

Securing Microservices

Microservices are the components that can be self executable and independently deployable may be in separate vm per each micro service or in cloud infrastructure. Microservices may offer APIs or web applications. Our sample application, NovelHealthCare, flybypoints etc.

Enabling Secure Socket Layer.

HTTP transfers data in plain text, but data transfer over the Internet in plain text is not secure. We definitely don't want to compromise user data, so we will provide the most secure way to access our web application. Therefore, we need to encrypt the information that is exchanged between the end user and our application. We'll use **Secure Socket Layer (SSL)** or **Transport Security Layer (TSL)** to encrypt the data.

SSL is a protocol designed to provide security (encryption) for network communications. HTTP associates with SSL to provide the secure implementation of HTTP, known as **Hyper Text Transfer Protocol Secure**, or **Hyper Text Transfer Protocol over SSL (HTTPS)**. HTTPS makes sure that the privacy and integrity of the exchanged data is protected. It also ensures the authenticity of websites visited. This security centers around the **distribution of signed digital certificates between the server hosting the application**, the end user's machine, and a third-party trust store server. Let's see how this process takes place:

1. The end user sends the request to the web application, for example <http://twitter.com>, using a web browser.
2. On receiving the request, the server redirects the browser to <http://twitter.com>, using the HTTP code 302.
3. The end user's browser connects to <https://twitter.com>, and, in response, the server provides the certificate containing the digital signature to the end user's browser.
4. The end user's browser receives this certificate and sends it to a trusted Certificate Authority (CA) for verification.
5. Once the certificate gets verified all the way to the root CA, an encrypted communication is established between the end user's browser and the application hosting server.

Securing Microservices



Note: Although SSL ensures security in terms of encryption and web application authenticity, it does not safeguard against phishing and other attacks. Professional hackers can decrypt information sent using HTTPS.

Now, after going over the basics of SSL, let's implement it for our sample project. We don't need to implement SSL for all microservices. All microservices will be accessed using our proxy or edge server; Zuul-server by the external environment, except our new microservice, security-service, which we will introduce in this chapter for authentication and authorization.

First, we'll set up SSL in edge server. **We need to have the keystore that is required for enabling SSL in embedded Tomcat.** We'll use the self-signed certificate for now. We'll use Java keytool to generate the keystore using the following command. We can use any other tool also:

```
keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -ext san=dns:localhost -storepass password -validity 365 -keysize 2048
```

It asks for information such as name, address details, organization, and so on (see the following screenshot):

Securing Microservices

```
C:\dev\workspace\ms\online-table-reservation-6>keytool -genkey -keyalg RSA -alias selfsigned -keystore
what is your first and last name?
[Unknown]: localhost
what is the name of your organizational unit?
[Unknown]: org unit
what is the name of your organization?
[Unknown]: org
what is the name of your City or Locality?
[Unknown]: city
what is the name of your State or Province?
[Unknown]: state
what is the two-letter country code for this unit?
[Unknown]: CN
Is CN=localhost, OU=org unit, O=org, L=city, ST=state, C=CN correct?
[no]: yes

Enter key password for <selfsigned>
(RETURN if same as keystore password):
Re-enter new password:

C:\dev\workspace\ms\online-table-reservation-6>
```

Be aware of the following points to ensure the proper functioning of self-signed certificates:

1) Use `-ext` to define **Subject Alternative Names (SAN)**. we can also use IP (for example, `san=ip:190.19.0.11`). Earlier, use of the hostname of the machine, where application deployment takes place was being used as most common name (CN). It prevents the `java.security.cert.CertificateException for No name matching localhost found`.

2) we can use a browser or OpenSSL to download the certificate. Add the newly generated certificate to the `cacerts` keystore located at `jre/lib/security/cacerts` inside active JDK/JRE home directory by using the `keytool -importcert` command. Note that **changeit** is the default password for the `cacerts` keystore. Run the following command:

```
keytool -importcert -file path/to/.crt -alias <cert alias> -keystore
<JRE/JAVA_HOME>/jre/lib/security/cacerts -storepass changeit
```

Note:

Self-signed certificates can be used only for development and testing purposes. The use of these certificates in a production environment does not provide the required security. Always use the certificates provided and signed by trusted signing authorities in production environments. Store your private keys safely.

Now, after putting the generated `keystore.jks` in the **src/main/resources** directory of the our project, along with `application.yml`, `application.properties` file, we can update this information in `EdgeServer application.yml` / `application.properties` file as follows:

Securing Microservices

```
server.ssl.key-store=keystore.js  
server.ssl.key-password=password  
server.ssl.key-store-password=password
```

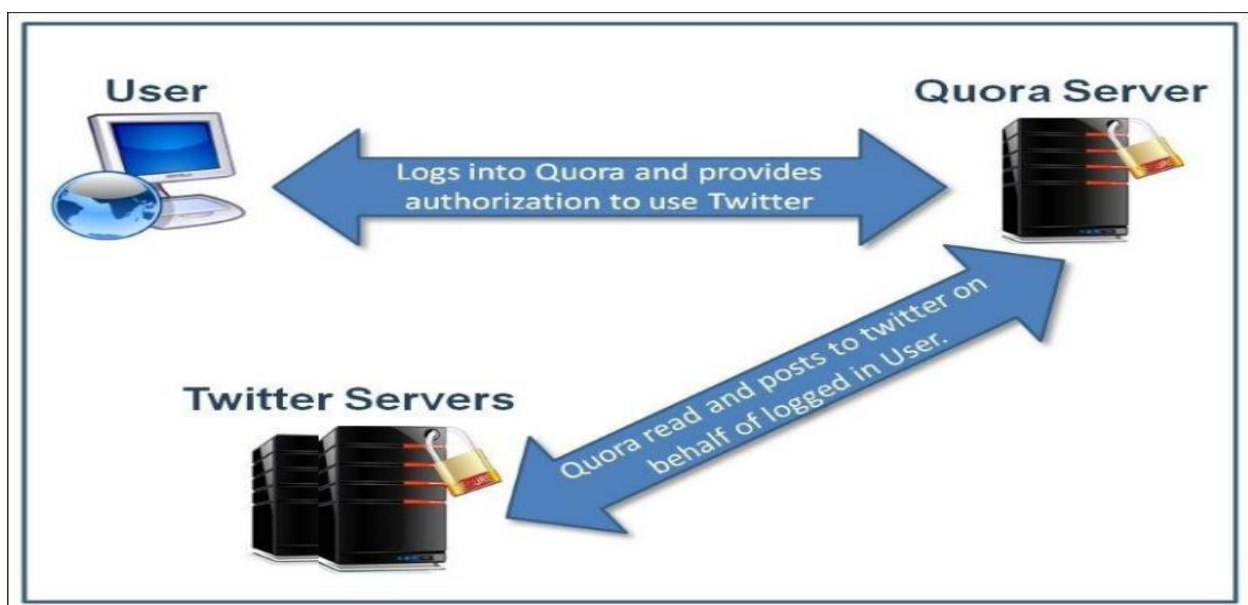
Rebuild the application to use the HTTPS.

Note:

- 1) The key store file can be stored in the preceding class path in Tomcat version 7.0.66+ and 8.0.28+. For older versions, we can use the path of the key store file for the server:ssl:key-store value.
- 2) Similarly, we can configure SSL for other microservices.

Authentication and authorization

Providing authentication and authorization is mandatory for web applications. The new paradigm that has evolved over the past few years is OAuth. OAuth is an **open authorization** mechanism, implemented in every major web application. Web applications can access each other's data by implementing the OAuth standard. It has become the most popular way to authenticate oneself for various web applications. Like on www.quora.com, you can register, and login using our Google or Twitter login IDs. It is also more user friendly, as client applications (for example, www.quora.com) don't need to store the user's passwords. The end user does not need to remember one more user ID and password.



Securing Microservices

OAuth 2.0

The **Internet Engineering Task Force (IETF)** governs the standards and specifications of OAuth. OAuth 1.0a was the most recent version before OAuth 2.0 that was having a fix for **session-fixation security flaw in the OAuth 1.0**.

OAuth 1.0 and 1.0a were very different from OAuth 2.0. **OAuth 1.0 relies on security certificates and channel binding**. OAuth 2.0 does not support security certification and channel binding. It works completely on Transport Layer Security (TLS). Therefore, OAuth 2.0 does not provide backward compatibility.

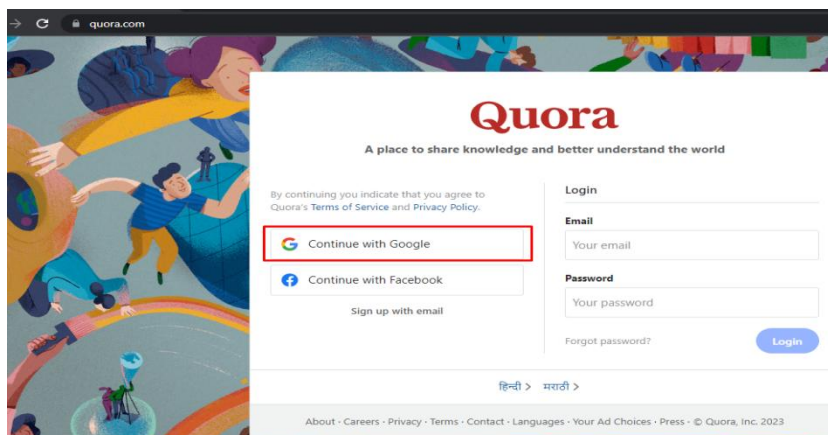
Use Cases

- 1) As discussed, it can be used for authorization. We might have seen it in various applications, displaying messages such as sign in using Facebook or sign in using Twitter.
- 2) Applications can use it to read data from other applications, such as by integrating a Facebook widget into the application, or having a Twitter feed on our blog.
- 3) Or, the opposite of the previous point can be true: we enable other applications to access the end user's data.

OAuth 2.0 specification

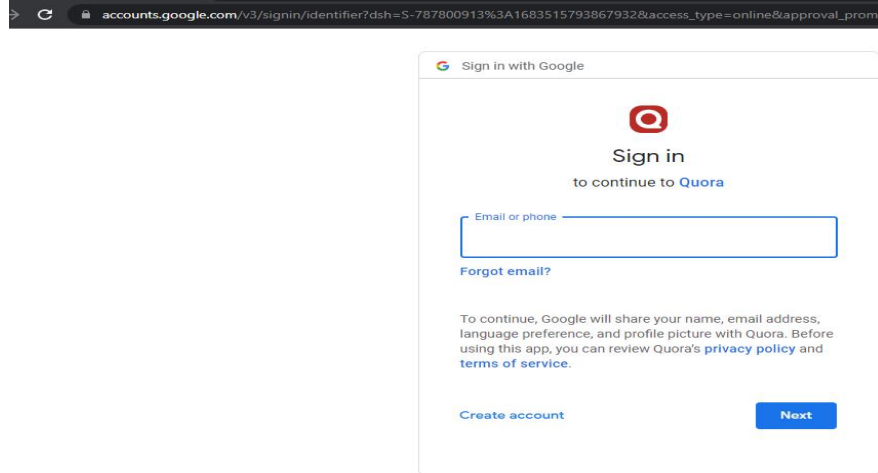
Before understanding the OAuth 2.0 specifications in a concise manner. Let's first see how signing in using Quora works.

- 1) The user visits the Quora home page. It shows various login options. We'll explore the process of the Continue with Google.



Securing Microservices

2) When the user clicks on the Continue with Google link, Quora opens a new window (in Chrome) that redirects the user to the `www.gmail.com` application. During this process few web applications redirect the user to the same opened tab/window.



3) In this new window/tab, the user signs in to `www.gmail.com` with their credentials.

4) If the user has not authorized the Quora application to use their data earlier, google asks for the user's permission to authorize Quora to access the user's information.

If the user has already authorized Quora, then this step is skipped.

5) After proper authentication, Google redirects the user to Quora's redirect URI with an authentication code.

6) Quora sends the client ID, client secret, and authentication code (sent by Google in step 5 to Google when Quora redirect URI entered in the browser.

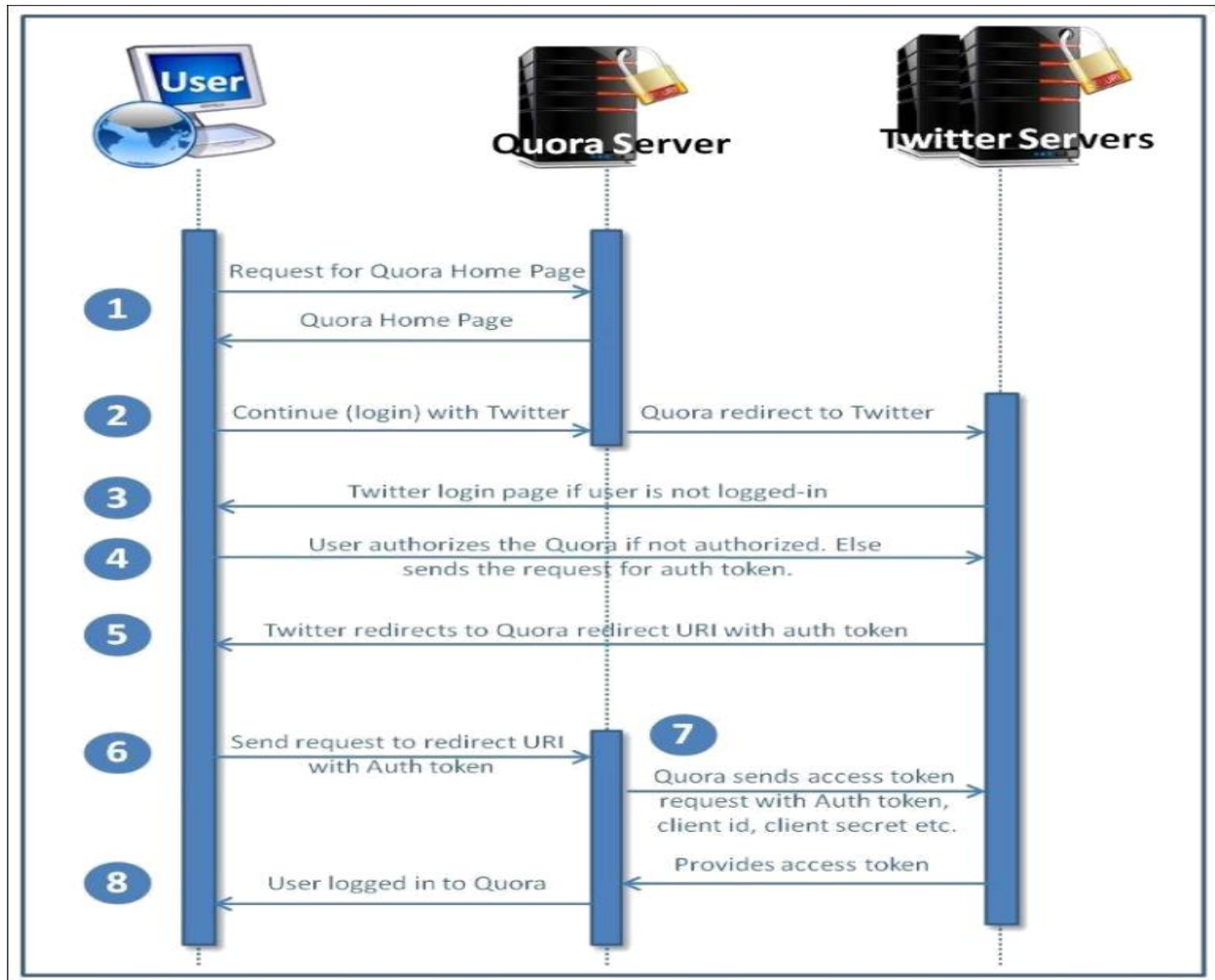
7) After validating these parameters, Google sends the access token to Quora.

8) The user is logged in to Quora on successful retrieval of the access token.

9) Quora may use this access token to retrieve user information from Google.

Here we must be wondering how Twitter got Quora's redirect URI, client ID, and secret token. **Quora works as a client application and Twitter as an authorization server.** Quora, as a client, registered on Twitter by using Twitter's OAuth implementation to use resource owner (end user) information. Quora provides a redirect URI at the time of registration. Twitter provides the client ID and secret token to Quora. It works this way. In OAuth 2.0, user information is known as user resources. Twitter provides a resource server and an authorization server.

Securing Microservices

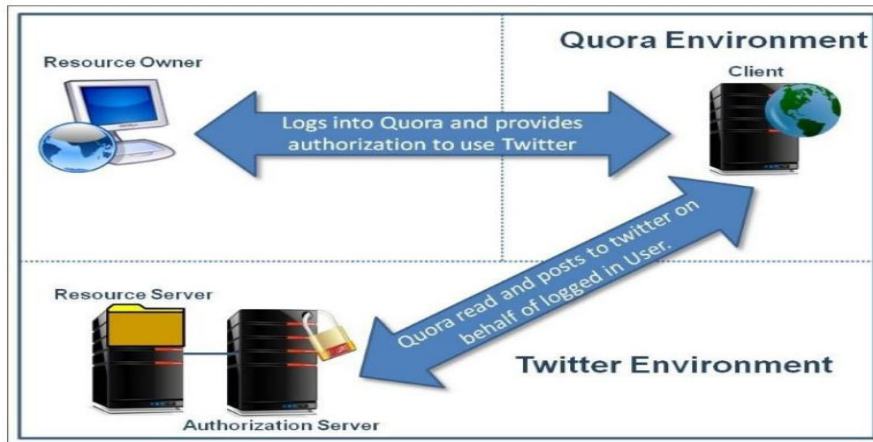


OAuth 2.0 roles

There are four roles defined in the OAuth 2.0 specifications:

- 1) Resource owner
- 2) Resource server
- 3) Client(Chrome Browser, Quora App etc)
- 4) Authorization server

Securing Microservices



Resource owner

For the Quora sign in using Twitter example, the **Twitter user was the resource owner**. The **resource owner is an entity that owns the protected resources** (for example user handle, tweets and so on) that are to be shared. This entity can be an application or a person. **We call this entity the resource owner because it can only grant access to its resources**. Specification also defines, when resource owner is a person, it is referred to as an end user.

Resource server

The resource server hosts the protected resources. It should be capable of serving the access requests to these resources using access tokens. For the Quora sign in using Twitter example, Twitter is the resource server.

Client

For the Quora sign in using Twitter example, **Quora is the client**. The client is the application that makes access requests for protected resources to the resource server on behalf of the resource owner.

Authorization server

The authorization server provides different tokens to the client application, such as access tokens or refresh tokens, only after the resource owner authenticates themselves.

*OAuth 2.0 does not provide any specifications for interactions between the **resource server and the authorization server**. Therefore, the authorization server and resource server can be on the same server, or can be on a separate one. A single authorization server can also be used to issue access tokens for multiple resource servers.*

Securing Microservices

OAuth 2.0 client registration

The client that communicates with the authorization server to obtain the access key for a resource should first be registered with the authorization server. **The OAuth 2.0 specification does not specify the way a client registers with the authorization server.** Registration does not require direct communication between the client and the authorization server. Registration can be done using self-issued or third-party-issued assertions. The authorization server obtains the required client properties using one of these assertions. Let's see what the client properties are:

- ✓ Client type
- ✓ Client redirect URI
- ✓ Any other information required by the authorization server, for example client name, description, logo image, contact details, acceptance of legal terms and conditions, and so on.

Client types

There are two types of client described by the specification, based on their ability to maintain the confidentiality of client credentials: **confidential and public**. Client credentials are secret tokens issued by the authorization server to clients in order to communicate with them.

Confidential client type

This is a client application that keeps passwords and other credentials securely or maintains them confidentially. In the Quora sign in using Twitter example, the Quora app server is secure and has restricted access implementation. Therefore, it is of the confidential client type. Only the Quora app administrator has access to client credentials.

Public client type

These are client applications that do not keep passwords and other credentials securely or maintain them confidentially. Any native app on mobile or desktop, or an app that runs on browser, are perfect examples of the public client type, as these keep client credentials embedded inside them. Hackers can crack these apps and the client credentials can be revealed.

Based on the OAuth 2.0 client types, a client can have the following profiles:

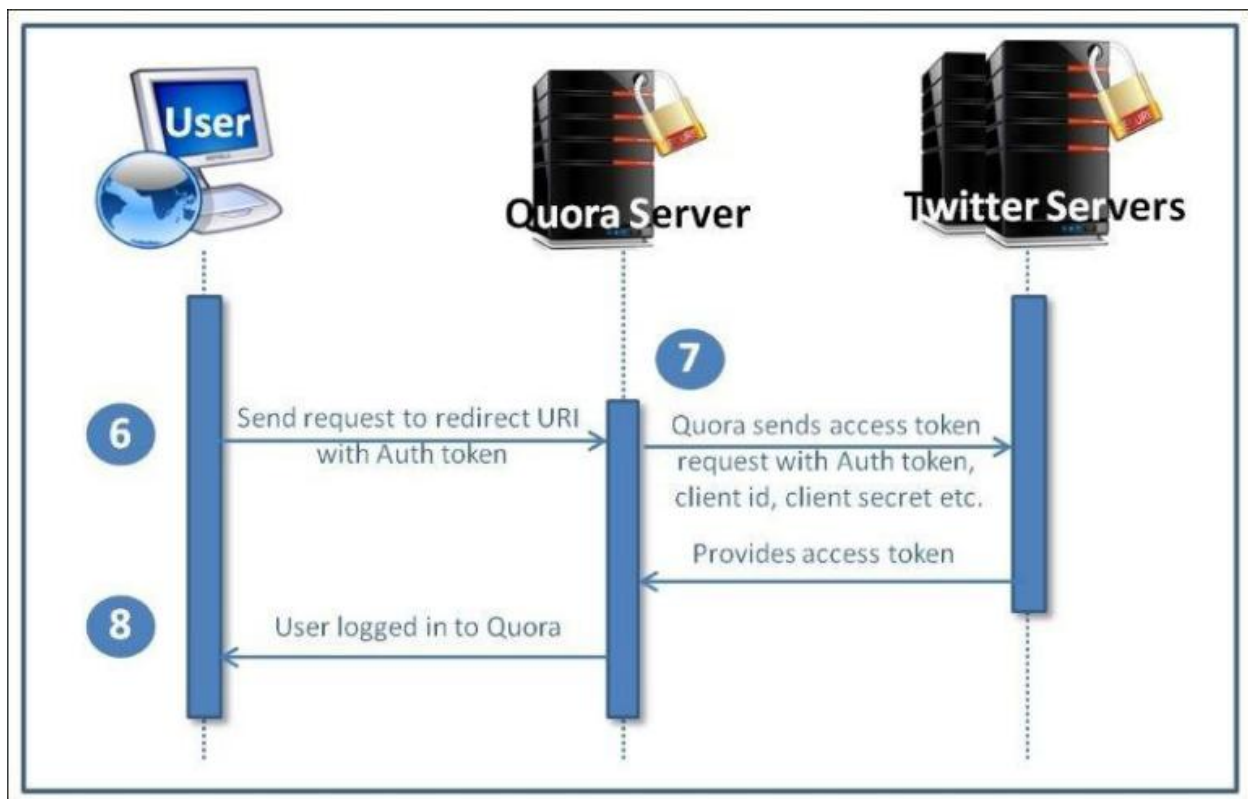
- ✓ Web application
- ✓ User agent-based application

Securing Microservices

- ✓ Native application

Web application

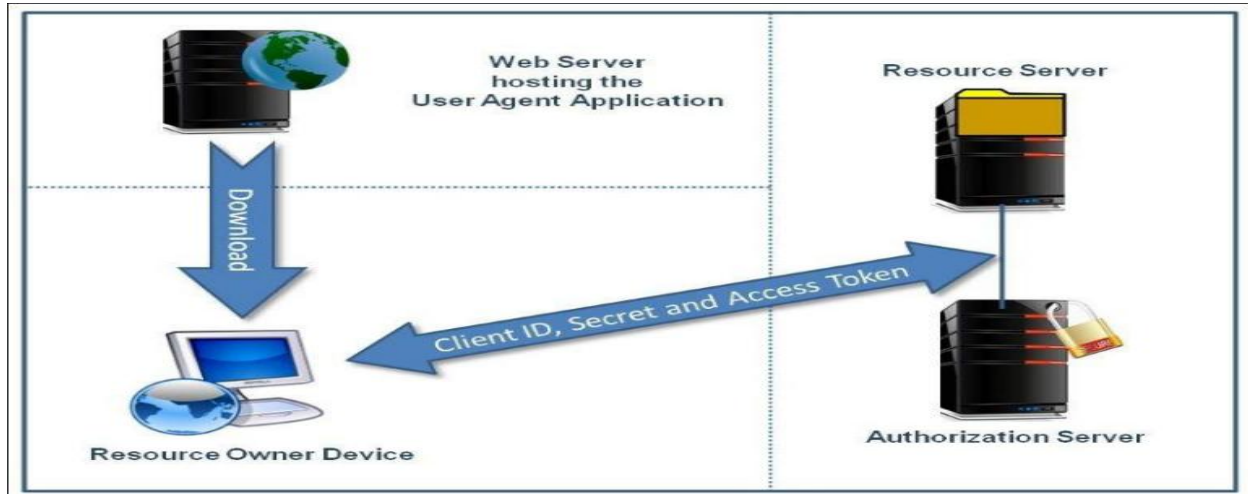
The Quora web application used in the Quora sign in using Twitter example is a perfect example of an OAuth 2.0 web application client profile. Quora is a confidential client running on a web server. The resource owner (end user) accesses the Quora application (OAuth 2.0 client) on the browser (user agent) using a HTML user interface on his device (desktop/tablet/cell phone). The resource owner cannot access the client (Quora OAuth 2.0 client) credentials and access tokens, as these are stored on the web server. We can see this behavior in the diagram of the OAuth 2.0 sample flow. See steps 6 to 8 in the following figure:



User agent-based application

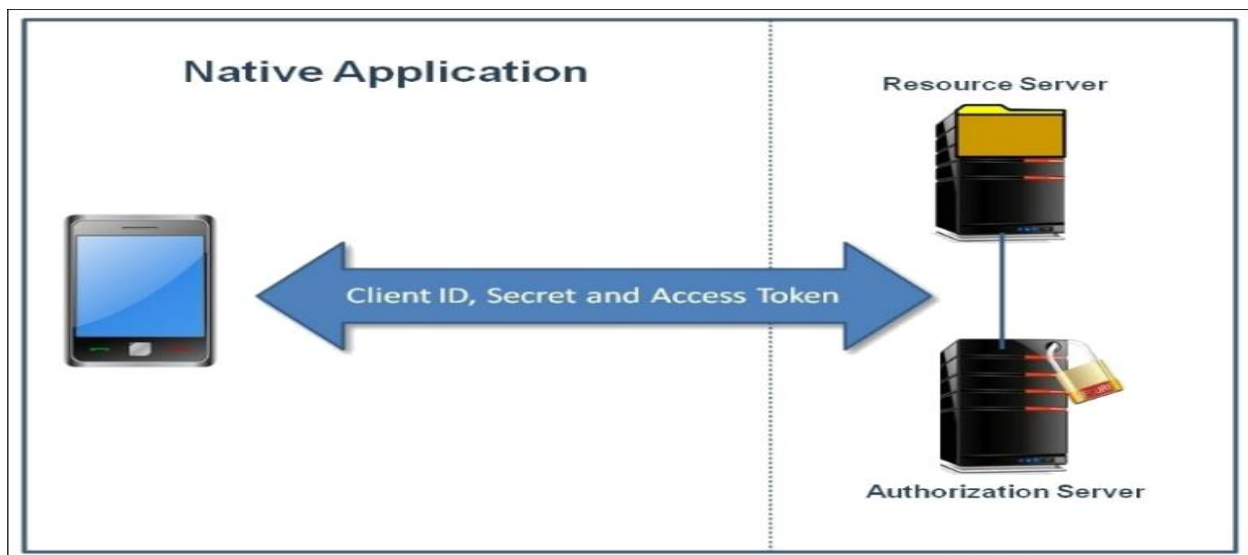
User agent-based applications are of the public client type. Here, though, the application resides in the web server, but the resource owner downloads it on the user agent (for example, a web browser) and then executes the application. Here, the downloaded application that resides in the user agent on the resource owner's device communicates with the authorization server. The resource owner can access the client credentials and access tokens. A gaming application is a good example of such an application profile.

Securing Microservices



Native application

Native applications are similar to user agent-based applications, except these are installed on the resource owner's device and execute natively, instead of being downloaded from the web server, and then executes inside the user agent. Many native clients (mobile apps) we download on our mobile are of the native application type. Here, the platform makes sure that other application on the device do not access the credentials and access tokens of other applications. In addition, native applications should not share client credentials and OAuth tokens with servers that communicate with native applications.



Client identifier

It is the authorization server's responsibility to provide a unique identifier to the registered client. This client identifier is a string representation of the information provided by the

Securing Microservices

registered client. The authorization server needs to make sure that this identifier is unique. The authorization server should not use it on its own for authentication.

The OAuth 2.0 specification does not specify the size of the client identifier. The authorization server can set the size, and it should document the size of the client identifier it issues.

Client authentication

The authorization server should authenticate the client based on their client type. The authorization server should determine the authentication method that suits and meets security requirements. It should only use one authentication method in each request.

Typically, the authorization server uses a set of client credentials, such as the client password and some key tokens, to authenticate confidential clients.

The authorization server may establish a client authentication method with public clients. However, it must not rely on this authentication method to identify the client, for security reasons.

A client possessing a client password can use basic HTTP authentication. OAuth 2.0 does not recommend sending client credentials in the request body. It recommends using TLS and brute force attack protection on endpoints required for authentication.

OAuth 2.0 protocol endpoints

An endpoint is nothing but a URI we use for REST or web components such as Servlet or JSP. OAuth 2.0 defines three types of endpoint. Two are authorization server endpoints and one is a client endpoint:

- ✓ Authorization endpoint (authorization server endpoint)
- ✓ Token endpoint (authorization server endpoint)
- ✓ Redirection endpoint (client endpoint)

Authorization endpoint

This endpoint is responsible for verifying the identity of the resource owner and, once verified, obtaining the authorization grant.

Securing Microservices

The authorization server require TLS for the authorization endpoint. The endpoint URI must not include the fragment component. The authorization endpoint must support the HTTP GET method.

The specification does not specify the following:

- ✓ The way the authorization server authenticates the client.
- ✓ How the client will receive the authorization endpoint URI. Normally, documentation contains the authorization endpoint URI, or the client obtains it at the time of registration.

Token endpoint

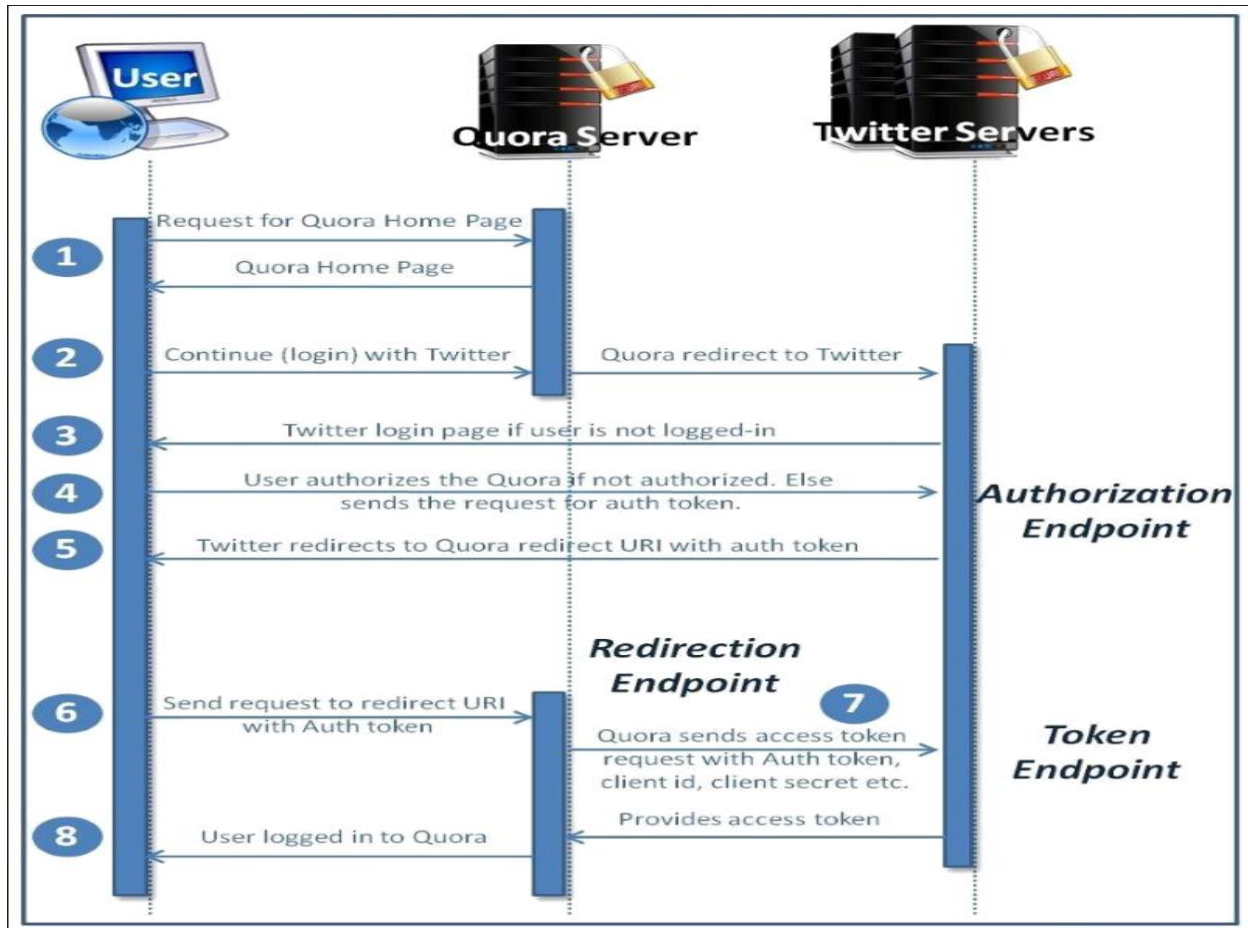
The client calls the token endpoint to receive the access token by sending the authorization grant or refresh token. The token endpoint is used by all authorization grants except an implicit grant.

Like the authorization endpoint, the token endpoint also requires TLS. The client must use the HTTP POST method to make the request to the token endpoint. Like the authorization endpoint, the specification does not specify how the client will receive the token endpoint URI.

Redirection endpoint

The authorization server redirects the resource owner's user agent (for example, a web browser) back to the client using the redirection endpoint, once the authorization endpoint's interactions are completed between the resource owner and the authorization server. The client provides the redirection endpoint at the time of registration. The redirection endpoint must be an absolute URI and not contain a fragment component.

Securing Microservices



OAuth 2.0 grant types

The client requests an access token from the authorization server, based on the obtained authorization from the resource owner. The resource owner gives authorization in the form of an authorization grant. OAuth 2.0 defines four types of authorization grant:

- ✓ Authorization code grant
- ✓ Implicit grant
- ✓ Resource owner password credentials grant
- ✓ Client credentials grant

We will use The resource owner password credentials grant approach.

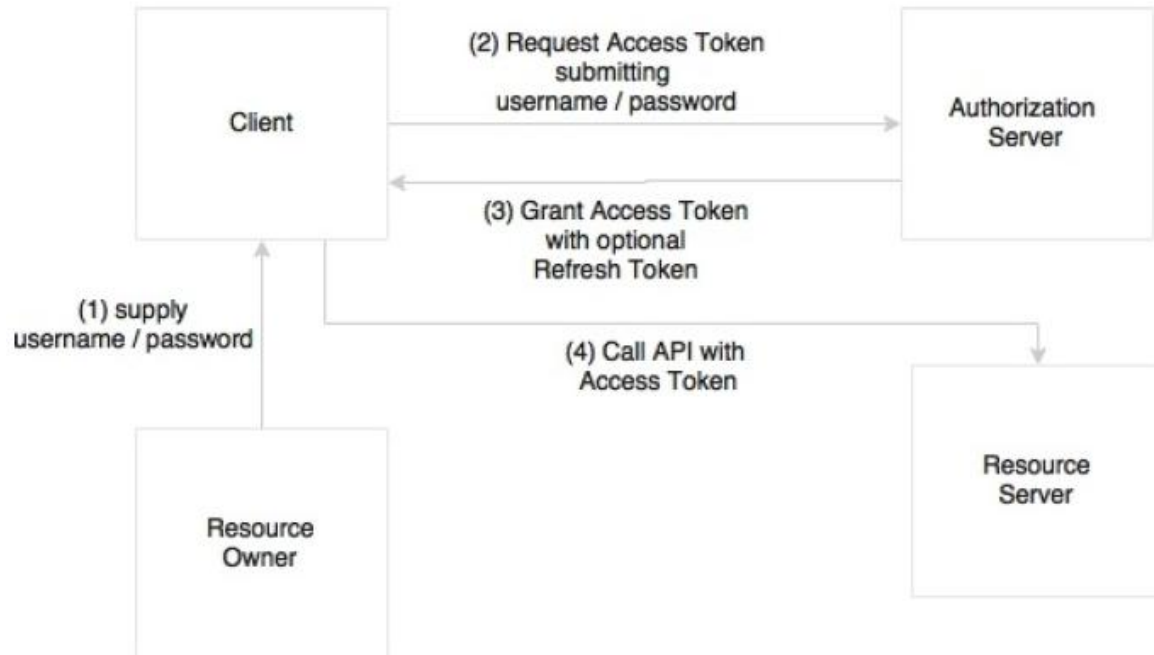
The resource owner password credentials grant approach

When a client application requires access to a protected resource, the client sends a request to an authorization server.

Securing Microservices

The authorization server validates the request and provides an access token.

This access token is validated for every client-to-server request. The request and response sent back and forth depends on the grant type.



As shown in the above picture,

Step-1: The resource owner provides the client with a username and password.

Step-2: The client then sends a token request to the authorization server by providing the credential information.

Step-3:

The authorization server authorizes the client and returns with an access token.

Step-4: On every subsequent request, the server validates the client token.

To implement OAuth2 in our application, we have to perform the following steps:

1. As a first step, update pom.xml with the OAuth2 dependency, as follows:

Securing Microservices

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.0.9.RELEASE</version>
</dependency>
```

2. Next, add two new annotations, `@EnableAuthorizationServer` and `@EnableResourceServer`, to the `Application.java` file.

The `@EnableAuthorizationServer`: annotation creates an authorization server with an in-memory repository to store client tokens and provide clients with a username, password, client ID, and secret.

The `@EnableResourceServer`: annotation is used to access the tokens. This enables a spring security filter that is authenticated via an incoming OAuth2 token.

In our application, both the authorization server and resource server are the same. However, in general, these two will run separately. Take a look at the following code:

```
@EnableResourceServer
@EnableAuthorizationServer
@SpringBootApplication
public class UserRegSystemApplication {

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

3. Add the following properties to the `application.properties` file:

```
server.port=9090
security.user.name=user
security.user.password=password
security.user.role= ADMIN

security.oauth2.client.clientId=trustedclient
security.oauth2.client.clientSecret=trustedclient123
security.oauth2.client.authorized-grant-types=authorization_code,refresh_token,password,client_credentials
security.oauth2.client.scope=apiAccess
```

```
server.port=9090

security.user.name=user

security.user.password=password
```

Securing Microservices

```
security.user.role= ADMIN  
security.oauth2.client.clientId=trustedclient  
security.oauth2.client.clientSecret=trustedclient123  
security.oauth2.client.authorized-grant-types:authorization_code,refresh_token,password,client_credentials  
security.oauth2.client.scope=apiAccess
```

4. Create an endpoint called /user in any controller or Application.java file to list the logged in user details by supplying the access token

```
@EnableResourceServer  
@EnableAuthorizationServer  
@RestController  
@SpringBootApplication  
public class UserRegSystemApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(UserRegSystemApplication.class, args);  
    }  
  
    @RequestMapping("/user")  
    public Principal user(Principal user) {  
        return user;  
    }  
}
```

```
@EnableResourceServer  
@EnableAuthorizationServer  
@RestController  
@SpringBootApplication  
public class UserRegSystemApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(UserRegSystemApplication.class, args);  
    }  
}
```

Securing Microservices

```
@RequestMapping("/user")

public Principal user(Principal user) {

    return user;

}

}
```

5. Restart the application and test it with the following grant_types

grant_type: client_credentials

Client_Credentials grant_type means the access token and refresh token will be generated on the client id and client secret.

To get the access token's for this grant_type, the request should look like the below

The screenshot shows a REST client interface with a POST request to `http://localhost:9090/oauth/token`. The 'Body' tab is selected, and the 'x-www-form-urlencoded' radio button is chosen. A table below lists the request parameters:

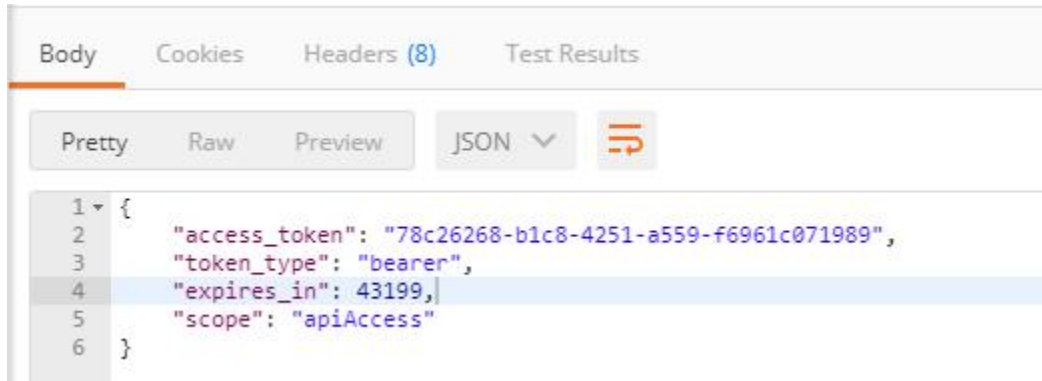
Key	Value
<input checked="" type="checkbox"/> grant_type	client_credentials

Also pass authorization header with the client details as

The screenshot shows a REST client interface with a POST request to `http://localhost:9090/oauth/token`. The 'Authorization' tab is selected, and the 'Basic Auth' dropdown is chosen. The 'Username' field contains 'trustedclient' and the 'Password' field contains 'trustedclient123'. A checkbox labeled 'Show Password' is checked. To the right, a note states: 'The authorization header will be generated and added as a custom header'. There is also an unchecked checkbox labeled 'Save helper data to request'.

Securing Microservices

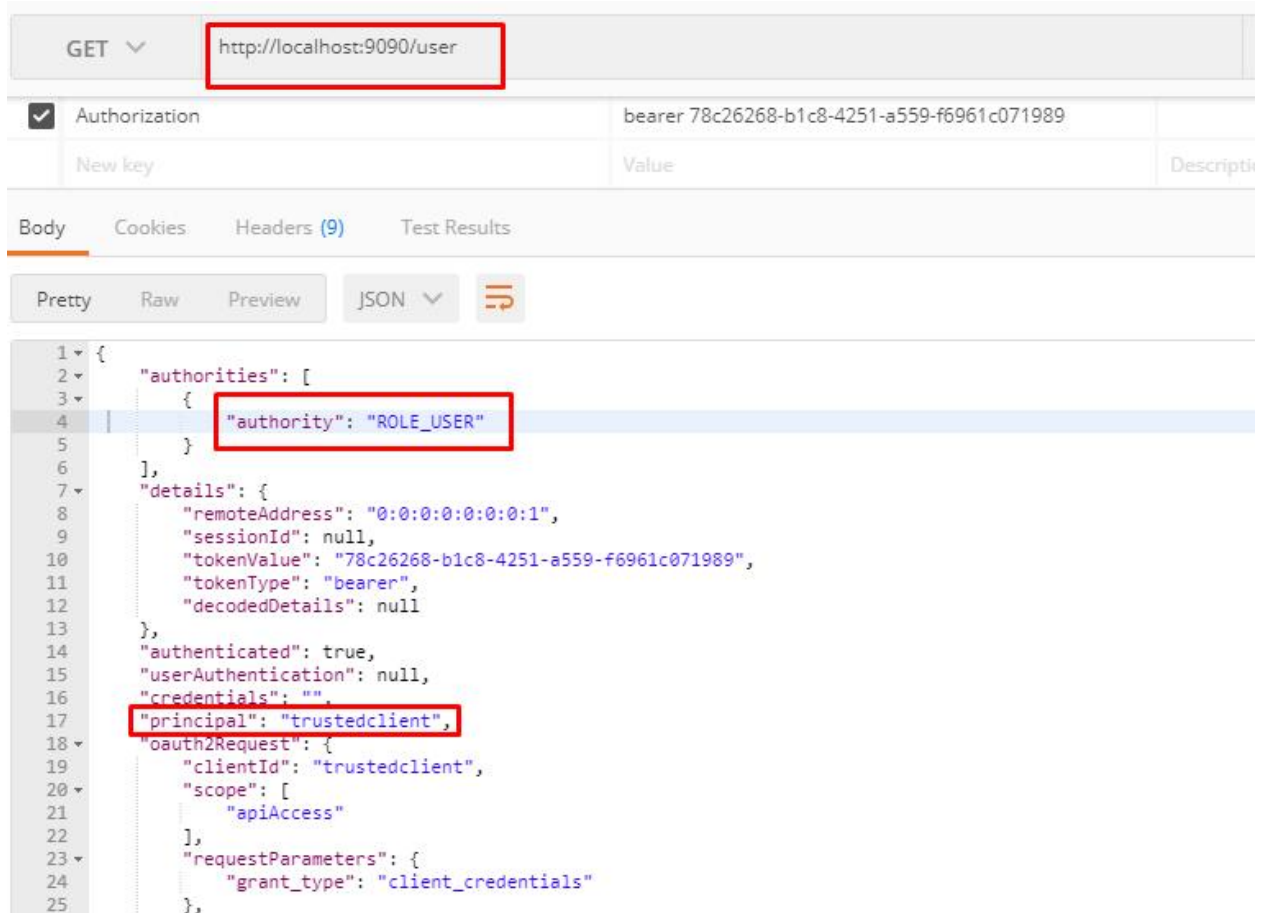
Then we will see the token generated by authorization server as shown below



The screenshot shows a REST client interface with tabs for Body, Cookies, Headers (8), and Test Results. The Body tab is selected, and the response is displayed in JSON format. The JSON object contains the following fields:

```
1 {
2   "access_token": "78c26268-b1c8-4251-a559-f6961c071989",
3   "token_type": "bearer",
4   "expires_in": 43199,
5   "scope": "apiAccess"
6 }
```

Now by supplying this token, we can get the logged in user details as shown below. Here in this grant_type client_id it self is the user



The screenshot shows a REST client interface with a GET request to `http://localhost:9090/user`. The request is authenticated using a Bearer token: `bearer 78c26268-b1c8-4251-a559-f6961c071989`. The response is displayed in JSON format. The JSON object contains the following fields:

```
1 {
2   "authorities": [
3     {
4       "authority": "ROLE_USER"
5     }
6   ],
7   "details": {
8     "remoteAddress": "0:0:0:0:0:0:1",
9     "sessionId": null,
10    "tokenValue": "78c26268-b1c8-4251-a559-f6961c071989",
11    "tokenType": "bearer",
12    "decodedDetails": null
13  },
14  "authenticated": true,
15  "userAuthentication": null,
16  "credentials": "",
17  "principal": "trustedclient",
18  "oauth2Request": {
19    "clientId": "trustedclient",
20    "scope": [
21      "apiAccess"
22    ],
23    "requestParameters": {
24      "grant_type": "client_credentials"
25    }
26  }
27 }
```

Securing Microservices

In general when any user is logged in, we have to generate an access token for that user and identify the user on the token, but this can not achieved using grant_type: client_credentials. Hence we go for grant_type: password.

grant_type: password

In grant_type: password, we have to supply the user credentials who wants to logged in to the application in the defined scope, on behalf of this user, client_id user will authenticates this user and provide the generated access_token to the user. Hence we have to supply client_id user details in header and the user credentials who wants to logged in to application, in the request body as shown below:

Header:

The screenshot shows a REST client interface with a POST request to `http://localhost:9090/oauth/token`. The 'Authorization' tab is selected. The request is configured with 'Basic Auth' type. The username is 'trustedclient' and the password is 'trustedclient123'. The 'Show Password' checkbox is checked. A note on the right states: 'The authorization header will be generated and added as a custom header'. There is also an unchecked checkbox for 'Save helper data to request'.

POST	http://localhost:9090/oauth/token	p
Authorization	Headers (1)	Body
Type: Basic Auth		
Username	trustedclient	The authorization header will be generated and added as a custom header
Password	trustedclient123	
<input checked="" type="checkbox"/> Show Password		<input type="checkbox"/> Save helper data to request

Body

Securing Microservices

POST http://localhost:9090/oauth/token

Authorization Headers (1) Body Pre-request Script Tests

☒ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary

	Key	Value
<input checked="" type="checkbox"/>	grant_type	password
<input checked="" type="checkbox"/>	scope	apiAccess
<input checked="" type="checkbox"/>	username	user
<input checked="" type="checkbox"/>	password	password

Response From authorization server

Body Cookies Headers (8) Test Results

Pretty Raw Preview JSON

```
1 {  
2   "access_token": "83223c57-5a9e-4672-b922-a0ed43fcffde",  
3   "token_type": "bearer",  
4   "refresh_token": "320ac3b5-1bf8-4e12-8037-a547e30ec00e",  
5   "expires_in": 42923,  
6   "scope": "apiAccess"  
7 }
```

Now by supplying this token, we can get the logged in user details as shown below.

Securing Microservices

The screenshot shows a REST client interface. At the top, a GET request is made to `http://localhost:9090/user`. Below the URL bar, the 'Authorization' header is set to 'bearer' followed by a token: `83223c57-5a9e-4672-b922-a0ed43fcffde`. The 'Body' tab is selected, showing a JSON response. The JSON is formatted as follows:

```
13 },
14 "authenticated": true,
15 "userAuthentication": {
16   "authorities": [
17     {
18       "authority": "ROLE_ADMIN"
19     }
20   ],
21   "details": {
22     "grant_type": "password",
23     "scope": "apiAccess",
24     "username": "user"
25   },
26   "authenticated": true,
27   "principal": {
28     "password": null,
29     "username": "user",
30     "authorities": [
31       {
32         "authority": "ROLE_ADMIN"
33       }
34     ]
35   }
36 }
```

grant_type: refresh_token

Some times, when the logged in user is doing any operations in the application, an access token might expires, Then we need to again get a new access token with an increased time limit. To refresh the token use the `grant_type` as `refresh_token` as shown below.

Header:

Always include an header with the `client_id` and `client_secret` as shown below

Securing Microservices

POST http://localhost:9090/oauth/token

Authorization Headers (1) Body Pre-request Script Tests

Type: Basic Auth

Username: trustedclient

Password:

☐ Show Password

The authorization header will be generated and added as a custom header

☐ Save helper data to request

Body:

Supply the grant_type and refresh token that we got when we logged in to the application first time as shown below:

POST http://localhost:9090/oauth/token

Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary

Key	Value
grant_type	refresh_token
refresh_token	320ac3b5-1bf8-4e12-8037-a547e30ec00e

Response from authorization server

Body Cookies Headers (8) Test Results

Pretty Raw Preview JSON

```
1 {
2   "access_token": "bab9c2db-7469-4a96-903c-6c98595121d9",
3   "token_type": "bearer",
4   "refresh_token": "320ac3b5-1bf8-4e12-8037-a547e30ec00e",
5   "expires_in": 43199,
6   "scope": "apiAccess"
7 }
```