

MICROSERVICES ADVANCED



[MICROSERVICES SECURITY USING OAUTH 2.0 AND JWT, API
GATEWAY, AWS CLOUD]

ASPIRE-RAMESH

Table of Contents

1. OAUTH 2.0.....	3
Introduction	3
OAuth Roles	4
Client Registration	5
Client Identifier.....	5
Client Secret	5
Client Authentication.....	5
Protocol Endpoints	6
Authorization Endpoint.....	6
Token Endpoint	7
Scope.....	7
OAuth Protocol Flow	8
Grant Types	8
Authorization Code Grant Type.....	9
Implicit Grant Type	13
Password Grant Type	15
Client Credentials Grant Type.....	17
Refresh Token Grant Type.....	19
Redis Token Store	21
Database Token Store.....	23
Separating Authorization and Resource servers	26
Authorization Server.....	27
Resource Server.....	27
2. OAuth 2.0 CLIENT	29
Client Authorization Code.....	29
Client Password	32
Client Client-Credentials	33
Client Implicit	35
3. Microservices Security With OAuth.....	39
Fares MicroService	39
Search Microservice	42
Booking Micorservice	42

CheckIn Microservice.....	44
PSS Website.....	45
4. JSON Web Token (JWT).....	50
Introduction	50
Header.....	50
Payload.....	51
Signature	51
Verifying JWT.....	52
Microservices with JWT	52
Authorization Server with Symmetric Key	53
MicroServices with Symmetric Key	55
Authorization Server with Asymmetric Key	57
MicroServices with Asymmetric Key.....	58
5. API Gateway using zuul.....	60
6. API Gateway and OAuth.....	65
7. Microservices with AWS	72

1. OAUTH 2.0

Introduction

OAuth2 is both **Authentication(AuthN)** and **Authorization(AuthZ) framework** that enables third-party application (such as Redbus) to automatically login to third-party application by using Twitter or Facebook or LinkedIn or Google or GitHub credentials i.e., the OAuth works by **delegating** user authentication process to Twitter or Facebook or LinkedIn or Google or GitHub and then automatic login to third-party application after **authorizing** third-party application.

Through OAuth login, our application obtains **limited access** on Twitter or Facebook or LinkedIn or Google or GitHub resources than direct login to social.

Microservices architecture says decompose single project into many smaller services. But we cannot ask every end user to sign-up and sign-in to every microservice. Instead the end user prefers single sign-on irrespective of number of microservices.

In case of single sign-on, the **common id** is required to represent user's identity.

Different microservices will run on different embedded servers. Hence there is no common session id. That means we cannot use session id across different microservices to represent user's identity.

Instead of using session id, rather use **oauth token** which is common across all microservices to represent user's identity.

The oauth token was generated by **Authorization server** and validated by **Resource server**.

Both Authorization server and Resource server together called as **OAuth provider**.

The microservices projects can use either existing OAuth providers (such as Twitter or Facebook or LinkedIn or Google or GitHub) **OR** implement our own OAuth Provider.

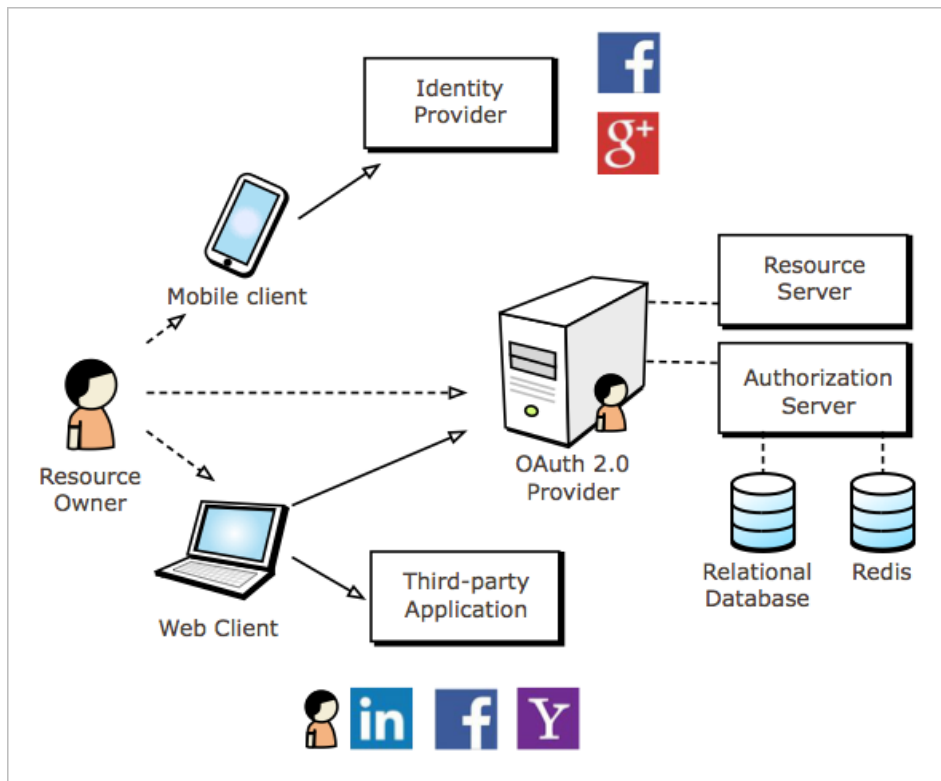
Since each microservice should perform token validation hence it is advised to implement our own OAuth provider. Otherwise if we use existing OAuth providers (such as Twitter or Facebook or LinkedIn or Google or GitHub) then every microservice should contact them for token validation in internet which is expensive. **Hence in microservices projects, it is recommended to implement our own OAuth provider.**

Conclusion: OAuth 2.0 is a delegation protocol, a means of letting someone who controls a resource allow a software application to access that resource on their behalf without impersonating them.

OAuth Roles

The OAuth 2.0 specification defines **four roles**:

1. Authorization Server
2. Resource Server
3. Resource Owner
4. Client Application (Third-party application such as Redbus)



The **Resource Owner** delegates authority for third-party application to use resources on resource owner's behalf. The resource owner may or may not be same as end user. In case of Redbus application, the resource owner is same as end user. But in case of Microservice projects, the resource owner protects resources in microservices and end user protects web pages in website. Hence resource owner and end user are different in microservice projects. **Hence both Resource Owner Authentication and End User Authentication required in microservice projects.**

The **Authorization server** generates access token only if resource owner successfully **authenticated** and **authorized**. The Authorization Server issues access token to client application only if client application successfully authenticated.

The **Resource Server** validates access token to protect resources.

The combination of both the Authorization Server and Resource Server together referred to as the OAuth Provider.

We can separate Authorization Server and Resource server. In this case, Redis or RDBMS store is used to share token.

In microservice projects there are multiple micro services. Each micro service contains a few resources. These resources need to be protected by Resource server. Hence every microservice needs resource

server to protect resources by validating the token i.e., multiple resource servers are needed. But only one authorization server is sufficient to issue a common token for all microservices. Hence implement one authorization server independently and include a separate resource server in each microservice. **So, microservice projects need single authorization server and multiple resource servers.** Hence in microservice projects, the authorization server and resource server are separated.

The **Client application** makes protected resource requests on behalf of the resource owner and with its authorization.

Client Registration

Before initiating the protocol, the client application (such as Redbus) should be registered with the OAuth provider (such as facebook).

The client registration is needed by OAuth provider in order to know whom (client application) to issue access token. Also, the client registration is needed by client application in-order to know which resources can be allowed to access by OAuth provider.

When registering a client, the client developer shall:

- 1) Specify the client type as confidential or public
- 2) Provide its client application redirection URI(s)
- 3) Include any other information required by the authorization server (such as application name, website, description, logo image, the acceptance of legal terms).

Client Identifier

After client registration, a client identifier is issued to client application.

Samples:

App ID: 596210484124781 (in case of Facebook)

clientapp (in our microservices advanced course)

Client Secret

After client registration, a client secret is issued to client application.

Samples:

App Secret: 126804b952a0ad7d193e8d6a2b982632 (in case of Facebook)

“123456” (in our microservices advanced course)

Client Authentication

The client application (such as Redbus) should be authenticated itself with OAuth provider (such as facebook) with the help of App ID and App Secret. This process is called as **client authentication**.

The client authentication is needed to get OAuth token from OAuth provider.

client_id

The client identifier issued to the client application during the client application registration process.

This is same as App ID in case of facebook.

client_secret

The client secret issued to the client application during the client application registration process. This is same as App Secret in case of facebook.

Conclusion so far: Microservice projects need 3 authentications such as Resource Owner Authentication, End User Authentication and Client Authentication.

Protocol Endpoints

The authorization process utilizes two authorization server endpoints:

- 1) **Authorization endpoint (/oauth/authorize)**
- 2) **Token endpoint (/oauth/token)**

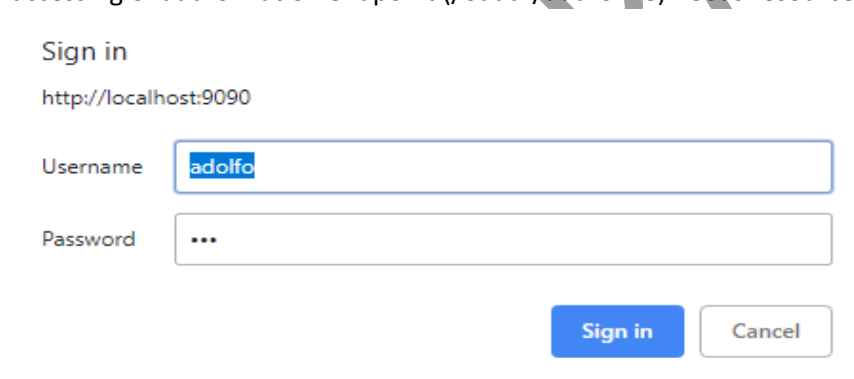
Authorization Endpoint

The authorization endpoint (/oauth/authorize) is used by the client application to obtain authorization from the resource owner. **The accessing of this end point needs resource owner authentication.**

Below is the partial authorization request url:

http://localhost:9090/oauth/authorize?client_id=clientapp&...

The above url will prompt login pop-up to enter resource owner's username and password because accessing of authorization endpoint (/oauth/authorize) needs resource owner authentication.



In case of microservice projects, we can define resource owner's credentials in application.properties file across all microservices.

security.user.name=adolfo

security.user.password=123

After authentication, we will get authorization page as below:

OAuth Approval

Do you authorize 'clientapp' to access your protected resources?

- scope.read_profile: ☒ Approve ☐ Deny

Authorize

Token Endpoint

The token endpoint (**/oauth/token**) is used by the client application to generate and obtain an access token from the authorization server. **The accessing of this end point needs client authentication.**

Below is the partial access token request url:

```
curl -X POST http://localhost:9090/oauth/token --user clientapp:123456 ...
```

Below is the access token response:

```
{
  "access_token":"1579ee4d-8b42-4caf-a89d-b789b697a8bd",
  "token_type":"bearer",
  "expires_in":43199,
  ...
}
```

Access token acts as a credential to access oauth protected resources.

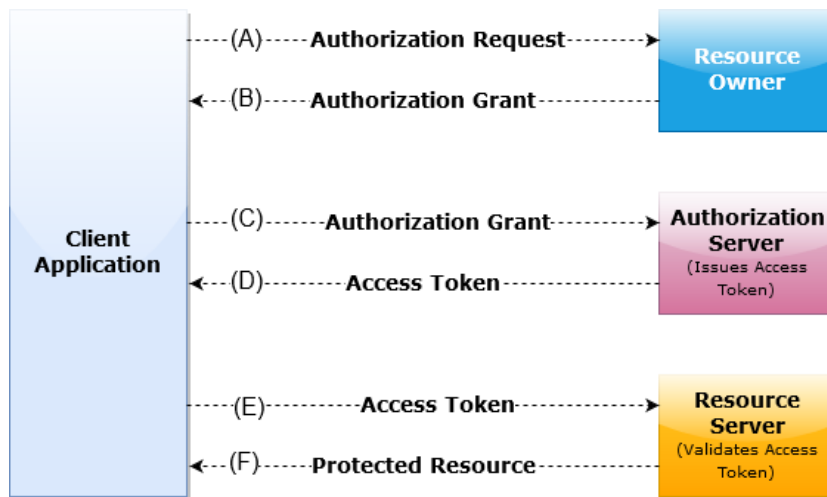
Scope

The authorization and token endpoints allow the client application to specify the scope of the access request using the "scope" request parameter. In turn, the authorization server uses the "scope" response parameter to inform the client application of the scope of the access token issued.

The value of the scope parameter is expressed as a list of space delimited, case-sensitive strings.

The strings are defined by the authorization server.

OAuth Protocol Flow



- (A) The client application requests **authorization** from the resource owner by presenting resource owner's credentials (adolfo/123).
- (B) The client application receives an **authorization grant**, which is a credential representing the resource owner's authorization, expressed using one of four grant types.
- (C) The client application requests an **access token** by presenting **both** client credentials (client-id & client-secret) as well as authorization grant with the authorization server.
- (D) The authorization server authenticates the client application and validates the authorization grant, and if valid, **issues an access token**.
- (E) The client application **requests the oauth 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.

Note: Authorization grant is a generalized term which means **give approval** to our client application to get an access token from authorization server.

Environment

- 1) Spring 4 or above
- 2) Spring Boot
- 3) Maven
- 4) Spring Security
- 5) **Spring OAuth 2.0**
- 6) STS (Spring Tool Suit)

Grant Types

To get an access token, the client application obtains authorization from the resource owner. The authorization is expressed in the form of an authorization grant, which the client application uses to get an access token. OAuth defines five grant types:

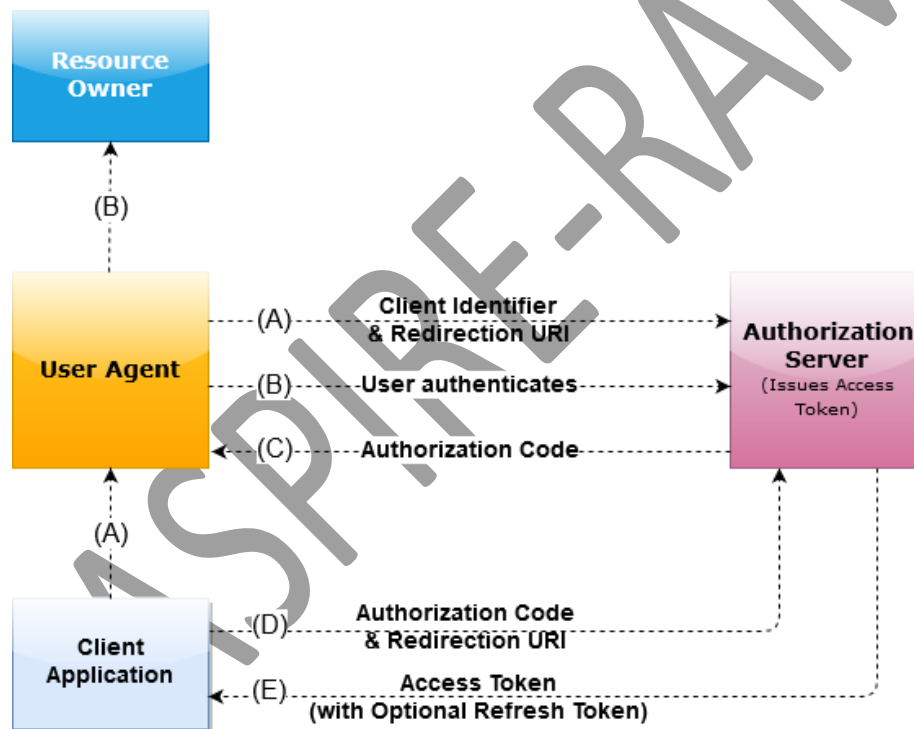
1. Authorization_code
2. Implicit
3. Resource owner password credentials
4. Client credentials
5. Refresh Token

Authorization Code Grant Type

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

Before directing the resource owner back to the client application 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 application.**

The authorization code can be used as an authorization grant to obtain an access token.



(A) The client application initiates the flow by directing the resource owner's user-agent to the **authorization endpoint** (/oauth/authorize). The client application includes its client identifier, scope, and optionally redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

(B) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

(C) Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client application using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an **authorization code**.

(D) The client application requests an access token from the authorization server by using **token endpoint (/oauth/token)** by including the authorization code received in the previous step. When making the request, the client application authenticates with the authorization server by presenting client-id and client-secret. The client application includes the redirection URI which was used to obtain the authorization code for **verification**.

(E) The authorization server authenticates the client application, validates the authorization code, and ensures that the redirection URI received matches the URI used to redirect the client in step (C). If valid, the authorization server responds back with an **access token** and, optionally, a refresh token.

The following steps will guide us to implement our own Authorization Server and Resource Server using Spring Security OAuth2.

Step1: Import 'auth-code-server' project from 'OAuthWorkspace'.

Step2: Ensure that Spring Security OAuth2 dependency added in pom.xml file.

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

Step3: Ensure that resource owner credentials added in application.properties file.

```
#Resource owner credentials
security.user.name=adolfo
security.user.password=123
server.port=9090
```

Step4: As we want to protect the user's resources through OAuth 2.0, we need to create something to be protected.

```
package com.packt.example.authcodeserver.api;
@Controller
public class UserController {
    @RequestMapping("/api/profile")
    public ResponseEntity<UserProfile> profile() {
        User user = (User) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
        String email = user.getUsername() + "@mailinator.com";

        UserProfile profile = new UserProfile();
        profile.setName(user.getUsername());
        profile.setEmail(email);
        return ResponseEntity.ok(profile);
    }
}
```

```
}
```

Step5: Configure Authorization Server to generate access token and optionally refresh token.

```
package com.packt.example.authcodeserver.config;
```

```
@Configuration
```

```
@EnableAuthorizationServer
```

```
public class OAuth2AuthorizationServer extends AuthorizationServerConfigurerAdapter {
```

```
    @Override
```

```
    public void configure (ClientDetailsServiceConfigurer clients) throws Exception {
```

```
        clients.inMemory()
```

```
            .withClient("clientapp").secret("123456") //client authentication details
```

```
            .redirectUri("http://localhost:9000/callback") //optional
```

```
            .authorizedGrantTypes("authorization_code")
```

```
            .scopes("read_profile");
```

```
    }
```

```
}
```

Redirection endpoint used by the authorization server to redirect resource owner back to client application.

If we preconfigure redirectUri in code then supplying redirect_uri in authorization request url and token request url is not required.

If we don't preconfigure redirectUri in code then supplying redirect_uri in both authorization request and token request urls is required.

By adding @EnableAuthorizationServer annotation, the following beans have been configured:

```
    AuthorizationEndpoint
```

```
    TokenEndpoint
```

Step6: Configure Resource Server to protect resources by validating access token.

```
package com.packt.example.authcodeserver.config;
```

```
@Configuration
```

```
@EnableResourceServer
```

```
public class OAuth2ResourceServer extends ResourceServerConfigurerAdapter {
```

```
    @Override
```

```
    public void configure(HttpSecurity http) throws Exception {
```

```
        http.authorizeRequests().anyRequest().authenticated().and()
```

```
            .requestMatchers().antMatchers("/api/**");
```

```
    }
```

```
}
```

By adding @EnableResourceServer annotation, we are importing some configurations which will add the filter OAuth2AuthenticationProcessingFilter within the Spring Security's FilterChain configuration. **This filter will be in charge of starting the access token validation process for any endpoint that matches the /api/** pattern.**

Step7: Start OAuth Provider by run as a standalone application.

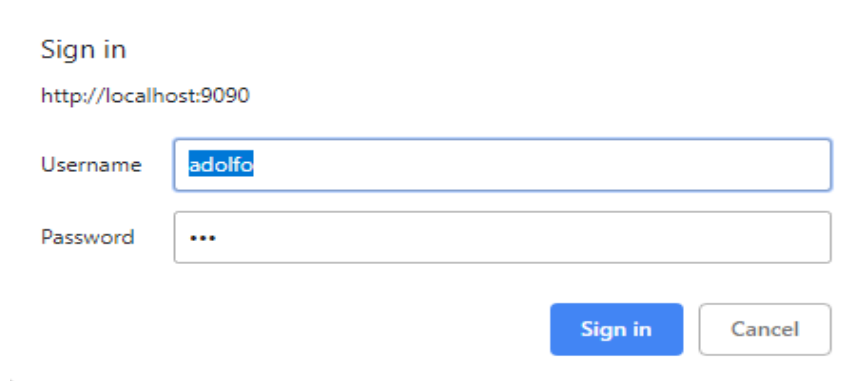
This grant type uses both authorization endpoint(/oauth/authorize) as well as token endpoint(/oauth/token).

The client application informs the authorization server of the desired authorization grant by using the “response_type” parameter.

The value of ‘response_type’ parameter must be “code” in case of authorization_code grant type.

a) Authorization Request

http://localhost:9090/oauth/authorize?client_id=clientapp&redirect_uri=http://localhost:9000/callback&response_type=code&scope=read_profile



The above login popup is used for resource owner authentication.

After resource owner authentication, the resource owner should authorize client application to access our protected resources.

OAuth Approval

Do you authorize 'clientapp' to access your protected resources?

• scope.read_profile: ☒ Approve ☐ Deny

b) Authorization Response

<http://localhost:9000/callback?code=w9rw3x>

Note: Client application cannot reuse authorization code more than once.

c) Access Token Request

```
curl -X POST http://localhost:9090/oauth/token --user clientapp:123456 -H "Content-Type: application/x-www-form-urlencoded" -d
```

```
"grant_type=authorization_code&redirect_uri=http://localhost:9000/callback&scope=read_profile&code=w9rw3x"
```

d) Access Token Response

```
{
  "access_token": "1579ee4d-8b42-4caf-a89d-b789b697a8bd",
  "token_type": "bearer",
  "expires_in": 43199,
```

```

    "scope":"read_profile"
  }

```

Note: For this grant type, refresh token is applicable but optional.

e) Resource Endpoint Request

To access resource endpoint, we have to present a valid access token retrieved after the resource owner's authorization to share their profile.

```
curl -X GET http://localhost:9090/api/profile -H "Authorization: Bearer 1579ee4d-8b42-4caf-a89d-b789b697a8bd"
```

f) Resource Endpoint Response

```
{"name":"adolfo","email":"adolfo@mailinator.com"}
```

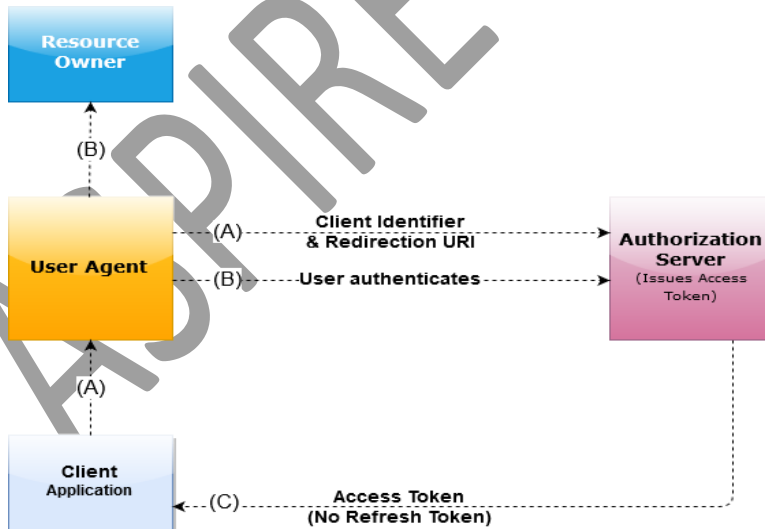
Implicit Grant Type

In the implicit flow, instead of issuing an authorization code to the client application, rather the client application is issued an access token directly after resource owner's authentication and authorization. The resource owner's authentication and authorization **implicitly** used as an authorization grant to obtain an access token.

This grant type uses only authorization endpoint but not token endpoint (since an access token is issued directly) i.e., unlike the `authorization_code` grant type, in which the client application makes separate requests for authorization and for an access token, the client application receives the access token as the result of the authorization request.

Since token endpoint is not needed hence client authentication is not required in case of Implicit grant type.

It does not support the issuance of refresh tokens since client application never knows anything about resource owner.



(A) The client application initiates the flow by directing the resource owner's user-agent to the authorization endpoint (`/oauth/authorize`). The client application includes its client identifier, requested scope, and optionally redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

(B) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

(C) Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client application using the redirection URI provided earlier. The redirection URI includes the **access token**.

The Implicit grant type improves the responsiveness and efficiency, since it reduces the number of round trips required to obtain an access token.

The following steps will guide us to configure an Authorization Server using Spring Security OAuth2 with Implicit grant type.

Step1: Copy 'auth-code-server' project and rename as '**implicit-server**'

Step2: Change grant type as "**implicit**" in Authorization Server configuration class. Also remove client_secret because the token end point (/oauth/token) is not required in case of implicit grant type.

```
package com.packt.example.authcodeserver.config;

@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends AuthorizationServerConfigurerAdapter {
    @Override
    public void configure (ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
            .withClient("clientapp")
            //.secret("123456") //remove
            .redirectUris("http://localhost:9000/callback") //optional
            .authorizedGrantTypes("implicit")
            .scopes("read_profile");
    }
}
```

Step3: Start OAuth Provider by running as a standalone application.

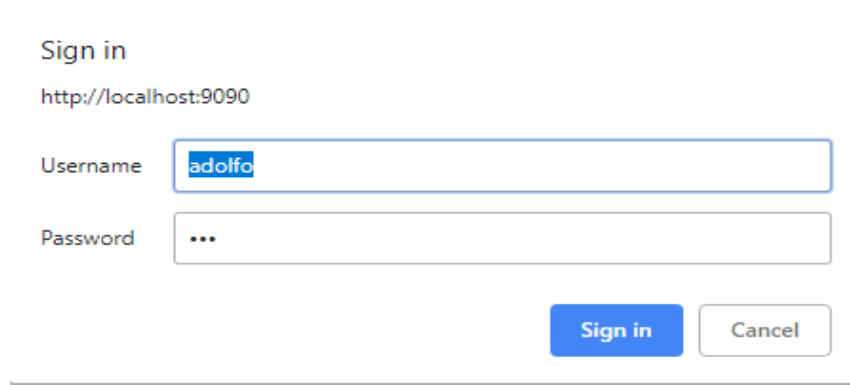
This grant type uses only authorization endpoint(/oauth/authorize) but not token endpoint(/oauth/token). Hence when issuing an access token during the implicit grant flow, the authorization server does not authenticate the client application i.e., the client authentication is not required.

The client application informs the authorization server of the desired grant type using the "**response_type**" parameter.

The value of 'response_type' parameter must be "**token**" in case of implicit grant type.

a) Authorization Request

http://localhost:9090/oauth/authorize?client_id=clientapp&redirect_uri=http://localhost:9000/callback&response_type=token&scope=read_profile



The above login popup is used for resource owner authentication.

After resource owner authentication, the resource owner should authorize client application to access our protected resources.

OAuth Approval

Do you authorize 'clientapp' to access your protected resources?

- scope.read_profile: ☒ Approve ☐ Deny

Authorize

b) Access Token Response

http://localhost:9000/callback#access_token=2bd425f8-9218-43bf-bd26-3fb970a5cd63&token_type=bearer&expires_in=43119

c) Resource Endpoint Request

```
curl -X GET http://localhost:9090/api/profile -H "Authorization: Bearer 2bd425f8-9218-43bf-bd26-3fb970a5cd63"
```

d) Resource Endpoint Response

```
{"name":"adolfo","email":"adolfo@mailinator.com"}
```

Password Grant Type

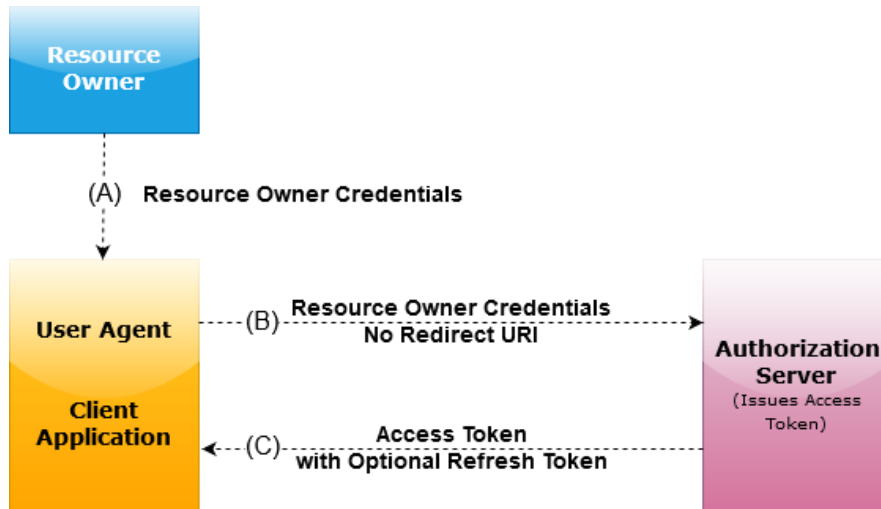
The resource owner's credentials (username and password) can be directly used as an authorization grant to obtain an access token.

The resource owner password credentials grant type is suitable in cases where **the resource owner has a trust relationship with the client application** i.e., this grant type should only be used when there is a high degree of trust between the resource owner and the client application.

This grant type is suitable for client applications capable of obtaining the resource owner's credentials (username and password).

The authorization server should take special care when enabling this grant type and only allow it when other flows are not viable.

It is also used to migrate existing client applications using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token.



- (A) The resource owner provides the client application with its username (spring.user.name) and password (spring.user.password).
- (B) The client application requests an access using token endpoint (/oauth/token) by including both resource owner credentials and client application credentials.
- (C) The authorization server authenticates the client application (using client-id & client-secret) and validates the resource owner credentials (using spring.user.name & spring.user.password), and if valid, issues an access token.

The following steps will guide us to configure an Authorization Server using Spring Security OAuth2 with Password grant type.

Step1: Copy 'auth-code-server' project and rename as **'password-server'**

Step2: Change grant type as **"password"** in Authorization Server configuration class. Also remove redirect_uri.

```

package com.packt.example.authcodeserver.config;
@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends AuthorizationServerConfigurerAdapter {
    @Override
    public void configure (ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
            .withClient("clientapp").secret("123456")
            //redirectUri("http://localhost:9000/callback") //remove
            .authorizedGrantTypes("password")
            .scopes("read_profile");
    }
}

@Autowired
private AuthenticationManager authenticationManager;
  
```

```

@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
    endpoints.authenticationManager(authenticationManager);
}
}

```

If we run the application, we will face the following error when trying to request an access token. Such an error occurs because the Password grant type requires us to declare an AuthenticationManager inside the OAuth2AuthorizationServer configuration class:

```

{
  "error": "unsupported_grant_type",
  "error_description": "Unsupported grant type: password"
}

```

Step3: Start OAuth Provider by run as a standalone application.

This grant type uses only token endpoint(/oauth/token) but not authorization endpoint(/oauth/authorize).

a) Access Token Request

```

curl -X POST http://localhost:9090/oauth/token --user clientapp:123456 -H "Accept: application/json" -H "Content-Type: application/x-www-form-urlencoded" -d "grant_type=password&username=adolfo&password=123&scope=read_profile"

```

b) Access Token Response

```

{
  "access_token": "2db1bcda-f2e0-4334-a9da-04919f3586ff",
  "token_type": "bearer",
  "expires_in": 43199,
  "scope": "read_profile"
}

```

Note: The refresh token is applicable (but optional) in case of 'password' grant type.

c) Resource Endpoint Request

```

curl -X GET http://localhost:9090/api/profile -H "Authorization: Bearer 2db1bcda-f2e0-4334-a9da-04919f3586ff"

```

d) Resource Endpoint Response

```

{"name": "adolfo", "email": "adolfo@mailinator.com"}

```

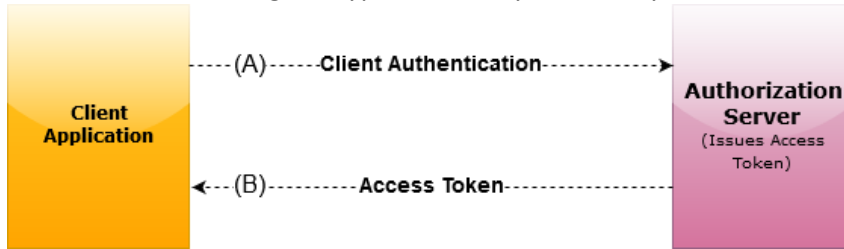
Client Credentials Grant Type

The client credentials (i.e., client-id and client-secret) can be directly used as an authorization grant to obtain an access token.

This grant type is supposed to be used by applications that do not access resources on behalf of any Resource Owner, but for their own purposes.

The client application can request an access token using only its client credentials (client-id & client-secret) without resource owner credentials.

The client credentials grant type MUST only be used by confidential clients.



(A) The client application authenticates with the authorization server and requests an access token using token endpoint (/oauth/token).

(B) The authorization server authenticates the client application, and if valid, issues an access token.

The following steps will guide us to configure an Authorization Server using Spring Security OAuth2 with “client_credentials” grant type.

Step1: Copy ‘auth-code-server’ project and rename as ‘client-credentials-server’

Step2: Change grant type as “client_credentials” in Authorization Server configuration class. Also remove redirect_uri.

```
package com.packt.example.authcodeserver.config;
@Configuration
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends AuthorizationServerConfigurerAdapter {
    @Override
    public void configure (ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
            .withClient("clientapp").secret("123456")
            //.redirectUri("http://localhost:9000/callback") //remove
            .authorizedGrantTypes("client_credentials")
            .scopes("read_profile");
    }
}
```

Step3: Below two properties in application.properties file are not needed because client_credentials grant type uses only client credentials but not resource owner credentials.

```
#security.user.name=adolfo //remove
#security.user.password=123 //remove
server.port=9090
```

Step4: Start OAuth Provider by run as a standalone application.

This grant type uses only token endpoint(/oauth/token) but not authorization endpoint(/oauth/authorize).

a) Access Token Request

```
curl -X POST "http://localhost:9090/oauth/token" --user clientapp:123456 -d
"grant_type=client_credentials&scope=read_profile"
```

b) Access Token Response

```
{
  "access_token":"2d10e917-b0d1-4a07-b18b-99fcfea296b5",
  "token_type":"bearer",
  "expires_in":42831,
  "scope":"read_profile"
}
```

c) Resource Endpoint Request

```
curl -X GET http://localhost:9090/api/profile -H "authorization: Bearer 2d10e917-b0d1-4a07-
b18b-99fcfea296b5"
```

d) Resource Endpoint Response

```
{"name":"clientapp","email":"clientapp@mailinator.com"}
```

Refresh Token Grant Type

Which allows for a better user experience because the Resource Owner does not have to go through all the steps of authentication and authorization against the Authorization Server every time an access token expires.

The refresh token is applicable for `authorization_code` and `password` grant types.

The following steps will guide us to configure an Authorization Server using Spring Security OAuth2 with 'refresh_token' grant type.

Step1: Copy 'auth-code-server' project and rename as 'refresh-server'

Step2: Set grant types as ("authorization_code", "password", "refresh_token") in Authorization Server configuration class.

```
package com.packt.example.authcodeserver.config;
```

```
@Configuration
```

```
@EnableAuthorizationServer
```

```
public class OAuth2AuthorizationServer extends AuthorizationServerConfigurerAdapter {
```

```
    @Override
```

```
    public void configure (ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
```

```
            .withClient("clientapp").secret("123456")
```

```
            .redirectUri("http://localhost:9000/callback") //optional or not required depends on grant_type
```

```
            .authorizedGrantTypes("authorization_code", "password", "refresh_token")
```

```
            .accessTokenValiditySeconds(120)
```

```
            .refreshTokenValiditySeconds(3000)
```

```
            .scopes("read_profile");
```

```
    }
```

```
// adding authenticationManager because we are supporting password grant type
@Autowired
private AuthenticationManager authenticationManager;

@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints)throws Exception {
    endpoints.authenticationManager(authenticationManager);
}
}
```

Note: The refresh token will be issued if and only if “refresh_token” grant type is added to the authorized grant types list.

Step3: Start OAuth Provider by run as a standalone application.

a) Access Token Request

```
curl -X POST http://localhost:9090/oauth/token --user clientapp:123456 -H "Accept:
application/json" -H "Content-Type: application/x-www-form-urlencoded" -d
"grant_type=password&username=adolfo&password=123&scope=read_profile"
```

b) Access Token Response

```
{
  "access_token":"2db1bcda-f2e0-4334-a9da-04919f3586ff",
  "token_type":"bearer",
  "refresh_token":"79cbc7ae-d6d1-4323-976f-e822661517c7"
  "expires_in":43199,
  "scope":"read_profile"
}
```

c) Resource Endpoint Request and Response

```
curl -X GET http://localhost:9090/api/profile -H "authorization: Bearer 2db1bcda-f2e0-4334-
a9da-04919f3586ff "
{"name":"adolfo","email":"adolfo@mailinator.com"}
```

If we try same resource endpoint request after 120 seconds, we will get following error.

```
curl -X GET http://localhost:9090/api/profile -H "Authorization: Bearer 2db1bcda-f2e0-4334-
a9da-04919f3586ff "
{
  "error":"invalid_token",
  "error_description":"Access token expired: 2db1bcda-f2e0-4334-a9da-04919f3586ff "
}
```

d) Refresh Token Request

Now it's time to request for a new access token using the previously issued refresh token.

```
curl -X POST http://localhost:9090/oauth/token --user clientapp:123456 -H "Content-Type: application/x-www-form-urlencoded" -d "grant_type=refresh_token&refresh_token=79cbc7ae-d6d1-4323-976f-e822661517c7"
```

e) Refresh Token Response

```
{
  "access_token": "b71e1c15-5561-4db4-8f67-7ce9b0712d24",
  "token_type": "bearer",
  "refresh_token": "79cbc7ae-d6d1-4323-976f-e822661517c7",
  "expires_in": 119,
  "scope": "read_profile"
}
```

Note: The same refresh token with new access token is generated.

f) Resource Endpoint Request (try again)

```
curl -X GET http://localhost:9090/api/profile -H "authorization: Bearer b71e1c15-5561-4db4-8f67-7ce9b0712d24"
{"name": "adolfo", "email": "adolfo@mailinator.com"}
```

Redis Token Store

We use Redis to store access tokens.

Perform the following steps to set up Redis to store tokens:

Step1: Install Redis from softwares folder. Download the latest stable version from <https://redis.io/download> (redis-2.4.5-win32-win64.zip).

Extract redis and navigate to 'D:\MICROSERVICES ADVANCED\Softwares\redis-2.4.5-win32-win64\64bit' folder.

Start redis server by double click on 'redis-server.exe'

Optionally start redis client by double click on 'redis-cli.exe'

```
D:\MICROSERVICES ADVANCED\Softwares\redis-2.4.5-win32-win64\64bit\redis-cli.exe
redis 127.0.0.1:6379>
```

```
redis 127.0.0.1:6379> keys *
redis 127.0.0.1:6379> flushall
```

Step2: Copy 'auth-code-server' project and rename as '**redis-server**'

Step3: Add redis dependency in pom file.

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

```
<artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

Step4: Add below property in application.properties file
spring.redis.url=redis://localhost:6379

Step5: Until this point, we are configuring the Authorization Server to store both client credentials and tokens using the in-memory cache. To start using Redis, inject an instance of RedisConnectionFactory within OAuth2AuthorizationServer. With an instance of RedisConnectionFactory, we are ready to declare one bean of type TokenStore which uses Redis as the store strategy. Now that we have the RedisTokenStore, let's define the token store strategy by setting up the AuthorizationServerEndpointsConfigurer within the OAuth2AuthorizationServer class.

@Configuration

@EnableAuthorizationServer

```
public class OAuth2AuthorizationServer extends AuthorizationServerConfigurerAdapter {
    @Autowired
    private RedisConnectionFactory connectionFactory;

    @Bean
    public TokenStore tokenStore() {
        RedisTokenStore redis = new RedisTokenStore(connectionFactory);
        return redis;
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
        endpoints.tokenStore(tokenStore());
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
            .withClient("clientapp").secret("123456")
            .redirectUri("http://localhost:9000/callback") //optional
            .authorizedGrantTypes("authorization_code")
            .scopes("read_profile");
    }
}
```

Step5: To check how RedisTokenStore persists the access tokens and related data, start the redis-server application.

```
redis 127.0.0.1:6379> keys *
```

(empty list or set)

```
redis 127.0.0.1:6379>
```

Try to request for an access token using one of the authorized grant types declared

http://localhost:9090/oauth/authorize?client_id=clientapp&redirect_uri=http://localhost:9000/callback&response_type=code&scope=read_profile

```
curl -X POST http://localhost:9090/oauth/token --user clientapp:123456 -H "Content-Type: application/x-www-form-urlencoded" -d "grant_type=authorization_code&redirect_uri=http://localhost:9000/callback&scope=read_profile&code=w9rw3x"
```

```
redis 127.0.0.1:6379> keys *
```

```
1) "access:27b0d3a2-da68-42d7-9fdf-a67e21aba3ae"
2) "auth:27b0d3a2-da68-42d7-9fdf-a67e21aba3ae"
3) "auth_to_access:643c3ce5f7876962d689f63a66a48d83"
4) "client_id_to_access:clientapp"
5) "uname_to_access:clientapp:adolfo"
```

```
redis 127.0.0.1:6379>
```

Database Token Store

Database provides us with a storage strategy that is more production-like instead of using an in-memory registry.

The strategy presented in this section helps us how to use a Relational Database Management System (RDBMS) to store all client details and data related to tokens.

Step1: Copy auth-code-server and rename as '**rdbm-server**'

Step2: Add oracle and Jpa dependencies in pom.xml file

```
<dependency>
  <groupId>oracle</groupId>
  <artifactId>oracle-jdbc</artifactId>
  <version>11</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Step3: Add db properties in application.properties file

```
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager
```

Step4: Create tables in database

The following SQL commands to start using the oauth2 provided built-in schema to create table that will be used to store all the data related to the client registration:

```
create table oauth_client_details (  
  client_id VARCHAR2(256) PRIMARY KEY,  
  resource_ids VARCHAR2(256),  
  client_secret VARCHAR2(256),  
  scope VARCHAR2(256),  
  authorized_grant_types VARCHAR2(256),  
  web_server_redirect_uri VARCHAR2(256),  
  authorities VARCHAR2(256),  
  access_token_validity number(11),  
  refresh_token_validity number(11),  
  additional_information VARCHAR2(256),  
  autoapprove VARCHAR2(256)  
);
```

All the configurations which were defined in the OAuth2AuthorizationServer through the ClientDetailsServiceConfigurer instance, can now be done by inserting a new row into the oauth_client_details table by running the following SQL command:

```
INSERT INTO oauth_client_details  
  (client_id, resource_ids, client_secret, scope, authorized_grant_types,  
   web_server_redirect_uri, authorities, access_token_validity, refresh_token_validity,  
   additional_information, autoapprove)  
VALUES  
  ('clientapp', null, '123456',  
   'read_profile,read_posts', 'authorization_code',  
   'http://localhost:9000/callback',  
   null, 3000, -1, null, 'false');  
commit;
```

To store all the issued access tokens, create the following table:

```
create table oauth_access_token (  
  token_id VARCHAR2(256),  
  token blob,  
  authentication_id VARCHAR2(256) PRIMARY KEY,  
  user_name VARCHAR2(256),  
  client_id VARCHAR2(256),  
  authentication blob,  
  refresh_token VARCHAR2(256)  
);
```

To store authorization code, create the following table:

```
create table oauth_code (  
  code VARCHAR2(256),
```

authentication blob

);

Also run the following commands so that Spring Security OAuth2 can store the user's approvals and issue refresh tokens respectively:

```
create table oauth_approvals (
```

```
    userId VARCHAR2(256),
```

```
    clientId VARCHAR2(256),
```

```
    scope VARCHAR2(256),
```

```
    status VARCHAR2(10),
```

```
    expiresAt TIMESTAMP,
```

```
    lastModifiedAt TIMESTAMP
```

```
);
```

```
create table oauth_refresh_token (
```

```
    token_id VARCHAR2(256),
```

```
    token blob,
```

```
    authentication blob
```

```
);
```

Step5: Inject DataSource to OAuth2AuthorizationServer.java file. Also override the method presented in the following code to set up the ClientDetailsServiceConfigurer to use the present dataSource attribute. Now declare the following beans within the OAuth2AuthorizationServer class to define the TokenStore and ApprovalStore.

```
package com.packt.example.authcodeserver.config;
```

```
@Configuration
```

```
@EnableAuthorizationServer
```

```
public class OAuth2AuthorizationServer extends AuthorizationServerConfigurerAdapter {
```

```
    /*@Override
```

```
        public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
```

```
            clients.inMemory()
```

```
                .withClient("clientapp").secret("123456")
```

```
                .redirectUri("http://localhost:9000/callback")
```

```
                .authorizedGrantTypes("authorization_code").scopes("read_profile");
```

```
        }*/
```

```
        @Autowired
```

```
        private DataSource dataSource;
```

```
        @Override
```

```
        public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
```

```
            clients.jdbc(dataSource);
```

```
        }
```

```
        @Bean
```

```
        public TokenStore tokenStore() {
```

```
            return new JdbcTokenStore(dataSource);
```

```

    }
    @Bean
    public ApprovalStore approvalStore() {
        return new JdbcApprovalStore(dataSource);
    }
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
        endpoints.approvalStore(approvalStore())
            .tokenStore(tokenStore());
    }
}

```

Step6: Start the rdbm-server application.

Try to request for an access token using one of the authorized grant types declared

http://localhost:9090/oauth/authorize?client_id=clientapp&redirect_uri=http://localhost:9000/callback&response_type=code&scope=read_profile

```

curl -X POST http://localhost:9090/oauth/token --user clientapp:123456 -H "Content-Type:
application/x-www-form-urlencoded" -d
"grant_type=authorization_code&redirect_uri=http://localhost:9000/callback&scope=read_profile&
code=w9rw3x"

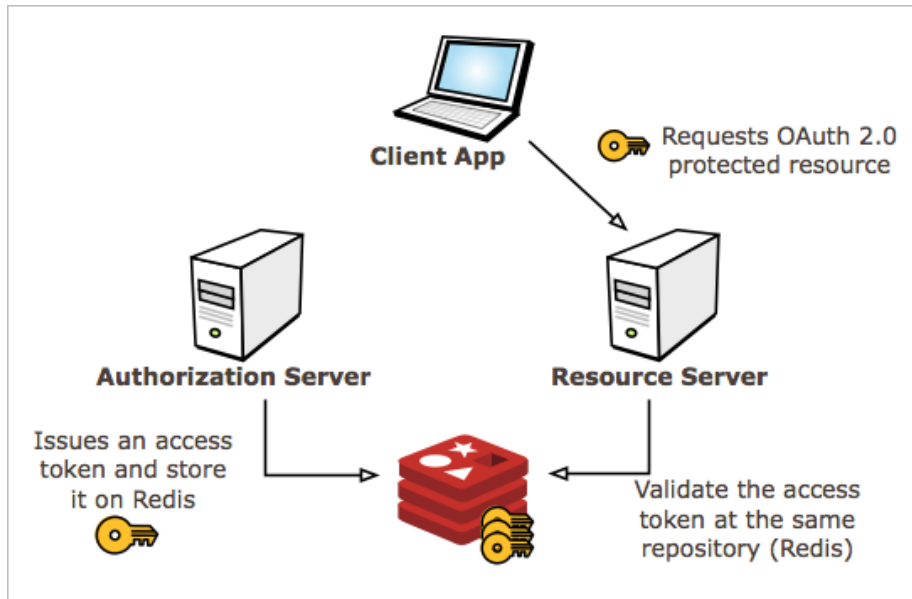
```

```
SQL:/> select utl_raw.cast_to_varchar2(dbms_lob.substr(TOKEN)) from oauth_access_token;
```

Separating Authorization and Resource servers

Dividing Authorization Server and Resource Server responsibilities into different projects.

As the Authorization Server is in charge of issuing access tokens and Resource Server has to validate access tokens, **both applications need a shared token store**. In that way, the Resource Server can query for an access token that was persisted in some token store by the Authorization Server. We will use Redis as a token store.



Authorization Server

The following steps are used to create a separate authorization server:

Step1: Copy 'redis-server' and rename as 'separate-authorization-server'

Step2: Remove Resource server configuration file from config package. Also remove UserController and UserProfile files from com.packt.example.authcodeserver.api package.

OAuth2ResourceServer.java	[removed]
com.packt.example.uthcodeserver.api	[removed]

Step3: Start redis server and then start separate-authorization-server project.

Resource Server

The following steps are used to create separate resource server.

Step1: Copy 'redis-server' and rename as 'separate-resource-server'

Step2: Change port number and remove resource owner credentials from application.properties file.

#security.user.name=adolfo

#security.user.password=123

server.port=**8081**

spring.redis.url=redis://localhost:6379

Step3: Remove authorization server configuration file from config package. Configure RedisConnectionFactory, RedisTokenStore and ResourceServerSecurityConfigurer in OAuth2ResourceServer.java.

@Configuration

@EnableResourceServer

```

public class OAuth2ResourceServer extends ResourceServerConfigurerAdapter {
    @Autowired
    private RedisConnectionFactory connectionFactory;

    @Bean
    public TokenStore tokenStore() {
        RedisTokenStore redis = new RedisTokenStore(connectionFactory);
        return redis;
    }

    @Override
    public void configure(ResourceServerSecurityConfigurer endpoints) throws Exception {
        endpoints.tokenStore(tokenStore());
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().authenticated().and()
            .requestMatchers().antMatchers("/api/**");
    }
}

```

As we can see, we are configuring a TokenStore the same way we did for the Authorization Server configuration. The difference is that we are setting the token store to an instance of ResourceServerSecurityConfigurer.

Step4: Start Resource server. Start redis server, if not already started, before starting resource server.

http://localhost:9090/oauth/authorize?client_id=clientapp&redirect_uri=http://localhost:9000/callback&response_type=code&scope=read_profile

```

curl -X POST http://localhost:9090/oauth/token --user clientapp:123456 -H "content-type: application/x-www-form-urlencoded" -d "grant_type=authorization_code&redirect_uri=http://localhost:9000/callback&scope=read_profile&code=w9rw3x"

```

```

curl -X GET http://localhost:8081/api/profile -H "authorization: Bearer ad1c7340-9459-4f93-80e4-d6e80d081b12"

```

2. OAUTH 2.0 CLIENT

In the previous chapter, all the interactions with the OAuth 2.0 Provider were made by using direct requests through Postman or CURL tool. This chapter will show us how to create oauth2 client applications to access oauth protected resources by using Spring Security OAuth2 and **OAuth2RestTemplate** class.

Client Authorization Code

This presents us with how to create a client application that interacts with the OAuth 2.0 Provider by using the `authorization_code` grant type.

Step1: Import 'client-authorization-code' from backup OAuthWorkspace.

Step2: This client application will provide a web interface to allow the user to see user profile. There will be two simple pages: the first one will just provide one link to allow the user to navigate to the profile's page, and the second page will show the user's profile.

```
#index.html
#dashboard.html
```

To process above html pages, make sure that `starter-thymeleaf` dependency added in `pom.xml` file.

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

Step3: Write spring security configuration file to provide end user authentication details. This authentication protects website i.e., this authentication is not meant for protecting resources.

```
package com.packt.example.clientauthorizationcode.security;
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/", "/index.html").permitAll()
            .anyRequest().authenticated().and()
            .formLogin().and()
            .logout().permitAll();
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user1@example.com").password("user1").roles("USER").and()
            .withUser("user2@example.com").password("user2").roles("USER");
    }
}
```

```
}
```

Step4: Write OAuth2.0 client configuration file.

```
package com.packt.example.clientauthorizationcode.oauth;
```

```
@Configuration
```

```
@EnableOAuth2Client
```

```
public class ClientConfiguration {
```

```
    @Autowired
```

```
    private OAuth2ClientContext oauth2ClientContext;
```

```
    @Bean
```

```
    public OAuth2ProtectedResourceDetails resourceDetails() {
```

```
        AuthorizationCodeResourceDetails resourceDetails = new AuthorizationCodeResourceDetails();
```

```
        // resourceDetails.setId("oauth2server");
```

```
        //resourceDetails.setTokenName("oauth_token");
```

```
        resourceDetails.setClientId("clientapp");
```

```
        resourceDetails.setClientSecret("123456");
```

```
        resourceDetails.setGrantType("authorization_code");
```

```
        resourceDetails.setPreEstablishedRedirectUri("http://localhost:9000/callback");
```

```
        //resourceDetails.setScope(Arrays.asList("read_profile")); //If present then auto approved by OAuth2RestTemplate
```

```
        resourceDetails.setUserAuthorizationUri("http://localhost:9090/oauth/authorize");
```

```
        resourceDetails.setAccessTokenUri("http://localhost:9090/oauth/token");
```

```
        resourceDetails.setUseCurrentUri(false);
```

```
        resourceDetails.setClientAuthenticationScheme(AuthenticationScheme.header);
```

```
        return resourceDetails;
```

```
    }
```

```
    @Bean
```

```
    public OAuth2RestTemplate oauth2RestTemplate() {
```

```
        OAuth2RestTemplate template = new OAuth2RestTemplate(resourceDetails(), oauth2ClientContext);
```

```
        /*AccessTokenProviderChain provider = new AccessTokenProviderChain(Arrays.asList(  
            new AuthorizationCodeAccessTokenProvider());
```

```
        template.setAccessTokenProvider(provider);
```

```
        */
```

```
        return template;
```

```
    }
```

```
}
```

The **@EnableOAuth2Client** annotation imports another annotation called **OAuth2ClientConfiguration** which in turn configures the context needed for an access token request.

The **ClientConfiguration** class declares two other important elements for an OAuth 2.0 client which is implemented using Spring Security OAuth 2.0. They are **OAuth2ProtectedResourceDetails** and **OAuth2RestTemplate**.

The OAuth2ProtectedResourceDetails allows us to set up all the required information that is used for authorization request and token request.

Note: The AuthorizationCodeResourceDetails is a subclass of OAuth2ProtectedResourceDetails interface. Spring Security OAuth2 module provides the resource details of concrete classes for each grant type described by the OAuth 2.0 protocol.

Step5: Create the controller which will manage all the requests to the respective client application endpoints.

```
package com.packt.example.clientauthorizationcode.user;
@Controller
public class UserDashboard {
    @Autowired
    private OAuth2RestTemplate restTemplate;

    @GetMapping("/")
    public String home() {
        return "index";
    }

    @GetMapping("/callback")
    public ModelAndView callback() {
        return new ModelAndView("forward:/dashboard");
    }

    @GetMapping("/dashboard")
    public ModelAndView dashboard() {
        UserDetails userDetails = (UserDetails)
        SecurityContextHolder.getContext().getAuthentication().getPrincipal();

        ModelAndView mv = new ModelAndView("dashboard");
        mv.addObject("user", userDetails);

        tryToGetUserProfile(mv);

        return mv;
    }

    private void tryToGetUserProfile(ModelAndView mv) {
        String endpoint = "http://localhost:8081/api/profile";
        try {
            UserProfile userProfile = restTemplate.getForObject(endpoint, UserProfile.class);
            mv.addObject("profile", userProfile);
        } catch (HttpClientErrorException e) {
            throw new RuntimeException("it was not possible to retrieve user profile");
        }
    }
}
```



```
}  
}  
}
```

Step6: Start Redis Server, Authorization server, Resource server and finally Client Application.
Go to <http://localhost:9000> → click on “Go to your dashboard” link → Enter end user login credentials (user1@example.com / user1) → Click on “Login” button → Redirected another login page to enter Resource owner credentials (adolfo/123) → Authorize request by selecting “Approve” and click on Authorize button → Redirected to client application (using redirect_uri) → finally renders profile details.

Client Password

Step1: Copy ‘client-authorization-code’ project and rename it as ‘client-password’.

Step2: Modify ClientConfiguration.java file

```
package com.packt.example.clientauthorizationcode.oauth;  
  
@Configuration  
@EnableOAuth2Client  
public class ClientConfiguration {  
    @Autowired  
    private OAuth2ClientContext oauth2ClientContext;  
  
    @Bean  
    public OAuth2ProtectedResourceDetails resourceDetails() {  
        ResourceOwnerPasswordResourceDetails resourceDetails = new ResourceOwnerPasswordResourceDetails();  
  
        //resourceDetails.setId("oauth2server");  
        //resourceDetails.setTokenName("oauth_token");  
        resourceDetails.setClientId("clientapp");  
        resourceDetails.setClientSecret("123456");  
        resourceDetails.setUsername("adolfo");  
        resourceDetails.setPassword("123");  
        //resourceDetails.setPreEstablishedRedirectUri(("http://localhost:9000/callback")); //N.A  
        resourceDetails.setGrantType("password");  
        resourceDetails.setScope(Arrays.asList("read_profile"));  
        // resourceDetails.setUserAuthorizationUri("http://localhost:9090/oauth/authorize"); //N.A  
        resourceDetails.setAccessTokenUri("http://localhost:9090/oauth/token");  
        // resourceDetails.setUseCurrentUri(false); //N.A  
        resourceDetails.setClientAuthenticationScheme(AuthenticationScheme.header);  
  
        return resourceDetails;  
    }  
  
    @Bean
```

```

public OAuth2RestTemplate oauth2RestTemplate() {
    OAuth2RestTemplate template = new OAuth2RestTemplate(resourceDetails(), oauth2ClientContext);
    /*AccessTokenProviderChain provider = new AccessTokenProviderChain(Arrays.asList(new
ResourceOwnerPasswordAccessTokenProvider()));
    template.setAccessTokenProvider(provider);
*/
    return template;
}
}

```

Step3: Make sure add AuthenticationManager in OAuth2AuthorizationServer.java file in 'seperate-authorization-server' project. Also make sure that all grant types are listed in OAuth2AuthorizationServer.java file.

@EnableAuthorizationServer

```

public class OAuth2AuthorizationServer extends AuthorizationServerConfigurerAdapter {
    ...
    @Autowired
    private AuthenticationManager authenticationManager; //needed by password grant

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
        endpoints.tokenStore(tokenStore());
        endpoints.authenticationManager(authenticationManager); //needed by password grant
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
            .withClient("clientapp").secret("123456") //client application credentials
            .redirectUri("http://localhost:9000/callback")
            .authorizedGrantTypes("authorization_code", "implicit", "password",
"client_credentials", "refresh_token")
            .scopes("read_profile");
    }
}
}

```

Step4: Start Redis Server, Authorization server, Resource server and finally Client application.

Go to <http://localhost:9000> → click on "Go to your dashboard" link → Enter end user login credentials (user1@example.com / user1) → Click on "Login" button → client application renders profile details.

Client Client-Credentials

Step1: Copy 'client-authorization-code' project and rename as 'client-client-credentials'.

Step2: Modify ClientConfiguration.java file

```
package com.packt.example.clientauthorizationcode.oauth;
@Configuration
@EnableOAuth2Client
public class ClientConfiguration {
    @Autowired
    private OAuth2ClientContext oauth2ClientContext;

    @Bean
    public OAuth2ProtectedResourceDetails resourceDetails() {
        ClientCredentialsResourceDetails resourceDetails = new ClientCredentialsResourceDetails();

        resourceDetails.setId("oauth2server");
        resourceDetails.setTokenName("oauth_token");
        resourceDetails.setClientId("clientapp");
        resourceDetails.setClientSecret("123456");
        resourceDetails.setGrantType("client_credentials");
        resourceDetails.setScope(Arrays.asList("read_profile"));
        //resourceDetails.setPreEstablishedRedirectUri("http://localhost:9000/callback"); //N.A
        //resourceDetails.setUserAuthorizationUri("http://localhost:9090/oauth/authorize"); //N.A
        resourceDetails.setAccessTokenUri("http://localhost:9090/oauth/token");
        //resourceDetails.setUseCurrentUri(false); //N.A
        resourceDetails.setClientAuthenticationScheme(AuthenticationScheme.header);

        return resourceDetails;
    }

    @Bean
    public OAuth2RestTemplate oauth2RestTemplate() {
        OAuth2RestTemplate template = new OAuth2RestTemplate(resourceDetails(), oauth2ClientContext);
        /*AccessTokenProviderChain provider = new AccessTokenProviderChain(Arrays.asList(new
        ClientCredentialsAccessTokenProvider()));
        template.setAccessTokenProvider(provider);
        */
        return template;
    }
}
```

Step3: Modify UserController.java in 'separate-resource-server'

```
package com.packt.example.authcodeserver.api;
@Controller
public class UserController {
    @RequestMapping("/api/profile")
    public ResponseEntity < UserProfile > profile() { // needed in case of client credentials grant type
```

```
String user = (String) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
String email = user + "@mailinator.com";
```

```
UserProfile profile = new UserProfile();
profile.setName(user);
profile.setEmail(email);
return ResponseEntity.ok(profile);
}
}
```

Step4: Start Redis Server, Authorization server, Resource server and finally Client code.

Go to <http://localhost:9000> → click on “Go to your dashboard” link → Enter end user login credentials (user1@example.com / user1) → Click on “Login” button → client application renders profile details.

Client Implicit

Step1: Copy ‘client-authorization-code’ project and name it as ‘client-implicit’.

Step2: Modify ClientConfiguration.java file

```
package com.packt.example.clientauthorizationcode.oauth;
@Configuration
@EnableOAuth2Client
public class ClientConfiguration {
    @Autowired
    private OAuth2ClientContext oauth2ClientContext;

    @Bean
    public OAuth2ProtectedResourceDetails resourceDetails() {
        ImplicitResourceDetails resourceDetails = new ImplicitResourceDetails();

        resourceDetails.setId("oauth2server");
        resourceDetails.setTokenName("oauth_token");
        resourceDetails.setClientId("clientapp");
        //resourceDetails.setClientSecret("123456");
        resourceDetails.setGrantType("implicit");
        resourceDetails.setUserAuthorizationUri("http://localhost:9090/oauth/authorize");
        //resourceDetails.setAccessTokenUri("http://localhost:9090/oauth/token");
        resourceDetails.setScope(Arrays.asList("read_profile"));
        resourceDetails.setPreEstablishedRedirectUri(("http://localhost:9000/callback"));
        resourceDetails.setUseCurrentUri(false);
        resourceDetails.setClientAuthenticationScheme(AuthenticationScheme.header);

        return resourceDetails;
    }
}
```

```

@Bean
public OAuth2RestTemplate oauth2RestTemplate() {
    OAuth2RestTemplate template = new OAuth2RestTemplate(resourceDetails(), oauth2ClientContext);
    AccessTokenProviderChain provider = new AccessTokenProviderChain(Arrays.asList(new
CustomImplicitAccessTokenProvider()));
    template.setAccessTokenProvider(provider);
    return template;
}

package com.packt.example.clientauthorizationcode.oauth;
public class CustomImplicitAccessTokenProvider implements AccessTokenProvider {
    @Override
    public OAuth2AccessToken obtainAccessToken(OAuth2ProtectedResourceDetails details,
    AccessTokenRequest request) throws RuntimeException {
        ImplicitResourceDetails resource = (ImplicitResourceDetails) details;
        Map<String, String> requestParameters = getParametersForTokenRequest(
            resource, request);
        UserRedirectRequiredException redirectException = new UserRedirectRequiredException(
            resource.getUserAuthorizationUri(), requestParameters);
        throw redirectException;
    }

    @Override
    public boolean supportsResource(OAuth2ProtectedResourceDetails resource) {
        return resource instanceof ImplicitResourceDetails
            && "implicit".equals(resource.getGrantType());
    }

    @Override
    public OAuth2AccessToken refreshAccessToken(
        OAuth2ProtectedResourceDetails resource,
        OAuth2RefreshToken refreshToken, AccessTokenRequest request)
        throws UserRedirectRequiredException {
        return null;
    }

    @Override
    public boolean supportsRefresh(OAuth2ProtectedResourceDetails resource) {
        return false;
    }

    private Map<String, String> getParametersForTokenRequest(

```

```

        ImplicitResourceDetails resource, AccessTokenRequest request) {

    Map<String, String> queryString = new HashMap<String, String>();
    queryString.put("response_type", "token");
    queryString.put("client_id", resource.getClientId());

    if (resource.isScoped()) {
        queryString.put("scope",
            resource.getScope().stream().reduce((a, b) -> a + " " + b)
                .get());
    }

    String redirectUri = resource.getRedirectUri(request);
    if (redirectUri == null) {
        throw new IllegalStateException(
            "No redirect URI available in request");
    }
    queryString.put("redirect_uri", redirectUri);

    return queryString;
}
}

```

Step3: Modify UserDashborad.java file

@Controller

```

public class UserDashboard {
    @Autowired
    private OAuth2RestTemplate restTemplate;

    @GetMapping("/")
    public String home() {
        return "index";
    }

    @GetMapping("/callback")
    public ModelAndView callback() {
        // return new ModelAndView("forward:/dashboard");
        ModelAndView mv = new ModelAndView("dashboard");
        return mv;
    }

    @GetMapping("/dashboard")
    public ModelAndView dashboard() {

```

```

        UserDetails userDetails = (UserDetails)
SecurityContextHolder.getContext().getAuthentication().getPrincipal();

        ModelAndView mv = new ModelAndView("dashboard");
        mv.addObject("user", userDetails);
restTemplate.getAccessToken(); // needed for 'implicit' grant
        // tryToGetUserProfile(mv);

        return mv;
    }

    /*private void tryToGetUserProfile(ModelAndView mv) {
        String endpoint = "http://localhost:8081/api/profile";
        try {
            UserProfile userProfile = restTemplate.getForObject(endpoint,
UserProfile.class);
            mv.addObject("profile", userProfile);
        } catch (HttpClientErrorException e) {
            throw new RuntimeException("it was not possible to retrieve user profile");
        }
    }
    */
}

```

Step4: Add profile.js and jquery.js files in static folder under src/main/resources

Step5: Add .cors() in OAuth2ResourceServer.java file in 'separate-resource-server' project. Also add @CrossOrigin in UserController.java file

```

@Override
public void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated().and()
        .requestMatchers().antMatchers("/api/**").and().
cors(); //needed in 'implicit' grant type
}

@Controller
public class UserController {
    @CrossOrigin
    @RequestMapping("/api/profile")
    public ResponseEntity<UserProfile> profile() {
    }
}

```

Step6: Start Redis server, Authorization Server, Resource Server and Client code.

3. MICROSERVICES SECURITY WITH OAUTH

Microservices architecture says decompose single project into many smaller services. But we cannot ask the end user to sign-up and sign-in to every microservice. Instead the end user prefers single sign-on irrespective of number of microservices.

In case of single sign-on, we need a common id to represent user's identity.

Different microservices will run on different embedded servers. Hence there is no common session id. That means we cannot use session id across different microservices to represent user's identity. Instead of using session id, rather use **oauth token** which is common across all microservices to represent user's identity.

The oauth token was generated by **Authorization server** and validated by **Resource server**.

In microservice projects there are multiple microservices. Each microservice contains a few resources. These resources need to be protected by Resource server. Hence every microservice needs resource server to protect resources by validating the token i.e., multiple resource servers are needed. But only one authorization server is sufficient to issue a common token for all microservices. Hence implement one authorization server independently and include resource server in each microservice. **So, microservice projects need single authorization server and multiple resource servers.** Hence in microservice projects, the authorization and resource servers are separated.

Microservice projects need single authorization server to generate a common access token irrespective of number of microservices and number of end users logged in to client application.

Microservice projects need multiple resource servers to validate token because resources are part of every microservice.

Add Spring Security in Website to protect it from end users with the help of end user authentication.

Fares MicroService

Step1: Import all microservices from spring microservices workspace.

Also import 'separate-authorization-server' project from OAuthWorkspace.

Step2: Add oauth2 and redis dependencies across all microservices' pom.xml files.

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

Step3: Add redis server url in application.properties file in all microservices
spring.redis.url=redis://localhost:6379

Step4: Write resource server configuration file


```

package com.brownfield.pss.resourceserver.config;
@Configuration
@EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdapter {
    @Autowired
    private RedisConnectionFactory connectionFactory;

    @Bean
    public TokenStore tokenStore() {
        RedisTokenStore redis = new RedisTokenStore(connectionFactory);
        return redis;
    }

    @Override
    public void configure(ResourceServerSecurityConfigurer endpoints) throws Exception {
        endpoints.tokenStore(tokenStore());
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().authenticated().and()
            .requestMatchers().antMatchers("/fares/**").and().
            cors(); //needed by 'implicit' grant type
    }
}

```

Add @SpringBootApplication(scanBasePackages= {"com.brownfield.pss.resourceserver.config", "com.brownfield.pss.fares.controller", "com.brownfield.pss.fares.component"}) in bootstrap class.

Step5: Start Redis Server, Authorization server and Fare microservice and run below url from postman.
<http://localhost:8081/fares/get?flightNumber=BF101&flightDate=22-JAN-16>

Try above url from postman without configuring authorization

```

{
  "error": "Unauthorized",
  "message": "Full authentication is required to access this resource",
}

```

Configure Authorization in postman.

Select TYPE as 'OAuth2.0' → Click on 'Get New Access Token' →

GET NEW ACCESS TOKEN

Token Name

AspireOAuthToken

Grant Type

Password Credentials

Access Token URL ⓘ

http://localhost:9090/oauth/token

Username

adolfo

Password

123

☒ Show Password

Client ID ⓘ

clientapp

Client Secret ⓘ

123456

Scope ⓘ

read_profile

Client Authentication

Send as Basic Auth header

Request Token

Click on 'Request Token' button.

Token Name

AspireOAuthToken

Access Token

54717c38-8e33-4ee3-8487-20edf00a19ee

Token Type

bearer

expires_in

6861

scope

read_profile

Use Token

Optionally click on "Preview Request" to update authorization header value.
Click on "Send" button.

```

{
  "id": 2,
  "flightNumber": "BF101",
  "flightDate": "22-JAN-16",
  "fare": "101"
}

```

Search Microservice

Step2 and Step3 are same as above.

Step4: Modify ant matchers in resource server configuration file (OAuth2ResourceServer.java).

```
@Override
public void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated().and()
        .requestMatchers().antMatchers("/search/**").and().
        cors(); //need by 'implicit' grant type
}
```

@SpringBootApplication(scanBasePackages= {"com.brownfield.pss.resourceserver.config", "com.brownfield.pss.search.controller", "com.brownfield.pss.search.component"})

Step5: Start search microservice and run below url from postman.

Set http method as "POST"

<http://localhost:8090/search/get>

Goto 'Body', Select 'raw' and set JSON (application/json)

```
{
    "origin": "SEA",
    "destination": "SFO",
    "flightDate": "22-JAN-16"
}
```

Try above url without bearer token. We will get below error.

```
{
    "error": "unauthorized",
    "error_description": "Full authentication is required to access this resource"
}
```

Configure token in 'Authorization' tab in postman

Select TYPE as 'OAuth2.0' → Click on 'Get New Access Token' → ...

Click on 'Use Token', 'Preview Request' and 'Send' buttons.

```
[{"id":1,"flightNumber":"BF100","origin":"SEA","destination":"SFO","flightDate":"22-JAN-16","fares":{"id":1,"fare":"100","currency":"USD"},"inventory":{"id":1,"count":100}}]
```

Booking Micorservice

Step2 and Step3 are same as above.

Step4: Modify ant matchers in resource server configuration file (OAuth2ResourceServer.java).

```
@Override
```

```

        public void configure(HttpSecurity http) throws Exception {
            http.authorizeRequests().anyRequest().authenticated().and()
                .requestMatchers().antMatchers("/booking/**").and().
                cors();
        }
    }

    @SpringBootApplication(scanBasePackages= {"com.brownfield.pss.resourceserver.config",
        "com.brownfield.pss.book.controller", "com.brownfield.pss.book.component"})

```

Step5: Configure OAuth2RestTemplate in BookingComponent.java file.

@Configuration

class AppConfig {

@Bean

public **OAuth2RestTemplate** restTemplate() {

ResourceOwnerPasswordResourceDetails resource = new ResourceOwnerPasswordResourceDetails();

resource.setClientId("clientapp");

resource.setClientSecret("123456");

resource.setUsername("adolfo");

resource.setPassword("123");

resource.setGrantType("password");

resource.setScope(Arrays.asList(new String[] { "read_profile"}));

resource.setAccessTokenUri("http://localhost:9090/oauth/token");

DefaultOAuth2ClientContext clientContext = new DefaultOAuth2ClientContext();

OAuth2RestTemplate restTemplate = new OAuth2RestTemplate(resource,

clientContext);

System.out.println("Access Token: " + restTemplate.getAccessToken());

return restTemplate;

}

}

@Service

public class BookingComponent {

@Autowired

private **OAuth2RestTemplate** restTemplate;

...

}

Step6: Start booking microservice and run below url from postman.

Set http method as "GET"

<http://localhost:8060/booking/get/1>

{

"error": "unauthorized",

"error_description": "Full authentication is required to access this resource"

}

Configure token in 'Authorization' tab in postman.

Select TYPE as 'OAuth2.0' → Click on 'Get New Access Token' →

Click on 'Use Token', 'Preview Request' and 'Send' buttons.

```
{ "id":1,"flightNumber":"BF101","origin":"NYC","destination":"SFO","flightDate":"22-JAN-16","bookingDate":1552143130517,"fare":"101","status":"BOOKING_CONFIRMED","passengers":[{"id":1,"firstName":"Gean","lastName":"Franc","gender":"Male"}]}
```

CheckIn Microservice

Step2 and Step3 are same as above.

Step4: Modify ant matchers in resource server configuration file (OAuth2ResourceServer.java).

```
@Override
public void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated().and()
        .requestMatchers().antMatchers("/checkin/**").and().
        cors();
}
```

@SpringBootApplication(scanBasePackages= {"com.brownfield.pss.resourceserver.config", "com.brownfield.pss.checkin.controller", "com.brownfield.pss.checkin.component"})

Step5: Configure OAuth2RestTemplate in CheckInComponent.java file

@Configuration

class AppConfig {

@Bean

```
public OAuth2RestTemplate restTemplate() {
    ResourceOwnerPasswordResourceDetails resource = new ResourceOwnerPasswordResourceDetails();
    resource.setClientId("clientapp");
    resource.setClientSecret("123456");
    resource.setUsername("adolfo");
    resource.setPassword("123");
    resource.setGrantType("password");
    resource.setScope(Arrays.asList(new String[] { "read_profile"}));
    resource.setAccessTokenUri("http://localhost:9090/oauth/token");
    DefaultOAuth2ClientContext clientContext = new DefaultOAuth2ClientContext();
    OAuth2RestTemplate restTemplate = new OAuth2RestTemplate(resource,
clientContext);
    System.out.println("Access Token: " + restTemplate.getAccessToken());
    return restTemplate;
}
```

```
}
```

@Service

public class CheckInComponent {

@Autowired

```

        private OAuth2RestTemplate restTemplate;
        ...
    }

```

Step6: Start checkin microservice and run below url from postman.

Set http method as "GET"

<http://localhost:8070/checkin/get/1>

```

{
  "error": "unauthorized",
  "error_description": "Full authentication is required to access this resource"
}

```

Configure token in 'Authorization' tab in postman

Select TYPE as 'OAuth2.0' → Click on 'Get New Access Token' →

Click on 'Use Token', 'Preview Request' and 'Send' buttons.

```

{"id":1,"lastName":"Franc","firstName":"Gean","seatNumber":"28A","checkInTime":1552143890540,"flightNumber":"BF101","flightDate":"22-JAN-16","bookingId":1}

```

PSS Website

Step2: Add oauth2 dependency in pss-website's pom.xml file. The Redis dependency is not required.

```

<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>

```

Step3: Remove end user credentials from application.properties file because these will be added to spring security config file in next step.

#security.user.name=guest

#security.user.password=guest123

Step4: Write new spring security configuration file and configure a few end user credentials.

package com.brownfield.pss.springsecurity.configuration;

@EnableWebSecurity

public class **SecurityConfig** extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.authorizeRequests()

.anyRequest().authenticated().and()

.formLogin();

}

@Override

```

public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("user1@example.com").password("user1").roles("USER").and()
        .withUser("user2@example.com").password("user2").roles("USER");
}
}
@SpringBootApplication(scanBasePackages= {"com.brownfield.pss.client",
"com.brownfield.pss.springsecurity.configuration"})

```

Step5: Configure OAuth2RestTemplate in BrownFieldSiteController.java file

```

@Configuration
class AppConfig {
    @Bean
    public OAuth2RestTemplate restTemplate() {
        ResourceOwnerPasswordResourceDetails resource = new ResourceOwnerPasswordResourceDetails();
        resource.setClientId("clientapp");
        resource.setClientSecret("123456");
        resource.setUsername("adolfo");
        resource.setPassword("123");
        resource.setGrantType("password");
        resource.setScope(Arrays.asList(new String[] { "read_profile"}));
        resource.setAccessTokenUri("http://localhost:9090/oauth/token");
        DefaultOAuth2ClientContext clientContext = new DefaultOAuth2ClientContext();
        OAuth2RestTemplate restTemplate = new OAuth2RestTemplate(resource,
clientContext);

        System.out.println("Access Token: " + restTemplate.getAccessToken());
        return restTemplate;
    }
}

@Service
public class BrownFieldSiteController{
    @Autowired
    OAuth2RestTemplate searchClient;
    @Autowired
    OAuth2RestTemplate bookingClient;
    @Autowired
    OAuth2RestTemplate checkInClient;
    ...
}

```

Step6: Also use OAuth2RestTemplate in Application.java file

```

public class Application implements CommandLineRunner {
    private static final Logger logger = LoggerFactory.getLogger(Application.class);
}

```

```

    @Autowired
    OAuth2RestTemplate searchClient;
    @Autowired
    OAuth2RestTemplate bookingClient;
    @Autowired
    OAuth2RestTemplate checkInClient;
    ...
}

```

Step7: Start PSS web site and test through web browser

Microservice projects need common access token irrespective of number of microservices and number of end users logged in to web application. That means end user's logout by spring security invalidates end user session but doesn't expire access token by microservice security. Similarly expiring access token by microservice security doesn't force end user's logout by spring security.

Step1: Design login page in PSS web site and place it in src/main/resources/templates folder.

aspirelogin.html

```

<!DOCTYPE html>
<html
    xmlns:th="http://www.thymeleaf.org"
    xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4">
    <body>
        <div class="container">
            <form th:action="@{/processlogin}" method="POST" cssClass="form-
horizontal">
                <div th:if="${param.error != null}" class="alert alert-danger">
                    <strong>Failed to login.</strong>
                    <span th:if="${session[SPRING_SECURITY_LAST_EXCEPTION] !=
null}">
                        <span
th:text="${session[SPRING_SECURITY_LAST_EXCEPTION].message}">Invalid credentials</span>
                    </span>
                </div>
                <div th:if="${param.logout != null}" class="alert alert-success">
                    You have been logged out.
                </div>
                <label for="username">Username</label>
                <input type="text" id="username" name="username"
autofocus="autofocus"/>
                <label for="password">Password</label>
                <input type="password" id="password" name="password"/>
                <div class="form-actions">
                    <input id="submit" class="btn" name="submit" type="submit"
value="Login"/>

```



```

        </div>
    </form>
</div>
</body>
</html>

```

Step2: Modify search.html page to add Logout link.

```

<nav>
    <ul>
        <li>BrownField Airline </li>
        <li><a th:href="@{/}">Search</a></li>
        <li><a th:href="@{/search-booking}">CheckIn</a></li>
        <li><a th:href="@{/logout}">Logout</a></li>
    </ul>
</nav>
<div align="center" class="form-style-8">
    <h2>Flight Search</h2>
    <form action="#" th:action="@{/search}" th:object="${uidata.searchQuery}" method="post">
        <table>
            ...

```

Step3: Configure login page details in SecurityConfig.java file

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/login/**").permitAll()
        .anyRequest().authenticated()
        .and().formLogin()
            .loginPage("/login/form")
            .loginProcessingUrl("/processlogin")
            .usernameParameter("username")
            .passwordParameter("password")
            .failureUrl("/login/form?error")
        .and().logout()
            .logoutUrl("/logout")
            .logoutSuccessUrl("/login/form?logout")
        .and().csrf().disable();
}

```

```

@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("user1@example.com").password("user1").roles("USER").and()
        .withUser("user2@example.com").password("user2").roles("USER");
}

```

```
}
```

Step4: Configure path in ThymeleafConfig.java file

```
package com.brownfield.pss.springsecurity.configuration;
```

```
public class ThymeleafConfig extends WebMvcConfigurerAdapter {
```

```
...
```

```
@Override
```

```
public void addViewControllers(final ViewControllerRegistry registry) {
```

```
    super.addViewControllers(registry);
```

```
    registry.addViewController("/login/form").setViewName("aspirelogin");
```

```
    //registry.addViewController("/default").setViewName("/");
```

```
    registry.setOrder(Ordered.HIGHEST_PRECEDENCE);
```

```
}
```

```
}
```

Step5: Add below dependency in pom.xml file of PSS Website project.

```
<dependency>
```

```
    <groupId>org.thymeleaf.extras</groupId>
```

```
    <artifactId>thymeleaf-extras-springsecurity4</artifactId>
```

```
</dependency>
```

Also add below dependency to avoid java.net.HttpRetryException: cannot retry due to server authentication, in streaming mode

```
<dependency>
```

```
    <groupId>org.apache.httpcomponents</groupId>
```

```
    <artifactId>httpclient</artifactId>
```

```
</dependency>
```

Step6: Add access token and refresh token validity in Authorization Server configuration file (OAuth2AuthorizationServer.java) file in 'separate-authorization-server' project.

```
@Override
```

```
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
```

```
    clients.inMemory()
```

```
        .withClient("clientapp").secret("123456") //client application credentials
```

```
        .redirectUri("http://localhost:9000/callback") //optional
```

```
        .authorizedGrantTypes("password","refresh_token")
```

```
        .accessTokenValiditySeconds(120)
```

```
        .refreshTokenValiditySeconds(5000)
```

```
        .scopes("read_profile");
```

```
}
```

Step7: Restart both authorization server and web site projects.

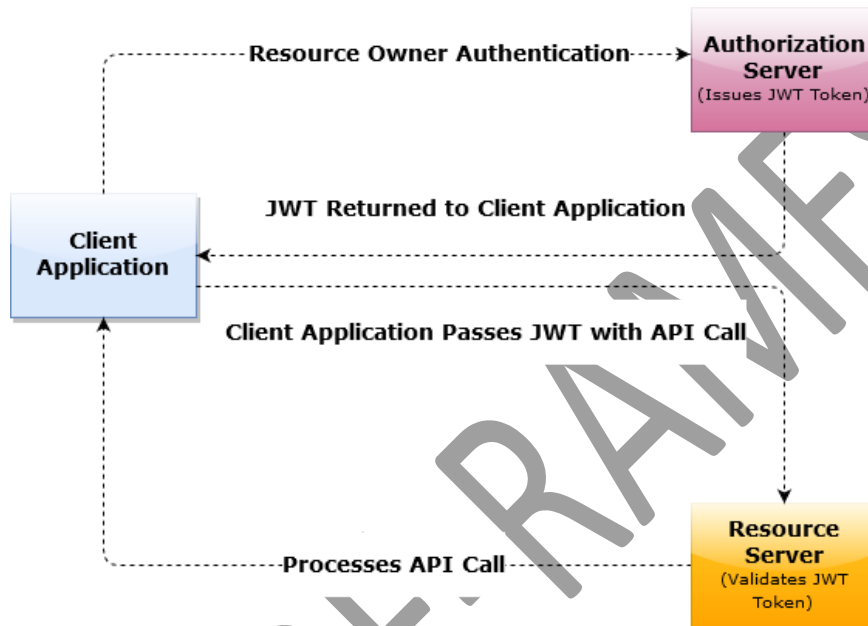
4. JSON WEB TOKEN (JWT)

Introduction

JWT is important in ensuring trust and security in our application.

A JSON Web Token (JWT) is a JSON object that is defined in RFC 7519 as a safe way to represent a set of information between two parties. The **token is composed of a header, a payload, and a signature**.

A JWT is just a string with the format: **header.payload.signature**



First resource owner has to authenticate and authorize client application by using resource owner's credentials (adolfo/123).

The authorization server **issues** the JWT and sends it back to the client application.

When the client application makes API calls to the protected resources in resource server, the client application passes the JWT along with the API call.

The resource server **validates** incoming JWT which was generated by authorization server and then processes resources.

The token is composed of **a header, a payload, and a signature**.

Header

The header component of the JWT contains information about how the JWT signature should be computed. The header is a JSON object in the following format:

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

The "typ" key specifies that the object is a JWT.

The “alg” key specifies which hashing algorithm is being used to create the JWT signature component. We’re using the HMAC-SHA256 algorithm, a hashing algorithm that uses a **hash secret key**, to compute the signature.

Payload

The payload component of the JWT is the data that is stored inside the JWT (this data is also referred to as the “claims” of the JWT).

In our example, the authorization server issues a JWT with the user information stored inside of it, specifically the user ID.

```
{  
  "userId": "b08f86af-35da-48f2-8fab-cef3904660bd"  
}
```

In our example, we are only putting one claim into the payload. We can put as many claims as we like. There are several different standard claims for the JWT payload, such as “iss” the issuer, “sub” the subject, and “exp” the expiration time. These fields can be useful when creating JWT, but they are optional.

Signature

The signature is computed using the following pseudo code:

```
// signature algorithm  
data = base64urlEncode(header) + "." + base64urlEncode(payload)  
hashedData = hash( data, hash secret key)  
signature = base64urlEncode(hashedData)
```

What this algorithm does is base64url encodes the header and the payload created. The algorithm then joins the resulting encoded strings together with a period (.) in between them. In our pseudo code, this joined string is assigned to data. The data string is hashed with the **hash secret key** using the hashing algorithm specified in the JWT header. The resulting hashed data is assigned to hashedData. This hashed data is then base64url encoded to produce the JWT signature.

In our example, both the header, and the payload are base64url encoded as:

```
// header  
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9  
// payload  
eyJ1c2VySWQiOiJiMDhmODZlZi0zNWRhLTQ4ZjltOGZhYi1jZWYzOTA0NjYwYmQifQ
```

Then, applying the specified signature algorithm with the hash secret key on the period-joined encoded header and encoded payload, we get the hashed data needed for the signature. In our case, this means applying the HS256 algorithm, with the hash secret key set as the string “secret”, on the data string to get the hashedData string. After, through base64url encoding the hashedData string we get the following JWT signature:

```
// signature  
-xN_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM
```

Below is the JWT token after putting all three JWT components together:

```
// JWT Token  
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjltOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM
```

The authorization server now sends this JWT to the client application.

Note: We can decode JWT through web browser at jwt.io.

It is important to understand that the purpose of using JWT is **NOT** to hide or obscure or protect data in any way. As demonstrated in the previous steps, the data inside JWT is encoded and hashed but not encrypted.

The purpose of **Encoding** is to transform data so that it can be properly consumed by a different type of system, e.g. binary data is being sent over email, or viewing special characters on a web page. The **Encoding** transforms data into another format using an algorithm (such as ASCII, Unicode [UTF-8, UTF-16], URL Encoding, base64) which is publicly available so that it can be easily reversed (decoded) using same algorithm that encoded the content i.e., no key is used. The goal of encoding is not to keep information secret, rather to ensure that it's able to be properly consumed.

Hashing is meant for storing sensitive data such as password in database server or ldap server or configuration file safely. Different hashing algorithms are SHA, MD5, HMAC, etc.

Note: Encoding and Hashing do not protect data between web browser and web server.

Encryption is used to protect data between web browser and web server by using digital certificates. Different encryption algorithms are DES, AES, BlowFish, RSA, etc.

Since JWT is encoded and hashed but not encrypted, hence JWT does not guarantee any security for sensitive data.

Hence it should be noted that JWT should be sent over HTTPS connections. Having HTTPS helps prevents unauthorized users from stealing the sent JWT by making it so that the communication between the servers and the client application cannot be intercepted.

Verifying JWT

The hash secret key is configured in resource server.

Since the resource server knows the hash secret key, when the client application makes a JWT-attached API call to the resource server, the resource server can perform the same signature algorithm as above. The resource server can then verify that the signature obtained from its own hashing operation matches the signature on the JWT itself (i.e. it matches the JWT signature created by the authorization server). If the signatures match, then that means the JWT is valid which indicates that the API call is coming from an authentic source.

Otherwise, if the signatures don't match, then it means that the received JWT is invalid, which may be an indicator of a potential attack on the application. So, by verifying the JWT, the resource server adds a layer of trust between itself and the client application.

Microservices with JWT

JWT token with OAuth protocol i.e., using JWT token in place of auth token.

This chapter introduces how JSON Web Tokens (JWT) can be safely used to carry client information to allow Resource Servers to locally validate JWT tokens.

It's important to bear in mind that even though we are signing or hashing the JWT signature, all the connections must be performed using TLS/SSL in production.

Authorization Server with Symmetric Key

Step1: Import all projects from MicroservicesOAuthWorkspace and rename 'separate-authorization-server' as 'authorization-server-jwt'.

Step2: Add spring-security-jwt dependency in 'authorization-server-jwt' project.

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
```

Also remove redis dependency from pom.xml file.

Because of the structure of a JWT, it is possible for the Resource Server to validate the jwt token locally, instead of having to use token store.

```
<!-- <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency> -->
```

Step3: Remove redis property from application.properties file.

```
#spring.redis.url=redis://localhost:6379
```

Step4: Configure JwtAccessTokenConverter and JwtTokenStore beans in OAuth2AuthorizationServer.java file to generate JWT token instead of auth token.

@EnableAuthorizationServer

public class OAuth2AuthorizationServer extends AuthorizationServerConfigurerAdapter {

 @Bean

 public JwtAccessTokenConverter accessTokenConverter() {

 JwtAccessTokenConverter converter = new JwtAccessTokenConverter();

 converter.setSigningKey("aspire");

 return converter;

 }

 @Bean

 public JwtTokenStore jwtTokenStore() {

 return new JwtTokenStore(accessTokenConverter());

 }

 @Override

 public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {

 //endpoints.tokenStore(tokenStore());

 endpoints.tokenStore(jwtTokenStore());



1

1

1

1

```
}
```

Payload:

```
{
  "exp": 1552198618,
  "user_name": "adolfo",
  "authorities": [
    "ROLE_USER"
  ],
  "jti": "f8a5adee-f326-4c1a-ae11-332048584aa4",
  "client_id": "clientapp",
  "scope": [
    "read_profile"
  ]
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  aspire
)
```

Signature Verified

MicroServices with Symmetric Key

We were symmetrically signing the JWT token. That is, we were using the same key to sign the payload at the Authorization Server and to validate it on the Resource Server.

The resource server in fare microservice should validate JWT token which was generated by authorization server.

Step1: Add spring-security-jwt dependency and remove redis dependency.

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
```

Also remove redis dependency from pom.xml file.

```
<!-- <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency> -->
```

Step2: Add JWT hash secret key in application.properties file. Also remove redis url from application.properties file.

#spring.redis.url=redis://localhost:6379

security.oauth2.resource.jwt.key-value=aspire

Note: The Resource Server has to use the same hash secret key which was used by the Authorization Server to compute signature.

Step3: Remove redis configurations from OAuth2ResourceServer.java file.

@EnableResourceServer

```
public class OAuth2ResourceServer extends ResourceServerConfigurerAdapter {
    /*@Autowired
    private RedisConnectionFactory connectionFactory;

    @Bean
    public TokenStore tokenStore() {
        RedisTokenStore redis = new RedisTokenStore(connectionFactory);
        return redis;
    }

    @Override
    public void configure(ResourceServerSecurityConfigurer endpoints) throws Exception {
        endpoints.tokenStore(tokenStore());
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().authenticated().and()
            .requestMatchers().antMatchers("/fares/**").and().
            cors(); //need by 'implicit' grant type
    }
}
```

To start validating JWT tokens when using Spring Security OAuth2 with Spring Boot, everything will be automatically configured just by adding the security.oauth2.resource.jwt.key-value property in the application.properties file.

Step4: Start FaresFlightTickets project.

Try below url from postman plug-in and send 'GET' request:

<http://localhost:8081/fares/get?flightNumber=BF101&flightDate=22-JAN-16>

Goto "Authorization" tab → click on "Get New Access Token" button → provide info and click on "Request Token" button.

Now click on "Use Token" button.

Click on "Preview Request" button and then click on "Send" button.

```
{
    "id": 2,
    "flightNumber": "BF101",
```

```

    "flightDate": "22-JAN-16",
    "fare": "101"
}

```

Repeat same steps in Search, Booking and CheckIn microservices. No changes required for PSS Website.

Authorization Server with Asymmetric Key

This approach for signing JWT using asymmetric keys, where **the Authorization Server uses a private key to sign the signature in JWT token and the Resource Server uses a public key to validate it.**

Step1: Import all projects from 'MicroservicesOAuthJWTWorkspace'

Step2: Modify JwtAccessTokenConverter in OAuth2AuthorizationServer.java file to generate keypair that holds both the private and public keys that have to be used to sign and to validate an JWT token, respectively.

We are using the RSA algorithm which relies on private and public keys to perform cryptographic functions.

SHA1PRNG (Secure Hashing Algorithm Pseudo Random Number Generator) algorithm is used to generate random key.

```

import java.security.SecureRandom;
@EnableAuthorizationServer
public class OAuth2AuthorizationServer extends AuthorizationServerConfigurerAdapter {
    //The below method generates public and private key pair.
    @Bean
    public JwtAccessTokenConverter accessTokenConverter() {
        JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
        try {
            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
            SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
            keyGen.initialize(1024, random);
            KeyPair keyPair = keyGen.generateKeyPair();
            converter.setKeyPair(keyPair);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return converter;
    }
    /**
     * Enables the usage of /oauth/token_key to retrieve the JWT signature (public
     * key).
     */
    @Override
    public void configure(AuthorizationServerSecurityConfigurer security) throws Exception {
        security.tokenKeyAccess("permitAll()");
    }
}

```

...
}

Step2: Start authorization server and test in postman

Goto 'Authorization' tab in postman -> Select Type as 'OAuth2.0' → Click on "Get Access Token" → Fill details and Click on "Request Token"

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1NTQ3ODYyNTgsInVzZXJfbmFtZSI6ImFkb2xmb3YsImF1dG8vcml0aWVvZljbllJPTeVfVFNfUjldLCJqdGkiOiJlZDM0NDY1MTk1MWRhLTRjZDQyYjBiMi0xYjQxMzU5ZjE0MGYiLCJjbGllbnRfaWQiOiJjbGllbnRhcHAiLCJyZ29wZSI6WyJyZWZkX3Byb2ZpbGUuXX0.djTyBeDxwZZXcEj59AJOCBQZ6HsyM0n-YbVwA-4B9XHE2nSLp8dKJgk65TPvNBBy6N5r4q_SywlGmjWAKePYIWv2Vs0pX6_XdLVdi0m1rFmOvoRYVzXtzZ3qLjg5_ymkIF-qip3A6w-CDArUQtvfCpNDePvm7jsD2WPx8V2G0zl
```

Send GET request in postman using http://localhost:9090/oauth/token_key → Copy value
"-----BEGIN PUBLIC KEY-----"

```
\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCLc+HTI5oIKMINMKtd/G86XtrjRDVV8oef7W7zeR\nNZi0SMMW4Q+KaBaSeLM4Eyg4pAGC2v47VzNppGAU4VmLrfBRADIB/36BSfw13zGb/uXu8Dqazqs5n7DXI\nxzxuOA5GVQweZHRltmw3dhpuRskJW18uYWbVZ9b2MUqHpVYCDjwIDAQAB\n-----END PUBLIC KEY-----"
```

Format above public key by removing double quotes and wrap in multiple lines as below:

-----BEGIN PUBLIC KEY-----

```
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCLc+HTI5oIKMINMKtd/G86XtrjRDVV8oef7W7zeR\nNZi0SMMW4Q+KaBaSeLM4Eyg4pAGC2v47VzNppGAU4VmLrfBRADIB/36BSfw13zGb/uXu8Dqazqs5n7DXI\nXlxzxuOA5GVQweZHRltmw3dhpuRskJW18uYWbVZ9b2MUqHpVYCDjwIDAQAB
```

-----END PUBLIC KEY-----

Verify the signature by using jwt.io website:

MicroServices with Asymmetric Key

Now it's time to know how to validate JWT token asymmetrically at the Resource Server side. That means instead of statically setting up the hash secret key to validate the JWT token, the Resource Server will retrieve the public key through the `/oauth/token_key` endpoint provided by the Authorization Server.

Step1: Configure `/oauth/token_key` endpoint in `application.properties` file to get public key from authorization server. Also remove hash secret key defined in `application.properties` file.

#security.oauth2.resource.jwt.key-value=aspire #remove

security.oauth2.resource.jwt.key-uri=http://localhost:9090/oauth/token_key

Step2: Start fare microservice.

Try below url from postman plug-in and send 'GET' request:

<http://localhost:8081/fares/get?flightNumber=BF101&flightDate=22-JAN-16>

Goto "Authorization" → "Get New Access Token" → provide info and click on "Request Token" button.

Now click on "Use Token" button.

Click on “Preview Request” button and then click on “Send” button.

```
{  
  "id": 2,  
  "flightNumber": "BF101",  
  "flightDate": "22-JAN-16",  
  "fare": "101"  
}
```

Repeat same steps for Search, Booking and CheckIn microservices. No changes required for PSS Website.

ASPIRE-RAMESH

5. API GATEWAY USING ZUUL

In most microservice implementations, internal microservice endpoints are not exposed outside. They are kept as private services.

Zuul is a simple gateway service.

The Zuul proxy is also capable of routing, monitoring, managing resiliency, security, and so on.

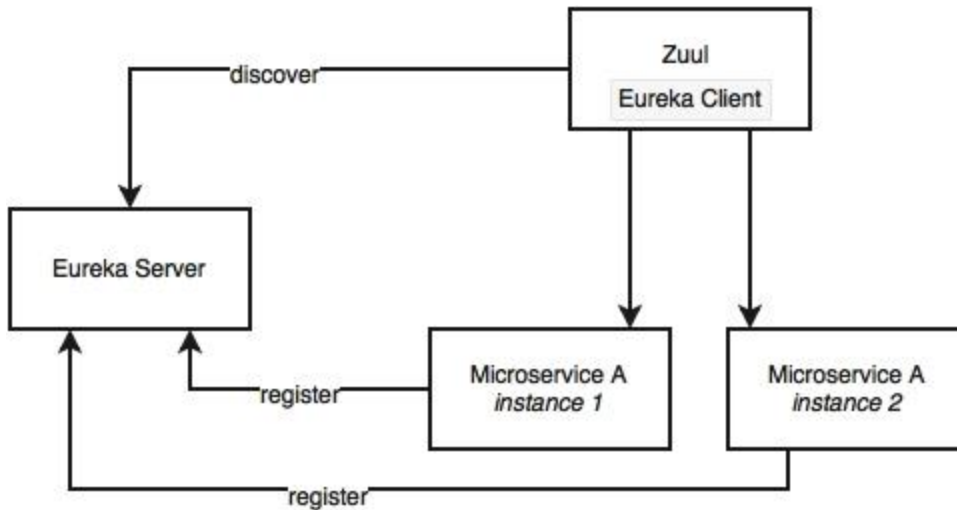
In microservices architecture, there could be a number of API services and few UI components that are talking to APIs. As of now, many microservices based applications still use **monolithic front-ends** where the entire UI is built as a single module. We may choose to go with **micro-frontends** where the UI is also decomposed into multiple microservices talking to APIs to get the relevant data. **Instead of letting the UI to know about all our microservices details, we can provide a unified proxy interface that will delegate the calls to various microservices based on a URL pattern.** In this chapter, we will learn how to create an API Gateway using Spring Cloud Zuul Proxy.

Basically, the API Gateway is a reverse proxy to microservices and acts as a **single-entry point** into the system.

An API Gateway, also known as Edge Service, provides a unified interface for a set of microservices so that clients no need to know about all the details of microservices internals.

There are many reasons to implement API Gateways:

- 1) It is hard to implement **client-specific transformations** at the service endpoint i.e., there are mobile apps where one understands XML payload and others understand JSON.
- 2) If there is data aggregation required, especially to avoid multiple client calls in a bandwidth-restricted environment, then a gateway is required in the middle.
- 3) If there are **client-specific policies** to be applied, it is easy to apply them in a single place rather than in multiple places.
- 4) Can be used to prevent exposing the internal microservices structure to clients.
- 5) Can centralize cross-cutting concerns like security, monitoring, rate limiting, etc.
- 6) Enforcing authentication and other security policies at the gateway instead of doing that on every microservice endpoint. The gateway can handle security policies, token handling, and so on before passing the request to the relevant services behind.
- 7) Business insights and monitoring can be implemented at the gateway level. Collect real-time statistical data, and push it to an external system for analysis. This will be handy as we can do this at one place rather than applying it across many microservices.



The Zuul proxy internally uses the Eureka server for service discovery, and Ribbon for load balancing between service instances.

We can use either a separate gateway for each microservice or common gateway for all microservices. We are opting common gateway.

Step1: Import all projects except spring-boot-admin from MicroServicesCloudEurekaWorkspace

Step2: Create a new Spring Starter project named '**common-apigateway**' and select Zuul, Config Client, Actuator, and Eureka Client.

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

```

Step3: Expand src/main/resources and rename application.properties file as bootstrap.properties. Add below properties

```

spring.application.name=common-apigateway
server.port=8098
spring.cloud.config.uri=http://localhost:8888

```

Step4: Create **common-apigateway.properties** in Git repo and add below properties
spring.application.name=common-apigateway

```
zuul.routes.search-api.serviceId=search-service
zuul.routes.booking-api.serviceId=booking-service
zuul.routes.checkin-api.serviceId=checkin-service
zuul.routes.fare-api.serviceId=fares-service
```

```
zuul.prefix=/api
```

```
zuul.routes.search-api.path=/search-path/**
zuul.routes.booking-api.path=/booking-path/**
zuul.routes.checkin-api.path=/checkin-path/**
zuul.routes.fare-api.path=/fare-path/**
```

Set sensitiveHeaders property on empty value to enable Authorization HTTP header forward.
By default, Zuul cut that header while forwarding our request to the target API which is incorrect because of the basic authorization demanded by our services behind gateway.

```
zuul.routes.search-api.sensitiveHeaders =
zuul.routes.booking-api.sensitiveHeaders =
zuul.routes.checkin-api.sensitiveHeaders =
zuul.routes.fare-api.sensitiveHeaders =
```

```
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
```

Add and commit changes in Git repo:

```
C:\Users\Aspire-Ramesh\config-repo>git add -A .
```

```
C:\Users\Aspire-Ramesh\config-repo>git commit -m "adding zuul properties"
```

Step5: Add **@EnableZuulProxy** and **@EnableDiscoveryClient** in boot strap class i.e., Application.java file.

@EnableZuulProxy

@EnableDiscoveryClient

@SpringBootApplication

```
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Step6: Start common-apigateway project. Ensure that both config and eureka servers were started before starting common-apigateway. Also start all microservices (fare, search, booking and checkin)

Step7: Make following changes in BrownFieldSiteController.java file in FlightsWebSite project.

```

@Controller
public class BrownFieldSiteController {
    @RequestMapping(value = "/search", method = RequestMethod.POST)
    public String greetingSubmit(@ModelAttribute UIData uiData, Model model) {
        //Flight[] flights = searchClient.postForObject("http://search-service/search/get",
        uiData.getSearchQuery(), Flight[].class); //with service id
        //Flight[] flights = searchClient.postForObject("http://common-apigateway/search-api/search/get",
        uiData.getSearchQuery(), Flight[].class); //common gateway and zuul route
        //Flight[] flights = searchClient.postForObject("http://common-apigateway/api/search-api/search/get",
        uiData.getSearchQuery(), Flight[].class); //with both zuul prefix and zuul route
        Flight[] flights = searchClient.postForObject("http://common-apigateway/api/search-path/search/get",
        uiData.getSearchQuery(), Flight[].class); //with both zuul prefix and zuul path
        uiData.setFlights(Arrays.asList(flights));
        model.addAttribute("uidata", uiData);
        return "result";
    }

    @RequestMapping(value = "/confirm", method = RequestMethod.POST)
    public String ConfirmBooking(@ModelAttribute UIData uiData, Model model) {
        ...
        //bookingId = bookingClient.postForObject("http://booking-service/booking/create", booking,
        long.class);
        //bookingId = bookingClient.postForObject("http://common-apigateway/booking-
        api/booking/create", booking, long.class);
        //bookingId = bookingClient.postForObject("http://common-apigateway/api/booking-
        api/booking/create", booking, long.class);
        bookingId = bookingClient.postForObject("http://common-apigateway/api/booking-
        path/booking/create", booking, long.class);
    }

    @RequestMapping(value = "/search-booking-get", method = RequestMethod.POST)
    public String searchBookingSubmit(@ModelAttribute UIData uiData, Model model) {
        //BookingRecord booking = bookingClient.getForObject("http://booking-service/booking/get/" + id,
        BookingRecord.class);
        //BookingRecord booking = bookingClient.getForObject("http://common-apigateway/booking-
        api/booking/get/" + id, BookingRecord.class);
        //BookingRecord booking = bookingClient.getForObject("http://common-apigateway/api/booking-
        api/booking/get/" + id, BookingRecord.class);
        BookingRecord booking = bookingClient.getForObject("http://common-apigateway/api/booking-
        path/booking/get/" + id, BookingRecord.class);
    }

    @RequestMapping(...)
    public String bookQuery(...) {

```



```

//long checkinId = checkInClient.postForObject("http://checkin-service/checkin/create", checkIn,
long.class);
//long checkinId = checkInClient.postForObject("http://common-apigateway/checkin-
api/checkin/create", checkIn, long.class);
//long checkinId = checkInClient.postForObject("http://common-apigateway/api/checkin-
api/checkin/create", checkIn, long.class);
long checkinId = checkInClient.postForObject("http://common-apigateway/api/checkin-
path/checkin/create", checkIn, long.class);
}
}

```

Note: By default, Zuul will strip the prefix and forward the request.

Step8: Add common gateway path in FareServiceProxy.java interface in BookingFlightTickets project.

```

//@FeignClient(name = "fares-service") //removed
@FeignClient(name = "common-apigateway/api/fare-path") //added
public interface FareServiceProxy {
    @RequestMapping(value = "/fares/get", method = RequestMethod.GET)
    Fare getFare(@RequestParam(value = "flightNumber") String flightNumber, @RequestParam(value =
"flightDate") String flightDate);
}

```

Step9: Add common gateway path in CheckinComponent.java file in CheckInCustomers project.

```

@Service
public class CheckinComponent {
    //private static final String bookingURL = "http://BOOKING-SERVICE/booking"; //removed
    private static final String bookingURL = "http://common-apigateway/api/booking-path/booking";
    ...
}

```

Start10: Start Fare, Search, Booking, Check microservices and PSS WebSite project.

6. API GATEWAY AND OAUTH

In this chapter we will implement both API Gateway as well as OAuth 2.0 with JWT token together.

Step1: Import all projects from MicroservicesAPIGatewaysWorkspace.

Step2: Also import authorization-server-jwt project from MicroservicesOAuthJWTWorkspace.

Step3: Add spring-security-oauth2 and spring-security-jwt dependencies across fares, search, booking and checkin microservices' pom.xml files.

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
</dependency>
```

Step4: Add JWT symmetric / asymmetric key across fare, search, booking and checkin microservices' bootstrap.properties files.

security.oauth2.resource.jwt.key-value=aspire
(OR)

security.oauth2.resource.jwt.key-uri=http://localhost:9090/oauth/token_key

Step5: Add Resource Server configuration file across fares, search, booking and checkin microservices' projects.

```
//fare microservice
package com.brownfield.pss.resourceserver.config;
@Configuration
@EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().authenticated().and()
            .requestMatchers().antMatchers("/fares/**").and().
            cors();
    }
}
```

```
//search microservice
package com.brownfield.pss.resourceserver.config;
@Configuration
@EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdapter {
```

```

        @Override
        public void configure(HttpSecurity http) throws Exception {
            http.authorizeRequests().anyRequest().authenticated().and()
                .requestMatchers().antMatchers("/search/**").and().
                cors();
        }
    }

//booking microservice
package com.brownfield.pss.resourceserver.config;
@Configuration
@EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().authenticated().and()
            .requestMatchers().antMatchers("/booking/**").and().
            cors();
    }
}

//checkin microservice
package com.brownfield.pss.resourceserver.config;
@Configuration
@EnableResourceServer
public class OAuth2ResourceServer extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().authenticated().and()
            .requestMatchers().antMatchers("/checkin/**").and().
            cors();
    }
}

```

Note: ResourceServer is not required for PSS WebSite

Step6: Add above package(s) to component scan in Application.java file across fares, search, booking and checkin microservices' projects.

```

//fare microservice
@SpringBootApplication(scanBasePackages = {
    "com.brownfield.pss.resourceserver.config",
    "com.brownfield.pss.fares.controller",
    "com.brownfield.pss.fares.component"
})
public class Application implements CommandLineRunner {}

```

```
//search microservice
@SpringBootApplication(scanBasePackages = {
    "com.brownfield.pss.resourceserver.config",
    "com.brownfield.pss.search.controller",
    "com.brownfield.pss.search.component"
})
```

```
//Booking Microservice
@SpringBootApplication(scanBasePackages = {
    "com.brownfield.pss.resourceserver.config",
    "com.brownfield.pss.book.controller",
    "com.brownfield.pss.book.component"
})
```

```
//Checkin Microservice
@SpringBootApplication(scanBasePackages = {
    "com.brownfield.pss.resourceserver.config",
    "com.brownfield.pss.checkin.controller",
    "com.brownfield.pss.checkin.component"
})
```

Step7: Make following changes only in BookingComponent.java file in Booking Microservice.

```
package com.brownfield.pss.book.component;
@Configuration
class AppConfig {
    @LoadBalanced
    @Bean
    public OAuth2RestTemplate restTemplate() {
        ResourceOwnerPasswordResourceDetails resource = new ResourceOwnerPasswordResourceDetails();
        resource.setUsername("adolfo");
        resource.setPassword("123");
        resource.setClientId("clientapp");
        resource.setClientSecret("123456");
        resource.setGrantType("password");
        resource.setScope(Arrays.asList(new String[] { "read_profile" }));
        resource.setAccessTokenUri("http://localhost:9090/oauth/token");
        DefaultOAuth2ClientContext clientContext = new DefaultOAuth2ClientContext();
        OAuth2RestTemplate restTemplate = new OAuth2RestTemplate(resource, clientContext);
        System.out.println("Access Token: " + restTemplate.getAccessToken());
        return restTemplate;
    }
}
```

```

@Service
//@EnableFeignClients
public class BookingComponent {
    //private static final String FareURL = "http://FARES-SERVICE/fares";
    private static final String FareURL = "http://common-apigateway/api/fare-path/fares";
    @Autowired
    private OAuth2RestTemplate restTemplate;
    // @Autowired
    // FareServiceProxy fareServiceProxy; //remove
    ...
    @Autowired
    public BookingComponent(BookingRepository bookingRepository, Sender sender,
        InventoryRepository inventoryRepository) {
        //this.restTemplate = new RestTemplate(); //remove
    }

    fare = restTemplate.getForObject(FareURL + "/get?flightNumber=" +
        record.getFlightNumber() + "&flightDate=" + record.getFlightDate(), Fare.class);
    //fare = fareServiceProxy.getFare(record.getFlightNumber(), record.getFlightDate());
}

```

Note: Feign doesn't use RestTemplate directly. But use following workaround solution to make it to work:

<https://stackoverflow.com/questions/29439653/spring-cloud-feign-with-oauth2resttemplate>

Step8: Make following changes only in CheckInComponent.java file.

```

@Configuration
class AppConfig {
    @LoadBalanced
    @Bean
    public OAuth2RestTemplate restTemplate() {
        ResourceOwnerPasswordResourceDetails resource = new ResourceOwnerPasswordResourceDetails();
        resource.setUsername("adolfo");
        resource.setPassword("123");
        resource.setAccessTokenUri("http://localhost:9090/oauth/token");
        resource.setClientId("clientapp");
        resource.setClientSecret("123456");
        resource.setGrantType("password");
        resource.setScope(Arrays.asList(new String[] {
            "read_profile"
        }));
        DefaultOAuth2ClientContext clientContext = new DefaultOAuth2ClientContext();
        OAuth2RestTemplate restTemplate = new OAuth2RestTemplate(resource, clientContext);
        System.out.println("Access Token: " + restTemplate.getAccessToken());
        return restTemplate;
    }
}

```

```
}
}
```

```
@Service
public class CheckinComponent {
//private static final String bookingURL = "http://BOOKING-SERVICE/booking";
private static final String bookingURL = "http://common-apigateway/api/booking-path/booking";
```

```
@Autowired
private OAuth2RestTemplate restTemplate; // Added by Ramesh
...
}
```

Step9: Add spring-security-oauth2 dependency in FlightsWebSite's pom.xml file since web site needs OAuth2RestTemplate.

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

Step10: Replace RestTemplate with OAuth2RestTemplate in BrownFieldSiteController.java in FlightsWebSite project.

```
@Configuration
class AppConfig {
    @LoadBalanced
    @Bean
    public OAuth2RestTemplate restTemplate() {
        ResourceOwnerPasswordResourceDetails resource = new ResourceOwnerPasswordResourceDetails();
        resource.setUsername("adolfo");
        resource.setPassword("123");
        resource.setClientId("clientapp");
        resource.setClientSecret("123456");
        resource.setGrantType("password");
        resource.setScope(Arrays.asList(new String[] {
            "read_profile"
        }));
        resource.setAccessTokenUri("http://localhost:9090/oauth/token");
        DefaultOAuth2ClientContext clientContext = new DefaultOAuth2ClientContext();
        OAuth2RestTemplate restTemplate = new OAuth2RestTemplate(resource, clientContext);
        System.out.println("Access Token: " + restTemplate.getAccessToken());
        return restTemplate;
    }
}
```

```

@Controller
public class BrownFieldSiteController {
    private static final Logger logger = LoggerFactory.getLogger(BrownFieldSiteController.class);
    @Autowired
    OAuth2RestTemplate searchClient;
    @Autowired
    OAuth2RestTemplate bookingClient;
    @Autowired
    OAuth2RestTemplate checkInClient;
    ...
}

```

Step11: Add new Spring Security Config file in website project package com.brownfield.pss.springsecurity.configuration;

@EnableWebSecurity

```

public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated().and()
            .formLogin();
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user1@example.com").password("user1").roles("USER").and()
            .withUser("user2@example.com").password("user2").roles("USER");
    }
}

```

Note: Optionally remove security.user.name=guest and security.user.password=guest123 properties from pss-website.properties from Git Repo.

Step12: Add component scan in Application.java file. Also remove @EnableGlobalMethodSecurity.

//@EnableGlobalMethodSecurity

```

@SpringBootApplication(scanBasePackages = {
    "com.brownfield.pss.client",
    "com.brownfield.pss.springsecurity.configuration"
})
@EnableDiscoveryClient
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}

```

```
}
```

Step13: Start ConfigServer, EurekaServer, common-apigateway, authorization-server-jwt, fare, search, booking, checkin microservices and website project.

ASPIRE-RAMESH

7. MICROSERVICES WITH AWS

Cloud computing promises many benefits such as cost advantage, speed, agility, flexibility, and elasticity. There are many cloud providers such as Amazon (AWS), Microsoft (Azure), Google (Google cloud), Pivotal (Cloud Foundry), IBM (Blue Mix), Red Hat (Open Shift), Rackspace (Rackspace), etc.

Manual deployment could severely challenge the microservices rollouts. With many server instances running, this could lead to significant operational overheads. Moreover, the chances of errors are high in this manual approach.

Microservices require **elastic** cloud-like infrastructure which can automatically provision VMs or containers, automatically deploy applications, adjust traffic flows, replicate new version to all instances, and gracefully phase out older versions. The automation also takes care of scaling up elastically by adding containers or VMs on demand, and scaling down when the load falls below threshold.

Step1: Create new free account in AWS cloud.

Step2: Create EC2 free tier instance.

Select "Create a new key pair" → Key pair name "MicroservicesAwsKeyPair" → Click on "Download Key Pair" button → Save it in D:\MICROSERVICES ADVANCED\Documents\Misc folder and give name as "MicroservicesAwsKeyPair.pem"

Step3: Get connection information from AWS by click on "Connect" button.

Copy sample command `ssh -i "MicroservicesAwsKeyPair.pem" ec2-user@ec2-52-14-98-110.us-east-2.compute.amazonaws.com` from 'Connect To Your Instance' popup.

Connect with EC2 instance from our local machine by using ssh client such as **putty** or git-bash.

Install putty from softwares folder.

Navigate to D:\Program Files\PuTTY and double click on **puttygen.exe** file.

Following steps are used to convert .pem into .ppk format because putty needs .ppk format but not .pem format.

1. Open PuTTYgen
2. Click on "Load" button
3. Set the file type to *.*
4. Browse to, and Open your .pem file
5. PuTTY will auto-detect everything it needs, and we just need to click "Save private key" and we can save our ppk key for use with PuTTY

Navigate to D:\Program Files\PuTTY and double click on **putty.exe** file.

Host Name (or IP address) : 52.14.98.110

Port : 22

Saved Sessions (optional): EC2_Instance1

Click on Save button

Expand Connection → Expand SSH → Click on Auth → Import "Private Key file for authentication" →

Browse MicroservicesAwsKeyPair.ppk

Click on "Open"

Login as: ec2-user

```
[ec2-user@ip-172-31-17-150 ~]$
```

Step4: Optionally we can use git-bash in place of putty.

open git-bash and navigate to .pem file location in our case 'D:\MICROSERVICES
ADVANCED\Documents\Misc' and run above command.

```
$ ssh -i "MicroservicesAwsKeyPair.pem" ec2-user@ec2-52-14-98-110.us-east-2.compute.amazonaws.com
```

Are you sure you want to continue connecting(yes/no)? yes

```
[ec2-user@ip-172-31-17-150 ~]$
```

Step5: Check java version

```
[ec2-user@ip-172-31-17-150 ~]$ java -version  
java version "1.7.0_201"
```

Remove java 7 because spring boot needs java 8.

```
[ec2-user@ip-172-31-17-150 ~]$ sudo yum remove java-1.7.0-openjdk
```

Install java8 by using below command:

```
[ec2-user@ip-172-31-17-150 ~]$ sudo yum install java-1.8.0
```

Check java version now:

```
[ec2-user@ip-172-31-17-150 ~]$ java -version  
openjdk version "1.8.0_201"
```

Step6: Create Oracle DB in AWS RDS cloud

Goto Services → Click on "RDS" under Database category → Click on "Create database" button →
Choose 'Oracle' → Choose "Oracle Standard Edition" → Select "Free Tier" → click on Next → Specify DB
Details

DB Instance Identifier: aws

Master Username: awsuser

Master Password: aspire1234

Database Name: ORCL

Database port: 1521

aspireorcl.co6tmhd9bey4.us-east-1.rds.amazonaws.com:1521

Step7: Connect with Oracle DB in cloud by using DB Visualizer / Toad

Install and Start DB Visualizer → Goto 'Database' menu → click on "Create Database Connection" →
click on "Use Wizard" →

Enter connection alias as: 'aspireawsdb'

Select database driver: Oracle Thin

Database Server: aws.co6tmhd9bey4.us-east-1.rds.amazonaws.com

Database Userid: awsuser

Database Password: aspire1234

Click on "Next" button

Step8: Create tablespaces, schemas, and optionally create tables and sequences by using 'Airline_PSS_Schema.doc' in Misc folder.

Step9: Install Rabbitmq by using following commands.

```
[ec2-user@ip-172-31-17-150 ~]$ sudo yum install -y erlang
[ec2-user@ip-172-31-17-150 ~]$ sudo wget https://www.rabbitmq.com/releases/rabbitmq-server/v3.5.6/rabbitmq-server-3.5.6-1.noarch.rpm
[ec2-user@ip-172-31-17-150 ~]$ sudo rpm -Uvh rabbitmq-server-3.5.6-1.noarch.rpm
[ec2-user@ip-172-31-17-150 ~]$ sudo chkconfig rabbitmq-server on
[ec2-user@ip-172-31-17-150 ~]$ sudo /sbin/service rabbitmq-server start
Starting rabbitmq-server: SUCCESS
rabbitmq-server.
[ec2-user@ip-172-31-17-150 ~]$ sudo rabbitmqctl status
[ec2-user@ip-172-31-17-150 ~]$ sudo rabbitmq-plugins enable rabbitmq_management
The following plugins have been enabled:
  mochiweb
  webmachine
  rabbitmq_web_dispatch
  amqp_client
  rabbitmq_management_agent
  rabbitmq_management
Applying plugin configuration to rabbit@ip-172-31-17-150... started 6 plugins.
[ec2-user@ip-172-31-17-150 ~]$ sudo /sbin/service rabbitmq-server restart
```

By default, the rabbitmq server installation comes with 'guest' user. But this can only connect via localhost.

Hence create new account in rabbitmq to access outside of localhost.

```
[ec2-user@ip-172-31-17-150 ~]$ sudo rabbitmqctl add_user aspire aspire
[ec2-user@ip-172-31-17-150 ~]$ sudo rabbitmqctl set_permissions -p / aspire ".*" ".*" ".*"
[ec2-user@ip-172-31-17-150 ~]$ sudo rabbitmqctl set_user_tags aspire administrator
[ec2-user@ip-172-31-17-150 ~]$ sudo /sbin/service rabbitmq-server restart
```

Note: Refer <https://gist.github.com/joshdvir/e4124a6494a6f6b8ba7e> for more details.

Access RabbitMq from web browser:

<http://52.14.98.110:15672/>

By default, above port is not accessible from outside. Hence, we need to change security configuration in EC2 instance.

Goto security-group in aws → select 'launch-wizard-3' → Click on 'Inbound' tab → Click on 'Edit' → Add Rule

Type[All ICMP -IPv4], Protocol [ICMP], Port Range [0-65535], Source [Anywhere] --> save

Type[All TCP], Protocol [TCP], Port Range [0-65535], Source [Anywhere] --> save

Instead of adding multiple rules as above, we rather can add single rule as given below:
Type[All traffic], Protocol [All], Port Range [0-65535], Source [Anywhere] --> save

Access RabbitMq from web browser again:

<http://52.14.98.110:15672/>

Enter username as 'aspire' and password as 'aspire'

Step10: Start STS, Import all projects from 'MicroservicesAwsWorkspace' workspace. Ensure that aws db properties in application.properties file across all microservice projects were set.

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

spring.datasource.url=jdbc:oracle:thin:@aws.cqssuvm1qmtx.us-east-2.rds.amazonaws.com:1521:ORCL

...

Step11: Configure Rabbitmq server details in search, booking and checkin microservices.

spring.rabbitmq.host=52.14.98.110

spring.rabbitmq.port=5672

spring.rabbitmq.username=aspire

spring.rabbitmq.password=aspire

Step12: Create Fat jar for all microservices and place them under designated folder.

Step13: Winscp is needed to copy files from our machine to cloud machine (remote machine). Hence install WinScp (WinSCP-5.13.7-Setup.exe) from Softwares folder. Also configure winscp.

Start 'WinSCP'

File Protocol: SFTP

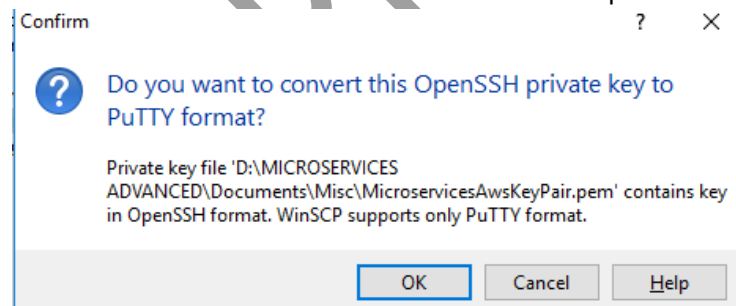
Host Name: 52.14.98.110

Port Number: 22

User name: ec2-user

Password: Not required!

Click on Advanced --> SSH --> Authentication --> Upload Private key file: MicroservicesAwsKeyPair.ppk



Click on "Ok".

Save converted private key (MicroservicesAwsKeyPair.ppk) in same location as .pem file.

Step14: Copy above fat jars from our machine to EC2 instance by using WinScp.

Step15: Run microservices from EC2 instance in cloud.

```
[ec2-user@ip-172-31-17-150 ~]$ java -jar FaresFlightTickets.jar &
```

```
[ec2-user@ip-172-31-17-150 ~]$ disown
```

```
[ec2-user@ip-172-31-17-150 ~]$ ps -a
```

The below command gives memory info:

```
[ec2-user@ip-172-31-17-150 ~]$ cat /proc/meminfo
```

```
[ec2-user@ip-172-31-17-150 ~]$ ps -aux | grep java
```

Check microservices' health:

<http://52.14.98.110:8081/health>

```
[ec2-user@ip-172-31-17-150 ~]$ java -jar SearchFlightTickets.jar &
```

```
[ec2-user@ip-172-31-17-150 ~]$ disown
```

<http://52.14.98.110:8090/health>

```
[ec2-user@ip-172-31-17-150 ~]$ cat /proc/meminfo
```

Step16: In Free-tier eligible EC2 instance the RAM size is only 1GB per instance. So far Fare, Search and Rabbitmq instances were running. Hence MemAvailable is low to run remaining booking, checkin and website instances. Hence create one more Free-tier eligible EC2 instance in Aws cloud.

Step17:

Configure Fare Microservices' IP address in BookingComponent.java file

```
private static final String FareURL = "http://52.14.98.110:8081/fares";
```

Create Fat Jar for Booking Microservice

Copy above fat jar from our machine to newly created EC2 instance by using WinScp by referring step13.

Connect with newly created EC2 instance by referring step3:

Uninstall java7 and install java8 by referring step5:

Navigate to /home/ec2-user/Microservices and run Booking Microservice in newly created EC2 instance by using below command:

```
[ec2-user@ip-172-31-20-204 Microservices]$ java -jar BookingFlightTickets.jar &
```

```
[ec2-user@ip-172-31-20-204 Microservices]$ disown
```

Check microservices' health:

<http://18.218.192.49:8060/health>

Configure Booking Microservices' IP address in CheckInComponent.java file

```
private static final String bookingURL = "http://18.218.192.49:8060/booking";
```

Create Fat Jar for CheckIn Microservice

Copy above fat jar from our machine to newly created EC2 instance by using WinScp.

Navigate to /home/ec2-user/Microservices and run CheckIn Microservice in newly created EC2 instance by using below command:

```
[ec2-user@ip-172-31-20-204 Microservices]$ java -jar CheckInCustomers.jar &
```

```
[ec2-user@ip-172-31-20-204 Microservices]$ disown
```

Check microservices' health:

<http://18.218.192.49:8070/health>

Configure Search, Booking and CheckIn Microservices' IP addresses in BrownFieldSiteController.java and Application.java files

<http://52.14.98.110:8090/search/get>

<http://18.218.192.49:8060/booking/create>

<http://18.218.192.49:8060/booking/get/>

<http://18.218.192.49:8070/checkin/create>

Create Fat Jar for Website

Copy above fat jar from our machine to newly created EC2 instance by using WinScp.

Navigate to /home/ec2-user/Microservices and run website in newly created EC2 instance by using below command:

```
[ec2-user@ip-172-31-20-204 Microservices]$ java -jar FlightsWebSite.jar &
```

```
[ec2-user@ip-172-31-20-204 Microservices]$ disown
```

Check microservices' health:

<http://18.218.192.49:8070/health>