthorizedClient* class. It's automatically bound by Spring using the *@RegisteredOAuth2AuthorizedClient* annotation with proper identification. In our case, it's pulled from the *article-client-authorization-code* that we configured previously in the *.yaml* file.

This authorization token is further passed to the HTTP request.

2.2.4 Running Our Project

Go in the browser and try to access <http://127.0.0.1:8080/articles> page.

We will be redirected to <http://auth-server:9000/login> URL.



After providing the proper username and password, the authorization server will redirect us back to the requested URL, the list of articles.

Further requests to the articles endpoint won't require logging in, as the access token will be stored in a cookie.

Ref [Spring Security OAuth Authorization Server | Baeldung](#)
Documentation [Getting Started](#)